DATABASE

DESIGN, APPLICATION DEVELOPMENT, & ADMINISTRATION

MICHAEL MANNINO

7e

DATABASE DESIGN, APPLICATION DEVELOPMENT, AND ADMINISTRATION

DATABASE DESIGN, APPLICATION DEVELOPMENT, AND ADMINISTRATION

SEVENTH EDITION

Michael V. Mannino
University of Colorado, Denver

CHICAGO
BUSINESS PRESS

CHICAGO
BUSINESS PRESS

DATABASE DESIGN, APPLICATION DEVELOPMENT, AND ADMINISTRATION
SEVENTH EDITION

For product information or assistance visit: www.chicagobusinesspress.com

ISBN-13: 978-1-948426-00-8
ISBN-10: 1-948426-00-5

BRIEF CONTENTS

CONTENTS

PREFACE

MOTIVATING EXAMPLE

Paul Hong, the owner of International Industrial Adhesives, Inc., is excited about potential opportunities in the growing global economy. He senses major opportunities in new product development, new sources of demand, and industry consolidation. These opportunities, however, involve substantial risks with major changes in his business and industry. He senses risk from new mergers and acquisitions, new competitors, increased government regulation and litigation in areas affecting his business, and data security threats. New mergers and acquisitions may involve challenges integrating disparate information technology and sharp increases in data and transaction volumes. The success of his business has attracted new competitors focusing on his most profitable customers and products. New environmental, financial, and health regulations impose costly data collection efforts, reporting requirements, and compliance activities. Data security breaches pose a constant threat especially with a large competitor having a recent, major disclosure of sensitive customer records. Despite tremendous opportunities for growth, he remains cautious about new directions to manage risk effectively.

Paul Hong must make timely and appropriate information technology investments to deal with strategic acquisitions, respond to competitors, control costs of government mandates, and thwart attacks on data assets. To manage mergers and acquisitions, he must increase information technology capacity to process large new volumes of transactions, manage increasing amounts of data for operations, business intelligence, and long-term archival storage, and integrate disparate systems and data. To match competitors, he needs more detailed and timely data about industry trends, competitors' actions, and intellectual property developments. To comply with new regulations, he must develop new data collection practices, conduct information technology audits, and fulfill other government reporting requirements for public companies. To thwart data attacks, he must review potential risks and invest in monitoring tools. For all of these concerns, he is unsure about managing risks, choosing information technology suppliers, and hiring competent staff.

These concerns involve significant usage of database technology as well as new data management initiatives to ensure accountability. New developments in NoSQL database technology, parallel processing architectures, and data lifecycle management can provide cost effective solutions to meet challenges of big data. These technologies can be deployed in cloud computing environments that provide economies of scale, elimination of fixed infrastructure costs, and dynamic scalability. A data governance organization can mitigate risks associated with the complex regulatory environment through a system of checks and balances using data rules and policies. Mergers and acquisitions often trigger data governance initiatives to ensure consistent data definitions and integrate corporate policies involving data privacy and security.

However, the solutions to Paul Hong's concerns involve more than technology. Utilization of appropriate information technology requires a vision for an organization's future, a deep understanding of technology, and traditional management skills to control risk. Paul Hong realizes that his largest challenge is to blend these skills to develop effective solutions for International Industrial Adhesives, Inc.

INTRODUCTION

This textbook provides a foundation to understand database technology supporting enterprise computing concerns such as those faced by Paul Hong. As a new student of database management, you first need to understand fundamental concepts of database management and the relational data model. Then you need to master skills in query formulation, database design, and database application development. This textbook provides tools to help you understand relational databases and acquire skills to solve basic and advanced problems in query formulation, data modeling, normalization, data requirements for business applications, and customization of database applications.

After establishing these skills, you are ready to study the organizational context, role of database specialists, and the processing environments in which databases are used. Students will learn about decision-making needs, accountability requirements, organization structures, business architectures, and roles of database specialists associated with databases and database technology. For environments, this textbook presents fundamental database technologies in each processing environment and relates these technologies to new advances in electronic commerce and business intelligence. You will learn vocabulary, architectures, and design issues of database technology that provide a background for advanced study of enterprise information systems, electronic commerce applications, and business intelligence.

WHAT'S NEW IN THE SEVENTH EDITION

The seventh edition makes substantial revisions to the sixth edition while preserving the proven pedagogy developed in the first sixth editions. Experience gained from instruction of university students and online learners along with feedback from adopters of the earlier editions has led to the development of new material and refinements to existing material. A five-course specialization developed for the Coursera platform in 2016 provided the impetus for substantial new material in the seventh edition about data warehouses.

The most significant changes in the seventh edition are a substantial expansion of data warehouse material and new coverage of NoSQL database technology and features. Many organizations focus on business intelligence to gain competitive advantage, manage risks, and connect with customers. Data warehouse technology and practices provide a foundation for business intelligence in many organizations. The seventh edition expands, updates, and reorganizes data warehouse material from two to four chapters. The seventh edition contains substantial new material about management of data warehouse development, data warehouses in major industries, the schema integration process, a mini case study about data warehouse design, SQL statements for data integration, data integration tools, SQL extensions for analytic queries, the Microsoft Multidimensional Expressions language, pivot table tools, and a business strategy game for managing development of data warehouses. Besides new material, the seventh edition substantially updates existing material in the sixth edition such as indicating the market decline of data warehouse appliances.

Organizations continue to face challenging demands from big data applications involving batch processing of large volumes of semi-structured data and online processing of intense levels of transactions. NoSQL database technology provides a foundation to deal with these big data applications in a growing number of organizations. The seventh edition contains substantial new material on NoSQL database technology about column-oriented storage, big data parallel processing architectures, and in-memory transaction processing. Both NoSQL DBMSs and enterprise relational DBMSs support these technologies. To understand explosive growth in NoSQL database DBMSs, the seventh edition provides an overview about features and data models in NoSQL DBMSs as well as detailed coverage of the JavaScript Object Notation (JSON) and the N1QL query language in Couchbase Server, a leading NoSQL DBMS.

Besides the expanded coverage of data warehouses and NoSQL database technology, the seventh edition provides numerous refinements to existing material based on classroom experience. Chapters 4 to 11 contain new examples in response to difficulties students had with textbook gaps. The seventh edition makes substantial revisions to coverage of data modeling tools, query formulation guidelines, normalization processes, and trigger coding guidelines. In addition, refinements and updates to most chapters have improved the presentation and currency of the material.

For database application development, the seventh edition covers SQL:2016, an evolutionary change from previous SQL standard versions (SQL:1999 to SQL:2011). The seventh edition explains the scope of SQL:2016, the difficulty of conformance with the standard, and new elements of the standard. Numerous refinements of details about database application development extend the proven coverage of the first sixth editions: query formulation guidelines, query formulation errors, count method for division problems, query formulation steps for hierarchal forms and reports, common errors in queries for forms, trigger formulation guidelines, and transaction design guidelines.

For database administration and processing environments, the seventh edition provides expanded coverage of NoSQL technology. The most significant new topics are columnstore indexes, in-memory transaction processing, and parallel processing architectures for big data applications.

In addition to new material and refinements to existing material, the seventh edition extends chapter supplements. The seventh edition contains new end-of-chapter questions and problems in most chapters. New material in the textbook's website includes detailed tutorials about Microsoft Access 2016, Visio Professional 2010, and Aqua Data Studio, assignments for first and second database courses, and sample exams. The software tutorials for Microsoft Access, Visio Professional, and Aqua Data Studio support concepts presented in textbook chapters 4, 5, 6, 9, and 10.

To make room for new material, the seventh edition eliminates two chapters from the sixth edition. The seventh edition contains two chapters of new material about data warehouses. New material about NoSQL technology replaces outdated material about object-oriented databases. To remove bloat, the seventh edition eliminates chapters covering a form-based approach for database design and a case study about database design. The course website contains these chapters for continuity with the sixth edition.

COMPETITIVE ADVANTAGES

This textbook provides outstanding features unmatched in competing textbooks. The unique features include detailed SQL coverage for both Microsoft Access and Oracle, problem-solving guidelines to aid acquisition of key skills, carefully designed sample databases and examples, advanced topic coverage, integrated lab material, prominent data modeling tools, extensive data warehouse details, and substantial NoSQL coverage. These features provide a complete package for both introductory and advanced database courses. The following list describes each feature in more detail while Table P-1 summarizes competitive advantages by chapter.

- *SQL Coverage*: The breadth and depth of the SQL coverage in this text is unmatched by competing textbooks. Table P-2 summarizes SQL coverage by chapter. Parts 2 and 5 provide thorough coverage of the CREATE TABLE, SELECT, UPDATE, INSERT, DELETE, CREATE VIEW, and CREATE TRIGGER statements. Part 6 provides extensive coverage of SQL statements for data warehouses. The chapters in parts 2 to 6 provide numerous examples of basic, intermediate, and advanced problems. The chapters in Part 7 cover statements useful for database administrators as well as statements used in specific processing environments.

- *Access and Oracle Coverage*: The chapters in Parts 2 and 5 provide detailed coverage of both Microsoft Access and Oracle SQL. Each example for the SELECT, INSERT, UPDATE, DELETE, and CREATE VIEW statements is shown for both DBMSs. Significant coverage of advanced Oracle 12c SQL features appears in Chapters 8, 9, 11, 14, 15, 17, and 19. In addition, the chapters in Parts 2 and 5 cover SQL:2016 syntax to support instruction with other prominent database management systems.
- *Problem-Solving Guidelines*: Students need more than explanations of concepts and examples to solve problems. Students need guidelines to help structure their thinking process to tackle problems in a systematic manner. The guidelines provide mental models to help students apply the concepts to solve basic and advanced problems. Table P-3 summarizes the unique problem-solving guidelines by chapter.
- *Sample Databases and Examples*: To provide consistency and continuity, Parts 2 to 5 use two sample databases in chapter bodies and problems. The University database is used in the chapter examples, while the Order Entry database is used in the end-of-chapter problems. Numerous examples and problems with these databases depict the fundamental skills of query formulation and application data requirements. Revised versions of the databases provide separation between basic and advanced examples. The website contains CREATE TABLE statements, sample data, data manipulation statements, and Access database files for both databases.

Chapters in Parts 3, 4, 6, and 7 use additional databases to broaden exposure to more diverse business situations. Students need exposure to a variety of business situations to acquire database design skills and understand concepts important to database specialists. The supplementary databases cover water utility operations, patient visits, academic paper reviews, personal financial tracking, airline reservations, placement office operations, automobile insurance, store sales tracking, and real estate sales. In addition, Chapter 12 on data warehouse concepts presents data warehouses in retail, education, and health care.

- *Optional Integrated Labs*: Database management is best taught when concepts are closely linked to the practice of designing, implementing, and using databases with a commercial DBMS. To help students apply the concepts described in the textbook, optional supplementary lab materials are available on the text's website. The website contains labs for five Microsoft Access versions (2003, 2007, 2010, 2013, and 2016) as well as practice databases and exercises. The Microsoft Access labs integrate a detailed coverage of Access with the application development concepts covered in Parts 2 and 5.
- *Data Modeling Tools*: The sixth edition expands coverage of commercial data modeling tools for database development. Students will find details about Aqua Data Studio, Oracle SQL Developer, and Visual Paradigm.
- *Data Warehouse Coverage*: The four data warehouse chapters (12 to 15) along with the database administration chapter provide details for an entire course on data warehouses in a business intelligence curriculum. No other competing textbook provides the breadth and depth of coverage about data warehouses. Chapter 12 presents data warehouse concepts and management with unique details about management of the data warehouse development process. Chapter 13 contains data warehouse design background with unique details about the schema integration process. Chapter 14 presents data integration concepts and tools with extensive coverage of data integration tools. Chapter 15 presents query formulation for data warehouses with extensive coverage of pivot table tools and SQL statement extensions. The course website contains assignments for pivot table tools, query formulation, data integration, schema integration, and materialized view processing to augment chapter coverage.

- *NoSQL Coverage*: Major organizations have strong demand for individuals with background about NoSQL technology and systems. The seventh edition supports this need with substantial material about features in NoSQL DBMSs as well as detailed coverage of a major NoSQL DBMS. Numerous examples and problems provide opportunity for students to obtain a foundation of skills for data modeling and query formulation using a NoSQL DBMS.

Due to the nature of NoSQL technology, the textbook distributes coverage across several chapters. Chapters 8, 17, and 18 present important technologies (column-oriented storage, in-memory transaction processing, big data parallel processing architectures, and BASE principle for distributed transaction processing) used in both NoSQL and enterprise relational DBMSs. Chapter 19 contains extensive details about features and data models used in NoSQL DBMSs. To provide practice-oriented coverage of NoSQL technology, Chapter 19 covers the Java Script Object Notation (JSON) and the Couchbase N1QL query language to manipulate JSON databases.

- *Current and Cutting-Edge Topics*: This textbook covers many topics omitted in competing textbooks: advanced query formulation, updatable views, development and management of stored procedures and triggers, hierarchical query formulation, business strategy game for managing data warehouse

TABLE P-1
Summary of Competitive Advantages by Chapter

Chapter	Unique Features
2	Conceptual introduction to the database development process
3	Visual representation of relational algebra operators
4	Query formulation guidelines; Errors in query formulation, Oracle, Access, and SQL:2016 SQL coverage
5	Emphasis on ERD notation, business rules, and diagram rules; Overview about data modeling notation in prominent commercial data modeling tools
6	Strategies for analyzing business information needs; Data modeling transformations; Detection of common design errors
7	Normalization guidelines and procedures
8	Index selection rules; SQL tuning guidelines, Integrated coverage of query optimization, file structures, and index selection
9	Query formulation guidelines; Oracle 12c, Access, and SQL:2016 coverage; Advanced topic coverage of nested queries, division problems, difference problems, null value handling, and hierarchical queries
10	Rules for updatable views; Data requirement steps for forms and reports; Common query formulation errors for hierarchical forms
11	Integrated coverage of database programming languages, stored procedures, and triggers; Trigger formulation guidelines; Common trigger coding errors
12	Management of data warehouse development; Business strategy game for data warehouse maturity; Examples of data warehouses in major industries
13	Building blocks for conceptual data warehouse design; Schema integration process
14	Data integration concepts, techniques, and tools; Supplementary material for data integration tool usage
15	Overview of Microsoft MDX and pivot table tools; Detailed coverage of SQL statement extensions for data warehouse queries
16	Guidelines to control trigger complexity, coding practices, and database dependencies; Data governance processes; Selection and evaluation process for a DBMS
17	Transaction design guidelines; Mini case study about transaction design
18	Integrated coverage of client-server processing, parallel database processing, and distributed databases integrated with impact of cloud computing
19	Object-relational features in SQL:2016 and Oracle 12c; NoSQL DBMS features; Query formulation using JSON documents and Couchbase N1QL

development, schema integration process, parallel database architectures, data integration tools, SQL extensions for data warehouses, in-memory transaction processing, object-relational features in SQL:2016 and Oracle 12c, and transaction

TABLE P-2
SQL Statement Coverage by Chapter

Chapter	SQL Statement Coverage
3	CREATE TABLE
4	SELECT, INSERT, UPDATE, DELETE; Access and Oracle coverage
9	SELECT (nested queries, outer joins, null value handling, hierarchical queries); Access and Oracle coverage
10	CREATE VIEW; retrieval and manipulation statements using views
11	CREATE PROCEDURE (Oracle), CREATE TRIGGER (Oracle and SQL:2016)
13	CREATE DIMENSION (Oracle)
14	MERGE (SQL:2016 and Oracle); Multiple table INSERT (Oracle)
15	SELECT statement extensions for subtotal computations and analytic functions (Oracle and SQL:2016); CREATE MATERIALIZED VIEW (Oracle) and query rewriting
16	GRANT, REVOKE, CREATE ROLE, CREATE ASSERTION, CHECK clause of the CREATE TABLE statement, CREATE DOMAIN
17	COMMIT, ROLLBACK, SET TRANSACTION, SET CONSTRAINTS, SAVEPOINT
19	CREATE TYPE, CREATE TABLE (typed tables and subtables), SELECT extensions (object identifiers, path expressions, dereference operator); SQL:2016 and Oracle coverage; Couchbase N1QL INSERT and SELECT statements for JSON databases

TABLE P-3
Problem-Solving Guidelines by Chapter

Chapter	Problem-Solving Guidelines
3	Visual representations of relationships and relational algebra operators
4	Conceptual evaluation process; Query formulation questions; Query formulation errors
5	Diagram rules
6	Guidelines for analyzing business information needs; Design transformations; Detection of common design errors; Conversion rules
7	Guidelines for identifying functional dependencies; Usage of sample data to falsify functional dependencies; Simple synthesis procedure
8	Index selection rules; SQL tuning guidelines
9	Difference problem formulation guidelines; Nested query evaluation; Count method for division problem formulation; Hierarchical query formulation guidelines
10	Rules for updatable queries; Steps for analyzing data requirements in forms and reports
11	Trigger execution procedure; Trigger formulation guidelines
12	Drivers of difficulties in data warehouse projects; Learning curve concepts; Architectures for data warehouse deployment; Architecture selection guidelines
13	Schema patterns with example data warehouse designs; Summarizability patterns; Steps of the schema integration process
14	Factors influencing refresh process objective; Features of data integration tools
15	Mapping of GROUPING SETS operator to CUBE and ROLLUP operators; Factors influencing analytic function extensions; Extension of SELECT statement processing for analytic functions; Comparison of traditional views and materialized views; Processing for materialized views; Matching requirements for query rewriting
16	Guidelines to manage stored procedures and triggers; Data planning process; DBMS selection process; Core processes and risk matrix in the Microsoft Data Governance Framework
17	Transaction timeline; Transaction design guidelines
18	Progression of transparency levels for distributed databases
19	Comparison between relational and object-relational representations; Multiple representations of JSON documents (fully nested, partially nested, and flat)

design principles. These topics enable motivated students to obtain a deeper understanding of database management.

- *Complete Package for Courses*: Depending on the course criteria, some students may need to purchase as many as four books for an introductory database course: a textbook covering principles, laboratory books covering details of a DBMS, a supplemental SQL book, and a casebook with realistic practice problems. This textbook and supplemental material provide a complete, integrated, and less expensive resource for students.

TEXT AUDIENCE

This book supports two database courses at the undergraduate or graduate level. At the undergraduate level, students should have a concentration (major or minor) or active interest in information systems. For two-year institutions, the instructor may want to skip advanced topics and place more emphasis on the optional Access lab book. Undergraduate students should have a first course covering general information systems concepts, spreadsheets, word processing, and possibly a brief introduction to databases.

At the graduate level, this book is suitable in either MBA or Master of Science (in information systems) programs. The advanced material in this book should be especially suitable for Master of Science students.

Except for Chapter 11, a previous course in computer programming can be useful background but is not mandatory. The other chapters reference some computer programming concepts, but writing code is not covered. For a complete mastery of Chapter 11, computer programming background is essential. However, the basic concepts and trigger details in Chapter 11 can be covered even if students do not have a computer programming background.

ORGANIZATION

As the title suggests, *Database Design, Application Development, and Administration* emphasizes three sets of skills. Before acquiring these skills, students need a foundation about basic concepts. Part 1 provides conceptual background for subsequent detailed study of database design, database application development, and database administration. The chapters in Part 1 present the principles of database management and a conceptual overview of the database development process.

Part 2 provides foundational knowledge about the relational data model. Chapter 3 covers table definition, integrity rules, and operators to retrieve useful information from relational databases. Chapter 4 presents guidelines for query formulation and numerous examples of SQL statements.

Parts 3 and 4 emphasize practical skills and design guidelines for the database development process. Students desiring a career as a database specialist should be able to perform each step of the database development process. Students should learn skills of data modeling, schema conversion, normalization, and physical database design. The Part 3 chapters (Chapters 5 and 6) cover data modeling using the Entity Relationship Model. Chapter 5 covers the structure of entity relationship diagrams, while Chapter 6 presents usage of entity relationship diagrams to analyze business information needs. The Part 4 chapters (Chapters 7 and 8) cover table design principles and practice for logical and physical design. Chapter 7 covers motivation, functional dependencies, normal forms, and practical considerations of data normalization. Chapter 8 contains broad coverage of physical database design including objectives, inputs, file structures, query optimization principles, and important design choices.

Part 5 provides a foundation for building database applications by helping students acquire skills in advanced query formulation, specification of data requirements

for data entry forms and reports, and coding triggers and stored procedures. Chapter 9 presents additional examples of intermediate and advanced SQL, along with corresponding query formulation skills. Chapter 10 describes motivation, definition, and usage of relational views along with specification of view definitions for data entry forms and reports. Chapter 11 presents concepts and coding practices of database programming languages, stored procedures, and triggers for customization of database applications.

Part 6 provides detailed coverage of data warehouse management, design, data integration, and query formulation. Chapter 12 presents basic concepts, management background, and examples of data warehouses in important industries. Chapter 13 describes conceptual design of data warehouses with coverage of multidimensional representation, schema patterns, summarizability patterns, and the schema integration process. Chapter 14 provides details about data integration concepts, techniques, and tools. Chapter 15 covers query formulation for online analytic processing, SQL SELECT statement extensions for subtotal calculations, SQL SELECT statement extensions for analytic functions, and summary data management. Together, the chapters in Part 6 provide a strong foundation for students to pursue a career as a data warehouse or business intelligence professional.

Beyond database design and application development skills, this textbook prepares students for careers as database specialists. Students need to understand the responsibilities, tools, and processes employed by data administrators and database administrators as well as the various environments in which databases operate.

The chapters in Part 7 emphasize the role of database specialists and the details of managing databases in various environments. Chapter 16 provides a context for the other chapters through coverage of the responsibilities, tools, and processes used by database administrators and data administrators. The other chapters in Part 7 provide a foundation for managing databases in important environments: Chapter 17 on transaction processing, Chapter 18 on distributed processing and data, and Chapter 19 on object and NoSQL databases. These chapters emphasize concepts, architectures, and design choices important for database specialists, while providing some coverage of advanced application development topics.

TEXT APPROACH AND THEME

To support acquisition of the necessary skills for learning and understanding application development, database design, and managing databases, this book adheres to three guiding principles.

(1) *Combine concepts and practice*. Database management is more easily learned when concepts are closely linked to the practice of designing and implementing databases using a commercial DBMS. The textbook and the accompanying supplements have been designed to provide close integration between concepts and practice through the following features:

- SQL examples for both Access and Oracle as well as SQL:2016 coverage
- Emphasis of the relationship between application development and query formulation
- Usage of data modeling notations supported by professional database development products
- Supplemental laboratory practice chapters that combine textbook concepts with details of commercial DBMSs

(2) *Emphasize problem-solving skills*. This book features problem-solving guidelines to help students master fundamental skills of data modeling, normalization, query formulation, and application development. The textbook and associated supplements provide a wealth of questions, problems, case studies, and laboratory practices in which students can apply their skills. With mastery of the

fundamental skills, students will be poised for future learning about databases and change the way they think about computing in general.

(3) *Provide introductory and advanced material*. Business students who use this book may have a variety of backgrounds. This book provides enough depth to satisfy the most eager students. However, the advanced parts are placed so that they can be skipped by the less inclined.

PEDAGOGICAL FEATURES

This book contains the following pedagogical features to help students navigate through chapter content in a systematic fashion:

- **Learning Objectives** focus on the knowledge and skills students will acquire from studying the chapter.
- **Overviews** provide a snapshot or preview of chapter contents.
- **Key Terms** are highlighted and defined in boxed areas as they appear in the chapter.
- **Examples** are clearly separated from the rest of the chapter material for easier review and studying purposes.
- **Running Database Examples** — examples using the University database as well as other databases with clear separation from surrounding text.
- **Closing Thoughts** summarize chapter content in relation to the learning objectives.
- **Review Concepts** are the important conceptual highlights from the chapter, not just a list of terminology.
- **Questions** are provided to review the chapter concepts.
- **Problems** help students practice and implement the detailed skills presented in the chapter.
- **References for Further Study** point students to additional sources on chapter content.
- **Chapter Appendixes** provide additional details, convenient summaries of SQL:2016 syntax, and other topics beyond normal chapter coverage.

At the end of the text, students will find the following additional resources:

- **Glossary**: Provides a complete list of terms and definitions used throughout the text.
- **Bibliography**: A list of helpful industry, academic, and other printed material for further research or study.
- **Index**: A list of keywords with page references to help readers of the printed edition.

MICROSOFT ACCESS LABS

Lab books for several versions of Microsoft Access (2003, 2007, 2010, 2013, and 2016) are available. The lab books provide detailed coverage of features important to beginning database students as well as many advanced features. The lab chapters provide a mixture of guided practice and reference material organized into the following chapters:

1. An Introduction to Microsoft Access
2. Database Creation Lab
3. Query Lab
4. Single Table Form Lab

5. Hierarchical Form Lab
6. Report Lab
7. Pivot Tables
8. User Interface Lab

Each lab chapter follows the pedagogy of the textbook with Learning Objectives, Overview, Closing Thoughts, Additional Practice exercises, and Appendixes of helpful tips. Most lab chapters reference concepts from the textbook for close integration with corresponding textbook chapters. Each lab book also includes a glossary of terms and an index.

INSTRUCTOR RESOURCES

A comprehensive set of supplements for the text and lab manuals is available to adopters.

- Powerpoint slides for each chapter
- Solutions to end of chapter problems for each chapter
- Solutions to end of chapter questions for each chapter
- Access databases for the university and order entry textbook databases
- Oracle SQL statements to create and populate the university and order entry textbook databases
- Files containing SQL statements used in the textbook chapters
- Case studies along with case study solutions
- Assignments used in a first database course. The assignments involve database creation, query formulation, application development with forms, data modeling, and normalization. In addition, a project assignment integrates material about database development and application development.
- Assignments used in a second database course on database administration and processing environments. The assignments involve database creation, triggers, data warehouse design, data integration practices, query formulation for data warehouses, and object relational databases. In addition, projects are provided about Oracle advanced features, benchmark development, and management practices to develop or manage a significant database or data warehouse in an organization.
- Assignments used in a second course on data warehouses. The assignments involve pivot table tools, data warehouse design, SELECT statement extensions for subtotal operators, SELECT statement extensions for analytic functions, and materialized views and query rewriting. A comprehensive case study integrating other parts of the course can be used in the last part of the course.
- Sample exams for a first course in database management
- Sample exams for an advanced course in database management
- Access databases for each lab chapter
- Access databases for end of chapter problems in each lab chapter

TEACHING PATHS

The textbook can be covered in several orders in a one- or a two-semester sequence. The author has taught a one-semester course with the ordering of relational database basics, query formulation, application development, database development, and database processing environments. This ordering has the advantage of covering the more concrete material (query formulation and application development) before the more conceptual material (database development). Lab chapters and assignments are used for practice beyond the textbook chapters. To fit into one semester, advanced topics are skipped in Chapters 8 and 11 to 19.

A second ordering is to cover database development before application development. For this ordering, the author recommends the following textbook chapter ordering: 1, 2, 5, 6, 3, 7, 4, 9, and 10. The material on schema conversion in Chapter 6 should be covered after Chapter 3. This ordering supports a more thorough coverage of database development while not neglecting application development. To fit into one semester, advanced topics are skipped in Chapters 8 and 11 to 19.

A third possible ordering is to use the textbook in a two-course sequence. The first course covers database management fundamentals from Parts 1 and 2, data modeling and normalization from Parts 3 and 4, and advanced query formulation, and application development with views. The second course emphasizes database administration skills with physical database design from Part 4, triggers and stored procedures from Part 5, and the processing environments from Part 7 along with Chapter 12 on data warehouses. A comprehensive project can be used in the second course to integrate application development, database development, and database administration.

An alternative second course covers data warehouses and database administration. This course fits well in a business intelligence track. This course uses the four chapters in Part 6 about data warehouses and selected parts of Chapter 16 on data and database administration. A detailed case study can be used to provide integrative material in the last part of the course.

ACKNOWLEDGMENTS

The seventh edition culminates 40 years of instruction, research, and industry experience. Before beginning the first edition, I wrote tutorials, laboratory practices, and case studies. This material was first used to supplement other textbooks. After encouragement from students, this material was used without a textbook. This material, revised many times through student comments, was the foundation for the first edition. During the development of the first edition, the material was classroom tested for three years with hundreds of undergraduate and graduate students, along with careful review through four drafts by many outside reviewers. The second edition was developed through classroom usage of the first edition for three years, along with teaching an advanced database course for several years. The third edition was developed through experience of three years with the second edition in basic and advanced database courses. The fourth edition was developed through three years of instruction with the third edition in beginning and advanced database courses. The fifth edition was developed through two years of instruction with the fourth edition in beginning and advanced database courses. The sixth edition was developed through two years of instruction with the fifth edition in beginning and advanced database courses. The seventh edition was developed through three years of instruction with the sixth edition as well as development of a Coursera specialization on data warehousing.

I wish to acknowledge the excellent support that I have received in completing the seventh edition. I thank my publisher, Paul Ducham, for his expertise in producing and marketing this textbook. Without his expertise, this textbook would not reach its intended audience. I thank my many database students, especially those in ISMG6080 and ISMG6480 at the University of Colorado Denver and more than 50,000 learners in the Coursera specialization, *Data Warehousing for Business Intelligence*. Your comments and reaction to my courses and the textbook have been invaluable to its improvement.

ABOUT THE AUTHOR

Michael V. Mannino has been involved in the database field since 1978 when he began graduate studies at the University of Arizona. He has taught database management since 1983 at several major universities (University of Florida, University of Texas at Austin, University of Washington, and University of Colorado Denver). His audiences have included undergraduate MIS students, graduate MIS students, MBA students, and doctoral students as well as thousands of Coursera learners. He has also been active in database research with publications in major journals of the IEEE (*Transactions on Knowledge and Data Engineering* and *Transactions on Software Engineering*), ACM (*Communications, Journal of Data and Information Quality, Transactions on Management Information Systems*, and *Computing Surveys*), and INFORMS (*Informs Journal on Computing* and *Information Systems Research*). His research includes several popular survey and tutorial articles as well as many papers presenting original research. Practical results of his research have been incorporated into Chapter 12 on management of the data warehouse development, especially the design and implementation of the educational game, Emerge2Maturity, for providing insights about data warehouse maturity in organizations.

part one

Introduction to Database Environments

Part 1 provides a background for subsequent detailed study of database design, database application development, and database administration. The chapters in Part 1 present the principles of database management and the nature of the database development process. Chapter 1 covers the basic concepts of database management including database characteristics, features and architectures of database management systems, the market for database management systems, and organizational impacts of database technology. Chapter 2 introduces the context, objectives, phases, and tools of the database development process.

chapter

1

Introduction to Database Management

Learning Objectives

This chapter provides an introduction to database technology and the impact of this technology on organizations. After this chapter the student should have acquired the following knowledge and skills:

- Describe the characteristics of business databases and the features of database management systems
- Understand the importance of nonprocedural access for software productivity
- Appreciate the advances in database technology and the contributions of database technology to modern society
- Understand the impact of database management system architectures on distributed processing and software maintenance
- Perceive career opportunities related to database application development and database administration

OVERVIEW

You may not be aware of it, but database technology dramatically affects your life. Modern organizations cannot operate efficiently without databases and associated database technology. You come into contact with databases on a daily basis through activities such as shopping at a supermarket, withdrawing cash using an automated teller machine, making an airline reservation, ordering a book online, and registering for classes. The proliferation of databases and supporting database technology provides convenience in your daily life.

Database technology is not only improving the daily operations of organizations but also the quality of decisions that affect our lives. Databases contain a flood of data about many aspects of our lives such as consumer preferences, telecommunications usage, credit history, television viewing habits, and taxation documents. Database technology helps to summarize this mass of data into useful information for decision-making. Management uses information gleaned from databases to make long-range decisions such as investing in plants and equipment, locating stores, adding new items to inventory, and entering new businesses. Government uses information mined from databases to target taxation enforcement, refine pollution control efforts, target interest groups for election appeals, and develop new laws.

This first chapter provides a starting point for your exploration of database technology. It surveys database characteristics, database management system features, system architectures, and human roles in managing and using databases. The other chapter in Part 1 (Chapter 2) provides a conceptual overview of the database development process. Chapter 1 provides a broad picture of database technology and shares the excitement about the journey ahead.

1.1 DATABASE CHARACTERISTICS

Database
a collection of persistent data that can be shared and interrelated.

Every day, businesses collect mountains of facts about persons, things, and events such as credit card numbers, bank balances, and purchase amounts. **Databases** contain these types of simple facts as well as nonconventional facts such as medical images, fingerprints, product photos, and maps. With the proliferation of the Internet and the means to capture data in digital format, a vast amount of data is available at the click of a mouse button. Organizing these data for ease of retrieval and maintenance is paramount. Thus, managing databases has become a vital task in most organizations.

Before learning about managing databases, you must first understand some important properties of databases, as discussed in the following list:

- Persistent means that data reside on stable storage such as a magnetic disk. For example, organizations need to retain data about customers, suppliers, and inventory on stable storage because these data are repetitively used. A variable in a computer program is not persistent because it resides in main memory and disappears after the program terminates. Persistency does not mean that data lasts forever. When data are no longer relevant (such as a supplier going out of business), they are removed or archived.

  Persistency depends on relevance of intended usage. For example, the mileage you drive for work is important to maintain if you are self-employed. Likewise, the amount of your medical expenses is important if you can itemize your deductions or you have a health savings account. Because storing and maintaining data is costly, only data likely to be relevant for actions and decisions should be stored.
- Shared means that a database can have multiple uses and users. A database provides a common memory for multiple functions in an organization. For example, a personnel database can support payroll calculations, performance evaluations, government reporting requirements, and so on. Many users can access a database at the same time. For example, many customers can simultaneously make airline reservations. Unless two users are simultaneously trying to change the same data, they can proceed without waiting on each other.
- Interrelated means that data stored as separate units can be connected to provide a whole picture. For example, a customer database relates customer data (name, address, …) to order data (order number, order date, …) to facilitate order processing. Databases contain both entities and relationships among entities. An entity is a cluster of data usually about a single subject that can be accessed together. An entity can denote a person, place, thing, or event. For example, a personnel database contains entities such as employees, departments, and skills as well as relationships showing employee assignments to departments, skills possessed by employees, and salary history of employees. A typical business database may have hundreds of types of entities and relationships.

To depict these characteristics, let us consider a number of databases. We begin with a simple university database (Figure 1.1) since you have some familiarity with the workings of a university. A simplified university database contains data about students, faculty, courses, course offerings, and enrollments. The database supports procedures such as registering for classes, assigning faculty to course offerings, recording grades, and scheduling course offerings. Relationships in the university database support answers to questions such as

- What offerings are available for a course in a given academic period?
- Who is the instructor for an offering of a course?
- What students are enrolled in an offering of a course?

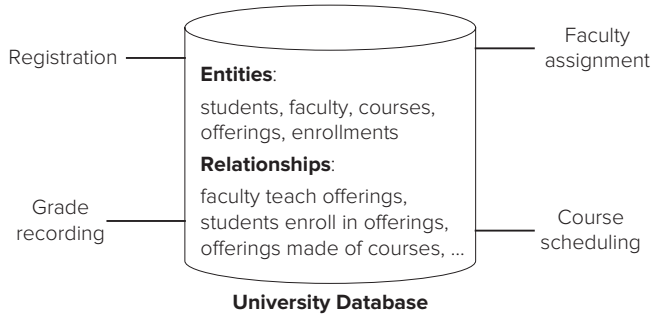


FIGURE 1.1
Depiction of a Simplified University Database

Note: Words surrounding the database denote processes that use the database.

Next, let us consider a water utility database as depicted in Figure 1.2. The primary function of a water utility database is billing customers for water usage. Periodically, the water utility measures a customer's water consumption from a meter and generates a bill. Many aspects can influence the preparation of a bill such as a customer's payment history, meter characteristics, type of customer (low income, renter, homeowner, small business, large business, etc.), and billing cycle. Relationships in the water utility database support answers to questions such as

- What is the date of the last bill sent to a customer?
- How much water usage was recorded when a customer's meter was last read?
- When did a customer make his/her last payment?

Finally, let us consider a hospital database as depicted in Figure 1.3. The hospital database supports treatment of patients by physicians. Many different health providers read and contribute to a patient's medical record. Physicians make diagnoses and prescribe treatments based on symptoms. Nurses monitor symptoms and provide medication. Food staff prepares meals according to dietary plans. Relationships in the database support answers to questions such as

- What are the most recent symptoms of a patient?
- Who prescribed a given treatment of a patient?
- What diagnosis did a doctor make for a patient?

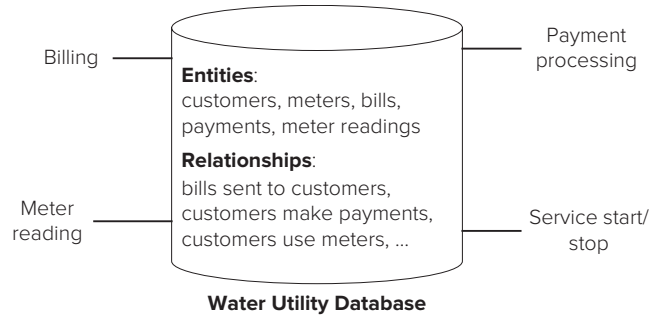


FIGURE 1.2
Depiction of a Simplified Water Utility Database

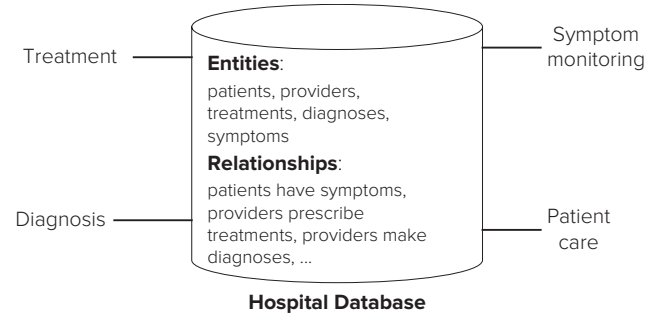


FIGURE 1.3
Depiction of a Simplified Hospital Database

These simplified databases lack many kinds of data found in real databases. For example, the simplified university database does not contain data about course prerequisites and classroom capacities and locations. Real versions of these databases would have many more entities, relationships, and additional uses. Nevertheless, these simple databases have the essential characteristics of business databases: persistent data, multiple users and uses, and multiple types of entities connected by relationships.

1.2 FEATURES OF DATABASE MANAGEMENT SYSTEMS

Database Management System (DBMS)
a collection of components that support data acquisition, dissemination, maintenance, retrieval, and formatting.

A **database management system** (DBMS) is a collection of components that supports the creation, use, and maintenance of databases. Initially, DBMSs provided efficient storage and retrieval of data. Due to marketplace demands and product innovation, DBMSs have evolved to provide a broad range of features for data acquisition, storage, dissemination, maintenance, retrieval, and formatting. The evolution of these features has made DBMSs rather complex. It can take years of study and use to master a particular DBMS. Because DBMSs continue to evolve, you must continually update your knowledge.

To provide insight about features that you will encounter in commercial DBMSs, Table 1-1 summarizes a common set of features. The remainder of this section presents examples of these features. Some examples are drawn from Microsoft Access, a popular desktop DBMS and Oracle, a prominent enterprise DBMS. Later chapters expand upon the introduction provided here.

1.2.1 Database Definition

Table
a named, two-dimensional arrangement of data. A table consists of a heading part and a body part.

SQL
an industry standard database language that includes statements for database definition, database manipulation, and database control.

To define a database, a database designer specifies entities and relationships. In most commercial DBMSs, **tables** store collections of entities. A table (Figure 1.4) has a heading row (first row) showing the column names and a body (other rows) showing the contents of the table. Relationships indicate connections among tables. For example, the relationship connecting the student table to the enrollment table shows the course offerings taken by each student.

Most DBMSs provide several tools to define databases. The Structured Query Language (SQL) is an industry standard language supported by most DBMSs. **SQL** can be used to define tables, relationships among tables, integrity constraints (rules that define allowable data), and authorization rights (rules that restrict access to data). Chapter 3 describes SQL statements to define tables and relationships.

In addition to SQL, many DBMSs provide graphical, window-oriented tools. Figures 1.5 and 1.6 depict graphical tools for defining tables and relationships. Using the Table Definition window in Figure 1.5, a user can define properties of columns such as the data type, field size, and format. Using the Relationship Definition window

TABLE 1-1
Summary of Common Features of DBMSs

Feature	Description
Database definition	Language and graphical tools to define entity types, relationships, integrity constraints, and authorization rights
Nonprocedural access	Language and graphical tools to access data without complicated coding
Application development	Graphical tools to develop menus, data entry forms, and reports; data requirements for forms and reports are specified using nonprocedural access
Procedural language interface	Language that combines nonprocedural access with full capabilities of a programming language such as Java or Javascript
Transaction processing	Control mechanisms to prevent interference from simultaneous users and recover lost data after a failure
Database tuning	Tools to monitor and improve database performance

StdFirstName	StdLastName	StdCity	StdState	StdZip	StdMajor	StdClass	StdGPA
HOMER	WELLS	SEATTLE	WA	98121-1111	IS	FR	3.00
BOB	NORBERT	BOTHELL	WA	98011-2121	FIN	JR	2.70
CANDY	KENDALL	TACOMA	WA	99042-3321	ACCT	JR	3.50
WALLY	KENDALL	SEATTLE	WA	98123-1141	IS	SR	2.80
JOE	ESTRADA	SEATTLE	WA	98121-2333	FIN	SR	3.20
MARIAH	DODGE	SEATTLE	WA	98114-0021	IS	JR	3.60
TESS	DODGE	REDMOND	WA	98116-2344	ACCT	SO	3.30

FIGURE 1.4
Display of Student Table in Microsoft Access

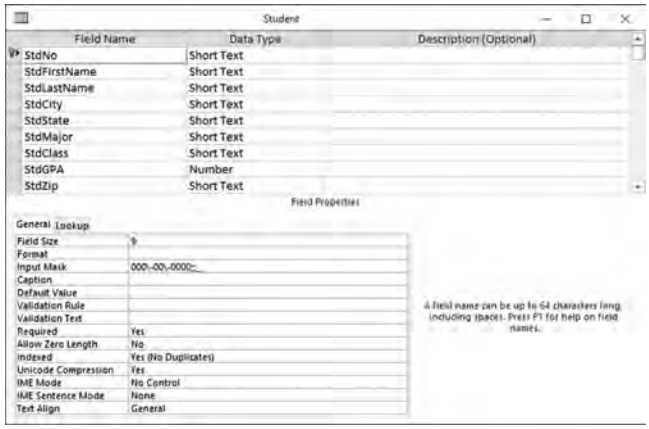


FIGURE 1.5
Table Definition Window in Microsoft Access

in Figure 1.6, relationships among tables can be defined. After defining the structure, a database can be populated. The data in Figure 1.4 should be added after the Table Definition window and Relationship Definition window are complete.

1.2.2 Nonprocedural Access

Nonprocedural Database Language
a language such as SQL that allows you to specify the parts of a database to access rather than to code a complex procedure. Nonprocedural languages do not include looping statements.

The most important feature of a DBMS is the ability to answer queries. A query is a request for data to answer a question. For example, the user may want to know customers having large balances or products with strong sales in a particular region. **Nonprocedural** access allows users with limited computing skills to submit queries. The user specifies the parts of a database to retrieve, not implementation details of how retrieval occurs. Implementation details involve coding complex procedures with

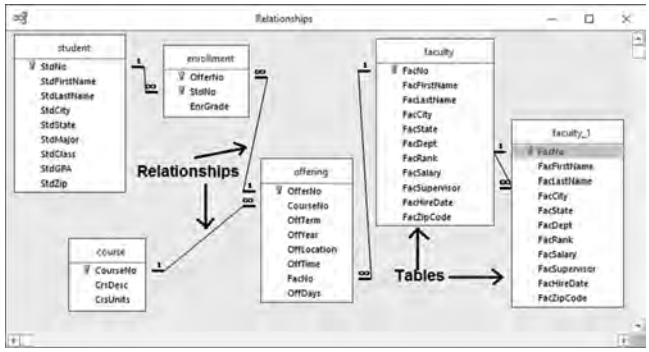


FIGURE 1.6
Relationship Definition Window in Microsoft Access

loops. Nonprocedural languages do not have looping statements (for, while, and so on) because only the parts of a database to retrieve are specified.

Nonprocedural access can reduce the number of lines of code by a factor of 100 as compared to procedural access. Because a large part of business software involves data access, nonprocedural access can provide a dramatic improvement in software productivity.

To appreciate the significance of nonprocedural access, consider an analogy to planning a vacation. You specify the destination, travel budget, length of stay, and departure date. These facts indicate the "what" of your trip. To specify the "how" of your trip, you need a detailed plan with details about the best route to your destination, the most desirable hotel, ground transportation, and so on. A planning professional can facilitate your planning process by completing these details. Like a planning professional, a DBMS performs the detailed planning to answer queries expressed in a nonprocedural language.

Most DBMSs provide more than one tool for nonprocedural access. The SELECT statement of SQL, presented in Chapter 4, provides a nonprocedural way to access a database. Most DBMSs also provide graphical tools to access databases. Figure 1.7 depicts a graphical tool available in Microsoft Access. To pose a query using the database, a user only indicates the required tables, relationships, and columns. Access generates the plan to retrieve the requested data. Figure 1.8 shows the result of executing the query in Figure 1.7.

1.2.3 Application Development and Procedural Language Interface

Most DBMSs go well beyond simply accessing data. DBMSs provide graphical tools for building complete applications using forms and reports. Data entry forms support convenient data entry and display, while reports enhance the appearance of data that is displayed or printed. The form in Figure 1.9 can be used to add new course assignments for a professor and to change existing assignments. The report in Figure 1.10 uses indentation to show courses taught by faculty in various departments.

FIGURE 1.7
Query Design Window in Microsoft Access

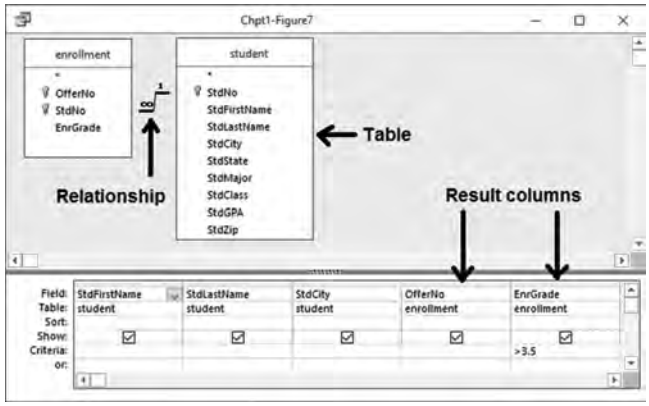


FIGURE 1.8
Result of Executing Query in Figure 1.7

StdFirstName	StdLastName	StdCity	OfferNo	EnrGrade
MARIAH	DODGE	SEATTLE	1234	3.8
BOB	NORBERT	BOTHELL	5679	3.7
ROBERTO	MORALES	SEATTLE	5679	3.8
MARIAH	DODGE	SEATTLE	6666	3.6
LUKE	BRAZZI	SEATTLE	7777	3.7
WILLIAM	PILGRIM	BOTHELL	9876	4

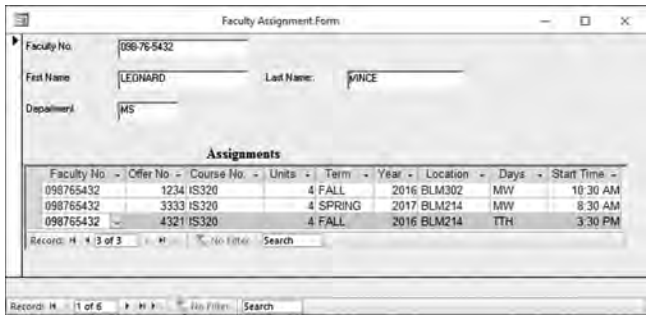


FIGURE 1.9
Microsoft Access Form for Assigning Courses to Faculty

Faculty Work Load Report for the 2016-2017 Academic Year

Department Name	*Term*	*Offer Number*	*Units*	*Limit*	*Enrollment*	*Percent Full*	*Low Enrollment*
FIN							
JULIA MILLS							
	WINTER	5678	4	20	1	5.00%	☑
	Summary for term' = WINTER (1 detail record)						
	Sum		*4*		*1*		
	Avg					*5.00%*	
Summary for JULIA MILLS							
Sum			*4*		*1*		
Avg						*5.00%*	
Summary for 'department' = FIN (1 detail record)							
Sum			*4*		*1*		
Avg						*5.00%*	

FIGURE 1.10
Microsoft Access Report of Faculty Workload

The indentation style can be easier to view than the tabular style shown in Figure 1.8. Many forms and reports can be developed with a graphical tool without detailed coding. For example, Figures 1.9 and 1.10 were developed without coding. Chapter 10 describes concepts underlying form and report development.

Nonprocedural access makes form and report creation possible without extensive coding. As part of creating a form or report, the user indicates the data requirements using a nonprocedural language (SQL) or graphical tool. To complete a form or report definition, the user indicates formatting of data, user interaction, and other details.

In addition to application development tools, a **procedural language interface** adds the full capabilities of a computer programming language. Nonprocedural access and application development tools, though convenient and powerful, are sometimes not efficient enough or do not provide the level of control necessary for application development. When these tools are not adequate, DBMSs provide the full capabilities of a programming language. Most commercial DBMSs have a procedural language interface. For example, Oracle has the language PL/SQL and Microsoft SQL Server has the language Transact-SQL. Chapter 11 describes procedural language interfaces and the PL/SQL language.

Procedural Language Interface
a method to combine a nonprocedural language such as SQL with a programming language such as Java or Visual Basic.

1.2.4 Features to Support Database Operations

Transaction processing enables a DBMS to process large volumes of repetitive work. A **transaction** is a unit of work that should be processed reliably without interference from other users and without loss of data due to failures. Examples of transactions are withdrawing cash at an ATM, making an airline reservation, and registering for a course. A DBMS ensures that transactions are free of interference from other users,

Transaction Processing
reliable and efficient processing of large volumes of repetitive work. DBMSs ensure that simultaneous users do not interfere with each other and failures do not cause lost work.

parts of a transaction are not lost due to a failure, and transactions do not make the database inconsistent. Transaction processing is largely an unseen, back-office affair. The user does not know the details about transaction processing other than the assurances about reliability.

Database tuning involves components to monitor and improve performance. Some DBMSs can monitor database performance and generate events indicating conditions that may warrant investigation. DBMSs provide components to improve performance such as reorganization of a database, selection of physical structures, and repair of damaged parts of a database.

Transaction processing and database tuning are most prominent on DBMSs that support large databases with many simultaneous users. These DBMSs are known as enterprise DBMSs, designed to support databases that are critical to the functioning of an organization. Enterprise DBMSs usually run on powerful servers and have a high cost. In contrast, desktop DBMSs running on personal computers and small servers support limited transaction processing features but have a much lower cost. Desktop DBMSs support databases used by work teams and small businesses. Embedded DBMSs are an emerging category of database software. As its name implies, an embedded DBMS resides in a larger system, either an application or a device such as a personal digital assistant (PDA) or a smart phone. Embedded DBMSs provide limited transaction processing features but have low memory, processing, and storage requirements.

1.2.5 Third-Party Features

In addition to features provided directly by vendors of DBMSs, third-party software is also available for many DBMSs. In most cases, third-party software extends the features available with the database software. For example, many third-party vendors provide advanced database design tools that extend the database definition and tuning capabilities provided by DBMSs. Figure 1.11 shows a database diagram (an entity relationship diagram) created with Visio Professional, a tool for database design. The ERD in Figure 1.11 can be converted into the tables supported by most commercial DBMSs. In some cases, third-party software competes directly with the database product. For example, third-party vendors provide application development tools that can be used in place of the ones provided with the database product.

1.3 DEVELOPMENT OF DATABASE TECHNOLOGY AND MARKET STRUCTURE

The previous section provided a quick tour of the features found in typical DBMSs. The features in today's products are a significant improvement over just a few years ago. Database management, like many other areas of computing, has undergone

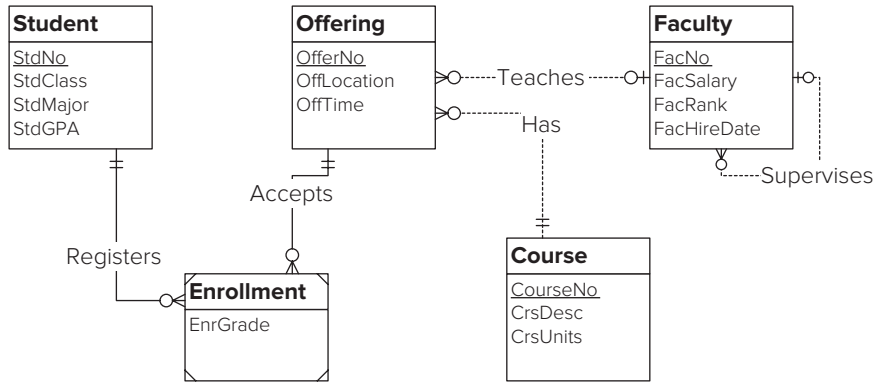


FIGURE 1.11
Entity Relationship Diagram (ERD) for the University Database

tremendous technological growth. To provide a context to appreciate today's DBMSs, this section reviews past changes in technology and suggests future trends. After this review, the current market for database software is presented.

1.3.1 Evolution of Database Technology

Table 1-2 depicts a brief history of database technology through four generations[1] of systems. The first generation supported sequential and random searching, but the user was required to write a computer program to obtain access. For example, a program could be written to retrieve all customer records or to just find the customer record with a specified customer number. Because first-generation systems did not offer much support for relating data, they are usually regarded as file processing systems rather than DBMSs. File processing systems can manage only one entity type rather than many entity types and relationships managed by a DBMS.

The second-generation products were the first true DBMSs as they could manage multiple entity types and relationships. However, to obtain access to data, a computer program still had to be written. Second-generation systems are referred to as "navigational" because the programmer had to write code to navigate among a network of linked records. Some of the second-generation products adhered to a standard database definition and manipulation language developed by the Committee on Data Systems Languages (CODASYL), a standards organization. The CODASYL standard had only limited market acceptance partly because IBM, the dominant computer company during this time, ignored the standard. IBM supported a different approach known as the hierarchical data model.

Rather than focusing on the second-generation standard, research labs at IBM and academic institutions developed the foundations for a new generation of DBMSs. The most important development involved nonprocedural languages for database access. Third-generation systems are known as relational DBMSs because of the foundation based on mathematical relations and associated operators. Optimization technology was developed so that access using nonprocedural languages would be efficient. Because nonprocedural access provided such an improvement over navigational access, third-generation systems supplanted the second generation. Since the technology was so different, most of the new systems were founded by start-up companies rather than by vendors of previous generation products. IBM was the major exception. It was IBM's weight that led to the adoption of SQL as a widely accepted standard.

Fourth-generation DBMSs have extended the boundaries of database technology to unconventional data, new kinds of distributed processing, data warehouse processing, and big data demands especially with semi-structured data. As an early emphasis, fourth-generation DBMSs provided support for unconventional data types such as images, videos, maps, sounds, animations, and web pages. Most DBMSs now feature

TABLE 1-2
Brief Evolution of Database Technology

Era	Generation	Orientation	Major Features
1960s	1st generation	File	File structures and proprietary program interfaces
1970s	2nd generation	Network navigation	Networks and hierarchies of related records, standard program interfaces
1980s	3rd generation	Relational	Nonprocedural languages, optimization, transaction processing
1990s to 2010s	4th generation	Object	Multi-media, active, distributed processing, more powerful operators, data warehouse processing, XML enabled, cloud computing, big data demands, semi-structured data

[1]The generations of DBMSs should not be confused with the generations of programming languages. In particular, fourth-generation language refers to programming language features, not DBMS features.

convenient ways to publish static and dynamic web pages using the eXtensible Markup Language (XML) as a publishing standard. Because these DBMSs view any kind of data as an object to manage, fourth-generation systems were called object-relational.

In the last 15 years, DBMS vendors have extended their fourth-generation products for data warehouse processing. A data warehouse is a database that supports mid-range and long-range decision making in organizations. The retrieval of summarized data dominate data warehouse processing, whereas a mixture of updating and retrieving data occur for databases that support the daily operations of an organization. Part 6 provides four chapters about data warehouse concepts and DBMS features to support data warehouse processing.

Cloud computing is a recent area of product development for both established DBMS vendors and new vendors. Cloud computing supports on-demand and pay-per use access for both data and software. Cloud computing usage is web-based without fixed costs of software ownership. Major DBMS vendors have developed cloud computing models as an alternative to their traditional approach of product licensing and ownership. In addition, a number of new vendors have created DBMS products tailored to the cloud computing model.

Part of the promise of cloud computing is support for applications with exploding data growth known as big data. The growth in data comes from a variety of sources such as sensors in smart phones, energy meters, and automobiles, interaction of individuals in social media websites, radio frequency identification tags in retail, and digitized media content in medicine, entertainment, and security. Big data exceeds the limits of commercial database software to support applications with exploding data growth.

NoSQL (Not only SQL) database technology has been developed to deal with some of the challenges of big data. As the name implies, NoSQL database technology does not use the traditional relational database model and SQL standard. Instead NoSQL database products use simplified database models, less stringent transaction processing models, and distributed processing to reduce bottlenecks for dealing with big data. NoSQL products cover a wide range of data models to support management of semi-structured data with key-record pairs, documents, and graphs.

The market for fourth generation systems is a battle between vendors of third-generation systems who are upgrading their products against a new group of systems often developed as open-source software with subscriptions for premium services. The existing companies seem to have the upper hand, but the open source DBMS products have gained important commercial usage.

1.3.2 Current Market for Database Software

The market positions for the enterprise DBMSs have changed a little in the last decade. According to a 2015 report by the Gartner Group, Oracle continues as the market leader with 41.6% of revenues. Microsoft SQL Server and IBM DB2 have switched positions after 2013 with Microsoft at 19.4% and IBM at 16.5%. SAP Sybase and Teradata complete the top 5 in 2015 revenues. Amazon Web Services has emerged as a strong competitor with a market share of 2.3%, just behind the Teradata share. The top five NoSQL vendors collectively have just above 1% of the market indicating the growing but still small impact of NoSQL products.

DB-Engines.com ranks DBMS products by popularity using the number of mentions on websites, frequency of search in Google Trends, job offers in leading job websites, and profiles in professional websites. The DB-Engines ranking (top 10) in June 2017 was Oracle, MySQL, Microsoft SQL Server, PostgreSQL, MongoDB (NoSQL product), DB2, Microsoft Access, Cassandra (NoSQL product), Redis (NoSQL), and SQLite.

Open source DBMS products have begun to challenge the commercial DBMS products at the low end of the enterprise DBMS market. Although source code for open source DBMS products is available without charge, most organizations purchase

support contracts so the open source products are not free. Still, many organizations have reported lower cost of ownership using open source DBMS products. MySQL, first introduced in 1995, is the leader in the open source DBMS market. Open source DBMS products feature prominently in the DB-Engines.com ranking with six open source products (MySQL, PostgreSQL, MongoDB, Cassandra, Redis, and SQLite).

In the market for desktop database software, Microsoft Access dominates at least in part because of the dominance of Microsoft Office. Desktop database software is primarily sold as part of office productivity software. With Microsoft Office holding about 90% of the office productivity market, Access holds a comparable share of the desktop database software market. Other significant products in the desktop database market are open source products LibreOffice Base and OpenOffice Base along with commercial product FileMaker Pro.

To provide coverage of both enterprise and desktop database software, this book provides significant coverage of Oracle and Microsoft Access. In addition, the emphasis on the SQL standard in Parts 2 and 5 provides database language coverage for the other major products.

Because of the potential growth of personal computing devices, most major DBMS vendors have now entered the embedded DBMS market. Embedded DBMS software is sold primarily by value-added software resellers as part of an application, such as an accounting package. Thus, embedded DBMSs are hidden from users and require little or no ongoing maintenance. Some of the leading embedded DBMS products are Oracle Berkeley DB, Firebird Embedded, MySQL Embedded, SQLite, UNICOM Global SolidDB, Microsoft SQL Server Compact, and Sybase Advantage Database Server.

The market for cloud-based DBMSs is rapidly evolving so market shares and size are difficult to determine. Most major DBMS vendors offer cloud based solutions with some vendors providing both traditional SQL and emerging NoSQL products. For example, Amazon offers Relational Data Service (SQL) and Amazon DynamoDB (NoSQL) while Microsoft offers Azure (SQL) and DocumentDB (NoSQL). The impact of cloud computing on the DBMS market has begun to mature in 2017.

1.4 ARCHITECTURES OF DATABASE MANAGEMENT SYSTEMS

To provide insight about the internal organization of DBMSs, this section describes two architectures or organizing frameworks. The first architecture describes an organization of database definitions to reduce the cost of software maintenance. The second architecture describes an organization of data and software to support remote access. These architectures promote a conceptual understanding rather than indicate actual DBMS implementation.

1.4.1 Data Independence and the Three Schema Architecture

In early DBMSs, there was a close connection between a database and computer programs that accessed the database. Essentially, the DBMS was considered part of a programming language. As a result, the database definition was part of the computer programs that accessed the database. In addition, the conceptual meaning of a database was not separate from its physical implementation on magnetic disk. The definitions about the structure of a database and its physical implementation were mixed inside computer programs.

The close association between a database and related programs led to problems in software maintenance. Software maintenance encompassing requirement changes, corrections, and enhancements can consume a large fraction of software development budgets. In early DBMSs, most changes to the database definition caused changes to computer programs. In many cases, changes to computer programs involved detailed inspection of the code, a labor-intensive process. This code inspection work is similar

to year 2000 compliance in which date formats were changed to four digits. Performance tuning of a database was difficult because sometimes hundreds of computer programs had to be recompiled for every change. Because database definition changes are common, a large fraction of software maintenance resources were devoted to database changes. Some studies have estimated the percentage as high as 50% of software maintenance resources.

Data Independence
a database should have an identity separate from the applications (computer programs, forms, and reports) that use it. The separate identity allows the database definition to be changed without affecting related applications.

The concept of **data independence** emerged to alleviate problems with program maintenance. Data independence means that a database should have an identity separate from the applications (computer programs, forms, and reports) that use it. The separate identity allows the database definition to be changed without affecting related applications. For example, if a new column is added to a table, applications not using the new column should not be affected. Likewise if a new table is added, only applications that need the new table should be affected. This separation should be even more pronounced if a change only affects physical implementation of a database. Database specialists should be free to experiment with performance tuning without concern about computer program changes.

Three Schema Architecture
an architecture for compartmentalizing database descriptions. The Three Schema Architecture was proposed as a way to achieve data independence.

In the mid-1970s, the concept of data independence led to the proposal of the Three Schema Architecture depicted in Figure 1.12. The word **schema** as applied to databases means database description. The Three Schema Architecture includes three levels of database description. The external level is the user level. Each group of users can have a separate external view (or view for short) of a database tailored to the group's specific needs.

In contrast, the conceptual and internal schemas represent the entire database. The conceptual schema defines the entity types and relationships. For a business database, the conceptual schema can be quite large, perhaps hundreds of entity types and relationships. Like the conceptual schema, the internal schema represents the entire database. However, the internal schema represents the storage view of the database whereas the conceptual schema represents the logical meaning of the database. The internal schema defines files, collections of data on a storage device such as a hard disk. A file can store one or more entity types described in the conceptual schema.

To make the three schema levels clearer, Table 1-3 shows differences among database definition at the three schema levels using examples from the features described in Section 1.2. Even in a simplified university database, the differences among the schema levels are clear. With a more complex database, the differences would be even

FIGURE 1.12
Three Schema Architecture

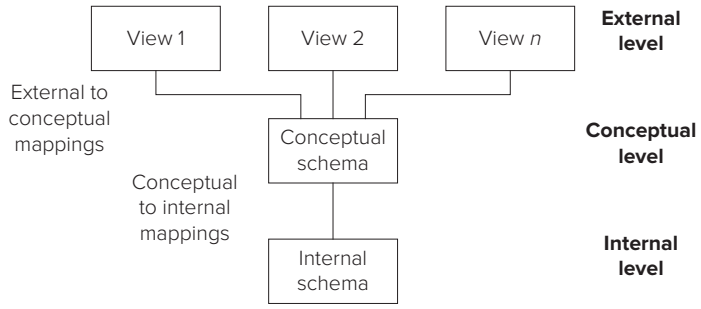


TABLE 1-3
University Database Example Depicting Differences among Schema Levels

Schema Level	Description
External	HighGPAView: data required for the query in Figure 1.7 FacultyAssignmentFormView: data required for the form in Figure 1.9 FacultyWorkLoadReportView: data required for the report in Figure 1.10
Conceptual	Student, Enrollment, Course, Faculty, and Enrollment tables and relationships (Figure 1.6)
Internal	Files needed to store the tables; extra files (indexed property in Figure 1.5) to improve performance

more pronounced with many more views, a much larger conceptual schema, and a more complex internal schema.

The schema mappings describe how a schema at a higher level is derived from a schema at a lower level. For example, the external views in Table 1-3 are derived from the tables in the conceptual schema. The mapping provides the knowledge to convert a request using an external view (for example, HighGPAView) into a request using the tables in the conceptual schema. The mapping between conceptual and internal levels shows how entities are stored in files.

DBMSs, using schemas and mappings, ensure data independence. Typically, applications access a database using a view. The DBMS converts an application's request into a request using the conceptual schema rather than the view. The DBMS then transforms the conceptual schema request into a request using the internal schema. Most changes to the conceptual or internal schema do not affect applications because applications do not use the lower schema levels. The DBMS, not the user, is responsible for using the mappings to make the transformations. For more details about mappings and transformations, Chapter 10 describes views and transformations between the external and conceptual levels. Chapter 8 describes query optimization, the process of converting a conceptual level query into an internal level representation.

The Three Schema Architecture is an official standard of the American National Standards Institute (ANSI). However, the specific details of the standard were never widely adopted. Rather, the standard serves as a guideline about data independence. The spirit of the Three Schema Architecture is widely implemented in third- and fourth-generation DBMSs.

1.4.2 Parallel and Distributed Database Processing

With the growing importance of computer networks and electronic commerce, distributed processing is becoming a crucial function of DBMSs. Distributed processing allows geographically dispersed computers to cooperate when providing data access. A large part of electronic commerce on the Web involves accessing and updating remote databases. Many databases in retail, banking, and security trading are now available through the Web. DBMSs use available network capacity and local processing capabilities to provide efficient remote database access.

Distributed processing can be applied to databases to distribute tasks among servers, divide a task among processing resources, and distribute data among network sites. To distribute tasks among servers, many DBMSs use the client-server architecture. A **client** is a program that submits requests to a server. A **server** processes requests on behalf of a client. For example, a client may request a server to retrieve product data. The server locates the data and sends them back to the client. The client may perform additional processing on the data before displaying the results to the user. DBMSs may employ one or more levels of servers to distribute different kinds of database processing. In Figure 1.13(a), the database server and database are located on a remote computer. In Figure 1.13(b), an additional middleware server is added to efficiently process messages from a large number of clients.

Client-Server Architecture an arrangement of components (clients and servers) among computers connected by a network. The client-server architecture supports efficient processing of messages (requests for service) between clients and servers.

In the last decade, parallel database technology has gained commercial acceptance for large organizations. Most enterprise DBMS vendors and some open source DBMSs support parallel database technology to meet market demand. Organizations are utilizing these products to realize the benefits of improved performance and availability. Parallel database processing can improve performance through speedup (performing a task faster) and scaleup (performing more work in the same time). Parallel database processing can increase availability because a DBMS can dynamically adjust to the level of available resources. Figure 1.14 depicts two common parallel database architectures that can provide improved performance and availability. In Figure 1.14(a) known as the shared disk (SD) architecture, each processor has its own memory but the processors share the disks. In Figure 1.14(b) known as shared nothing (SN) architecture, each processor has its own memory and disks.

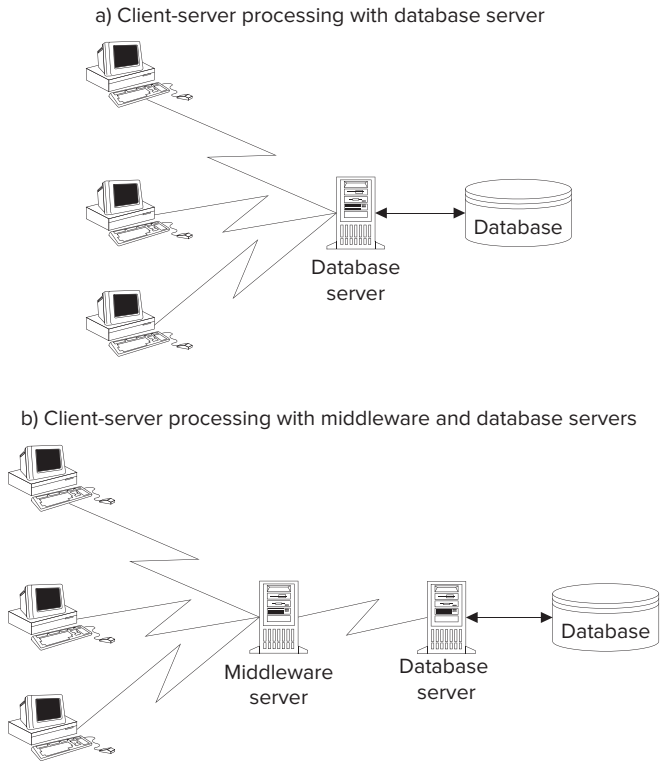


FIGURE 1.13
Typical Client-Server Architectures

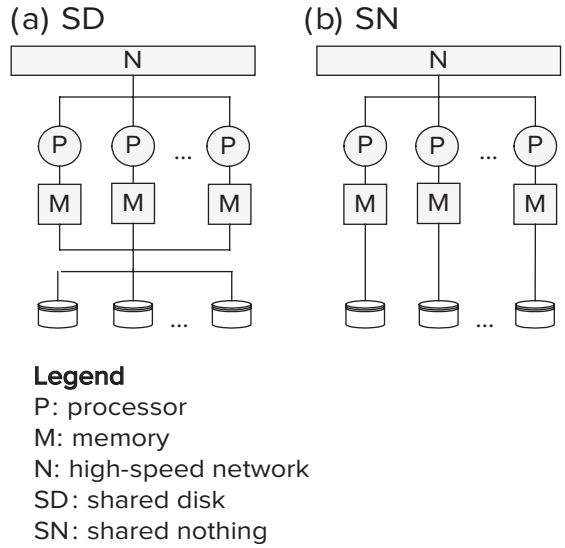


FIGURE 1.14
Basic Parallel Database Architectures

Parallel DBMS
a DBMS capable of utilizing tightly-coupled computing resources (processors, disks, and memory). Tight coupling is achieved by networks with data exchange time comparable to the time of the data exchange with a disk. Parallel database technology promises performance improvements and high availability.

Distributed data provides local control and reduced communication costs. Distributing a database allows the location of data to match an organization's structure. Decisions about sharing and maintaining data can be set locally to provide control closer to the data usage. Data should be located so that 80 percent of the requests are local. Local requests incur little or no communication costs and delays compared to remote requests. Figure 1.15 depicts a distributed database with three sites in Denver, London, and Tokyo. Each site can control access to its local data and cooperate to provide data sharing for tasks needing data from more than one site.

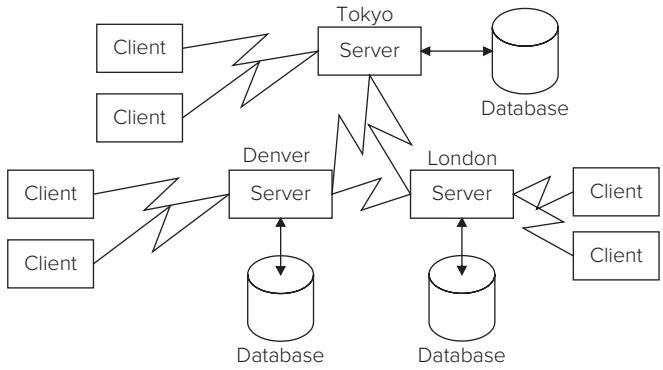


FIGURE 1.15
Distributed Database with Three Sites

Distributed Database
a database in which parts are located at different network sites. Distributed database technology supports local control of data, data sharing for requests involving data from more than one site, and reduced communication overhead.

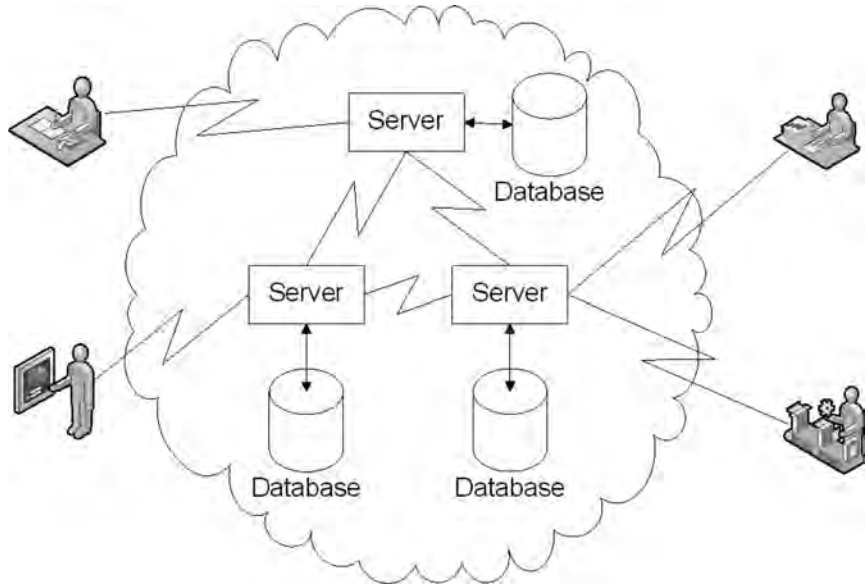


FIGURE 1.16
Cloud-Based Database Architecture

Client-server architectures, parallel database processing, and distributed databases provide flexible ways for DBMSs to interact with computer networks. The distribution of data and processing among clients and servers and the possible choices to locate data and software are much more complex than described here. You will learn more details about these architectures in Chapter 18.

The architectures presented in this section assume a traditional product licensing and hosting approach. Cloud computing provides a new approach without initial product licensing costs and no hosting requirements. Using web-based interfaces, organizations can design and deploy databases with dynamic resource allocation provided by the cloud as depicted in Figure 1.16. The cloud service may restrict the design flexibility for database design and operations available for database usage. Internally, the cloud can use any distributed processing approach although the internal details of the cloud are invisible to organizations using the cloud service.

1.5 ORGANIZATIONAL IMPACTS OF DATABASE TECHNOLOGY

This section completes your introduction to database technology by discussing the effects of database technology on organizations. The first subsection describes possible interactions that you may have with a database in an organization. The second

subsection describes approaches to plan and control data produced and used by an organization. Special attention is given to management roles that you can play as part of an effort to control data resources. Chapter 16 provides more detail about the tools and processes used in these management roles.

1.5.1 Interacting with Databases

Because databases are pervasive, there are a variety of ways in which you may interact with databases. The classification in Figure 1.17 distinguishes between functional users who interact with databases as part of their work and information systems professionals who participate in designing and implementing databases. Each box in the hierarchy represents a role that you may play. You may simultaneously play more than one role. For example, a functional user in a job such as a financial analyst may play all three roles in different databases. In some organizations, the distinction between functional users and information systems professionals is blurred. In these organizations, functional users may participate in designing and implementing databases.

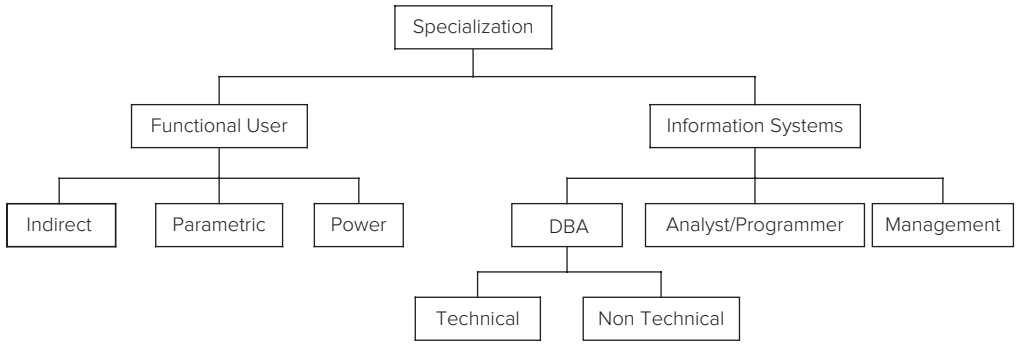
Functional users can play a passive or an active role when interacting with databases. Indirect usage of a database is a passive role. An indirect user is given a report or some data extracted from a database. A parametric user is more active than an indirect user. A parametric user requests existing forms or reports using parameters, input values that change from usage to usage. For example, a parameter may indicate a date range, sales territory, or department name. The power user is the most active. Because decision-making needs can be difficult to predict, ad hoc or unplanned usage of a database is important. A power user is skilled enough to build a form or report when needed. Power users should have a good understanding of nonprocedural access, a skill described in Parts 2 and 5 of this book.

Information systems professionals interact with databases as part of developing an information system. Analyst/programmers are responsible for collecting requirements, designing applications, and implementing information systems. They create and use external views to develop forms, reports, and other parts of an information system. Management has an oversight role in the development of databases and information systems. Information systems professionals in analyst/programmer roles should have a good knowledge of database development and application development in Parts 3 to 5 of this book.

Database Administrator
a support position that specializes in managing individual databases and DBMSs.

Database administrators assist both information systems professionals and functional users. Database administrators have a variety of both technical and nontechnical responsibilities (Table 1-4). Technical skills are more detail-oriented; nontechnical responsibilities are more people-oriented. The primary technical responsibility is database design. On the non-technical side, the database administrator's time is split among a number of activities. Database administrators can also have responsibilities in planning databases and evaluating DBMSs. Chapter 16 provides more details about responsibilities and tools of database administrators.

FIGURE 1.17
Classification of Roles



Technical	Non-technical
Designing conceptual schemas	Setting database standards
Designing internal schemas	Devising training materials
Monitoring database performance	Promoting benefits of databases
Selecting and evaluating database software	Consulting with users
Managing security for database usage	Planning new databases
Troubleshooting database problems	

TABLE 1-4
Responsibilities of the Database Administrator

1.5.2 Managing Data Resources in Organizations

Organizations have used two approaches to manage data resources. The more established approach, information resource management, focuses on information technology as a tool for processing, distributing, and integrating information throughout an organization. Management of information resources has many similarities with managing physical resources such as inventory. Inventory management involves activities such as safeguarding inventory from theft and deterioration, storing it for efficient usage, choosing suppliers, handling waste, coordinating movement, and reducing holding costs. Information resource management involves similar activities: planning databases, acquiring data, protecting data from unauthorized access, ensuring reliability, coordinating flow among information systems, and eliminating duplication.

Due to the rapid growth of electronic commerce and financial scandals in the 2000s, data governance has emerged as a complementary approach for managing data resources. According to the Data Governance Institute (www.dgi.com), "data governance is the exercise of decision-making and authority for data-related matters." Data governance provides a system of checks and balances to develop data rules and policies, support application of data rules and policies, and evaluate compliance of data rules and policies. Organizations use the artifacts of data governance to mitigate risks associated with the complex regulatory environment, information security, and information privacy especially for personal identifiable data and related business transactions.

As part of controlling data resources, new management responsibilities have been created in many organizations. The **data administrator** is a management role with responsibilities to plan the development of new databases and control usage of data throughout an organization. The data administrator maintains an enterprise data architecture that describes existing databases and new databases and also evaluates new information technologies and determines standards for managing databases. The data administrator supports data governance through participation in the data governance organization and consultation on activities managed by the data governance office.

Data Administrator
a management position that performs planning and policy setting for the data resources of an organization.

The data administrator role typically has broader responsibilities than the database administrator role. A data administrator primarily has planning and policy setting roles, while a database administrator has a more technical role focused on individual databases and DBMSs. A data administrator also views data resources in a broader context and considers all kinds of data, both traditional business data and non-traditional unstructured data such as images, videos, and social media. A major effort in many organizations is to develop a data governance program to manage risks associated with usage of corporate data assets. Data administrators typically assume in a leadership role in the data governance program while database administrators serve in support roles by implementing controls for data governance policies.

Because of broader responsibilities, the data administrator typically is higher in an organization chart. Figure 1.18 depicts two possible placements of data administrators and database administrators. In a small organization, both roles may be combined in systems administration.

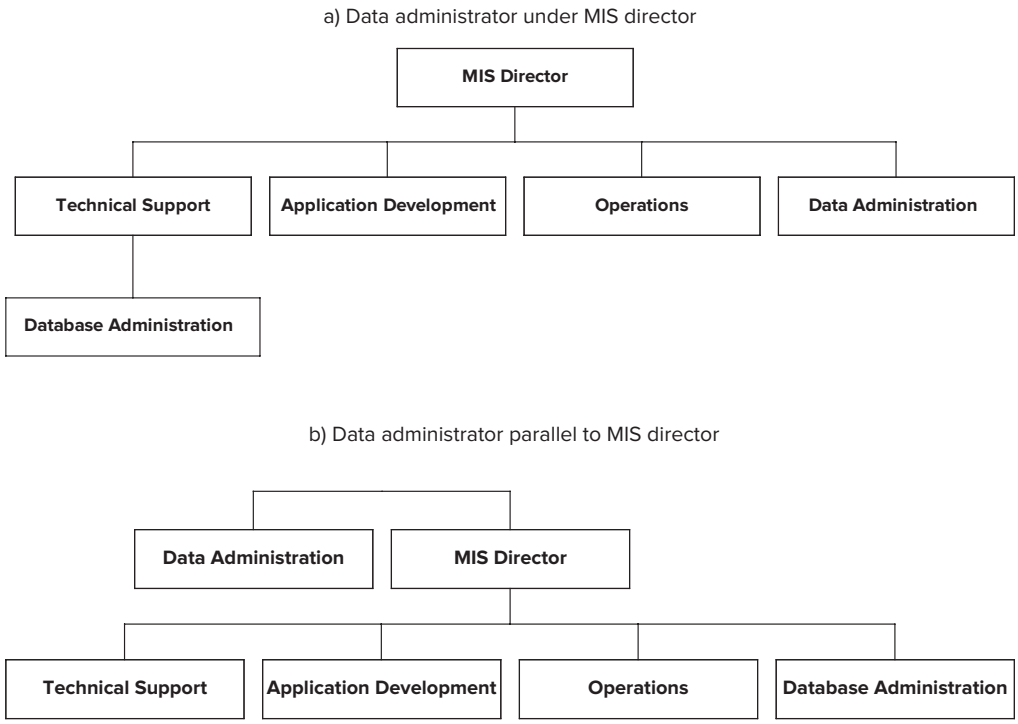


FIGURE 1.18
Organizational Placement of Data and Database Administration

CLOSING THOUGHTS

Chapter 1 has provided a broad introduction to DBMSs. You should use this background as a context for the skills and knowledge you will acquire in subsequent chapters. You learned that databases contain interrelated data that can be shared across multiple parts of an organization. DBMSs support transformation of data for decision making. To support this transformation, database technology has evolved from simple file access to powerful systems that support database definition, nonprocedural access, application development, transaction processing, and performance tuning. Nonprocedural access is the most vital element because it allows access without detailed coding. You learned about two architectures that provide organizing principles for DBMSs. The Three Schema Architecture supports data independence, an important concept for reducing the cost of software maintenance. Client-server architectures, parallel database processing, and distributed databases allow databases to be accessed over computer networks, a feature vital in today's networked world.

The skills emphasized in later chapters should enable you to work as an active functional user or analyst. Both kinds of users need to understand the skills taught in the second part of this book. The fifth part of the book provides skills for analysts/programmers. This book also provides the foundation of skills to obtain a specialist position as a database or data administrator. The skills in the third, fourth, sixth, and seventh parts of this book are most useful for a position as a database administrator. However, you will probably need to take additional courses, learn details of popular DBMSs, and acquire management experience before obtaining a specialist role. A position as a database specialist can be an exciting and lucrative career opportunity that you should consider.

REVIEW CONCEPTS

- Database characteristics: persistent, interrelated, and shared
- Features of database management systems (DBMSs)
- Nonprocedural access: a key to software productivity
- Transaction: a unit of work that should be processed reliably
- Application development using nonprocedural access to specify the data requirements of forms and reports
- Procedural language interface for combining nonprocedural access with a programming language such as Java or Visual Basic
- Evolution of database software over four generations of technological improvement
- Current emphasis on database software for multimedia support, distributed processing, more powerful operators, data warehouses, and big data
- Types of DBMSs: enterprise, desktop, embedded
- Impact of big data demands and NoSQL database technology to deal with big data challenges
- Data independence to alleviate problems with maintenance of computer programs
- Three Schema Architecture for reducing the impact of database definition changes
- Client-server processing, parallel database processing, and distributed database processing for using databases over computer networks
- Cloud-based database architecture for scalable, on-demand database services without ownership costs and risks
- Database specialist roles: database administrator and data administrator
- Information resource management for utilizing information technology
- Data governance for mitigating risks associated with the complex regulatory environment, information security, and information privacy

QUESTIONS

1. Describe a database that you have used on a job or as a consumer. List the entities and relationships that the database contains. If you are not sure, imagine the entities and relationships that are contained in the database.
2. For the database in question (1), list different user groups that can use the database.
3. For one of the groups in question (2), describe an application (form or report) that the group uses.
4. Explain the persistent property for databases.
5. Explain the interrelated property for databases.
6. Explain the shared property for databases.
7. What is a DBMS?
8. What is SQL?
9. Describe the difference between a procedural and a nonprocedural language. What statements belong in a procedural language but not in a nonprocedural language?
10. Why is nonprocedural access an important feature of DBMSs?

11. What is the connection between nonprocedural access and application (form or report) development? Can nonprocedural access be used in application development?
12. What is the difference between a form and a report?
13. What is a procedural language interface?
14. What is a transaction?
15. What features does a DBMS provide to support transaction processing?
16. For the database in question (1), describe a transaction that uses the database. How often do you think that the transaction is submitted to the database? How many users submit transactions at the same time? Make guesses for the last two parts if you are unsure.
17. What is an enterprise DBMS?
18. What is a desktop DBMS?
19. What is an embedded DBMS?
20. What were the prominent features of first-generation DBMSs?
21. What were the prominent features of second-generation DBMSs?
22. What were the prominent features of third-generation DBMSs?
23. What are the prominent features of fourth-generation DBMSs?
24. For the database you described in question (1), make a table to depict differences among schema levels. Use Table 1-4 as a guide.
25. What is the purpose of the mappings in the Three Schema Architecture? Is the user or DBMS responsible for using the mappings?
26. Explain the ways that the Three Schema Architecture supports data independence.
27. In a client-server architecture, why are processing capabilities divided between a client and server? In other words, why not have the server do all the processing?
28. What benefits can be provided by parallel database processing?
29. What benefits can be provided by distributing parts of a database among different network sites?
30. For the database in question (1), describe how functional users may interact with the database. Try to identify indirect, parametric, and power uses of the database.
31. Explain the differences in responsibilities between an active functional user of a database and an analyst. What schema level is used by both kinds of users?
32. Which role, database administrator or data administrator, is more appealing to you as a long-term career goal? Briefly explain your preference.
33. What advantages are reported by organization using open source DBMS products?
34. What is the state of the cloud computing segment of the DBMS marketplace?
35. What is the relationship between the cloud-based database architecture and other distributed processing architectures for database computing?
36. What is information resource management?
37. What is data governance?
38. What are the responsibilities of data administrators versus database administrators for data governance programs?
39. Identify several sources of data growth that challenge organizations and vendors of database products.
40. What features of NoSQL database products address the challenges of big data?

PROBLEMS

Because of the introductory nature of this chapter, there are no problems in this chapter. Problems appear at the end of most other chapters.

REFERENCES FOR FURTHER STUDY

The Databases section of InfoWorld (www.infoworld.com/category/database) provides details about database software, data management practices, and current industry trends. To learn more about the role of database specialists and information resource management, you should consult Mullins (2012).

chapter

2

Introduction to Database Development

Learning Objectives

This chapter provides an overview of the database development process. After this chapter, the student should have acquired the following knowledge and skills:

- Explain the steps in the information systems life cycle
- Describe the role of databases in an information system
- Explain the goals of database development
- Understand the relationships among phases in the database development process
- Describe features typically provided by CASE tools for database development

OVERVIEW

Chapter 1 provided a broad introduction to database usage in organizations and database technology. You learned about the characteristics of business databases, essential features of database managements systems (DBMSs), architectures for deploying databases, and organizational roles interacting with databases. This chapter continues your introduction to database management with a broad focus on database development. You will learn about the context, goals, phases, and tools of database development to facilitate the acquisition of specific knowledge and skills in Parts 3 and 4.

Before you can learn specific skills, you need to understand the broad context for database development. This chapter presents a context for databases as part of an information system. You will learn about components of information systems, the life cycle of information systems, and the role of database development as part of information systems development. This information systems context provides a background for database development. You will learn the phases of database development, the kind of skills used in database development, and software tools that can help you develop databases.

2.1 INFORMATION SYSTEMS

Databases exist as part of an information system. Before you can understand database development, you must understand the larger environment that surrounds a database. This section describes the components of an information system and several methodologies to develop information systems.

2.1.1 Components of Information Systems

A system is a set of related components that work together to accomplish some objectives. Objectives are accomplished by interacting with the environment and performing functions. For example, the human circulatory system, consisting of blood, blood vessels, and the heart, makes blood flow to various parts of the body. The circulatory system interacts with other systems of the body to ensure that the right quantity and composition of blood arrives in a timely manner to various body parts.

An information system is similar to a physical system (such as the circulatory system) except that an information system manipulates data rather than a physical object like blood. An information system accepts data from its environment, processes data, and produces information for decision making. For example, an information system for processing student loans (Figure 2.1) helps a service provider track loans for lending institutions. The environment of this system consists of lenders, students, and government agencies. Lenders send approved loan applications and students receive cash for school expenses. After graduation, students receive monthly statements and remit payments to retire their loans. If a student defaults, a government agency receives a delinquency notice.

Databases provide long-term memory for information systems, an essential role. The long-term memory contains entities and relationships. The database in Figure 2.1 contains data about students, loans, and payments so that the statements, cash disbursements, and delinquency notices can be generated. Information systems without permanent memory or with only a few variables in permanent memory are typically embedded in a device to provide a limited range of functions rather than an open range of functions as business information systems provide.

Databases are not the only components of information systems. Information systems also contain people, procedures, input data, output data, software, and hardware. Thus, developing an information system involves more than developing a database, as discussed in the next subsection.

2.1.2 Information Systems Development Process

Figure 2.2 shows the phases of the traditional systems development life cycle. The particular phases of the life cycle are not standard. Different authors and organizations

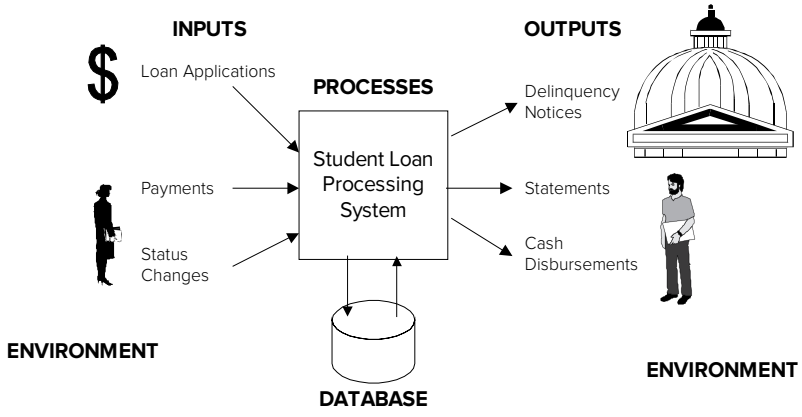


FIGURE 2.1
Overview of Student Loan Processing System

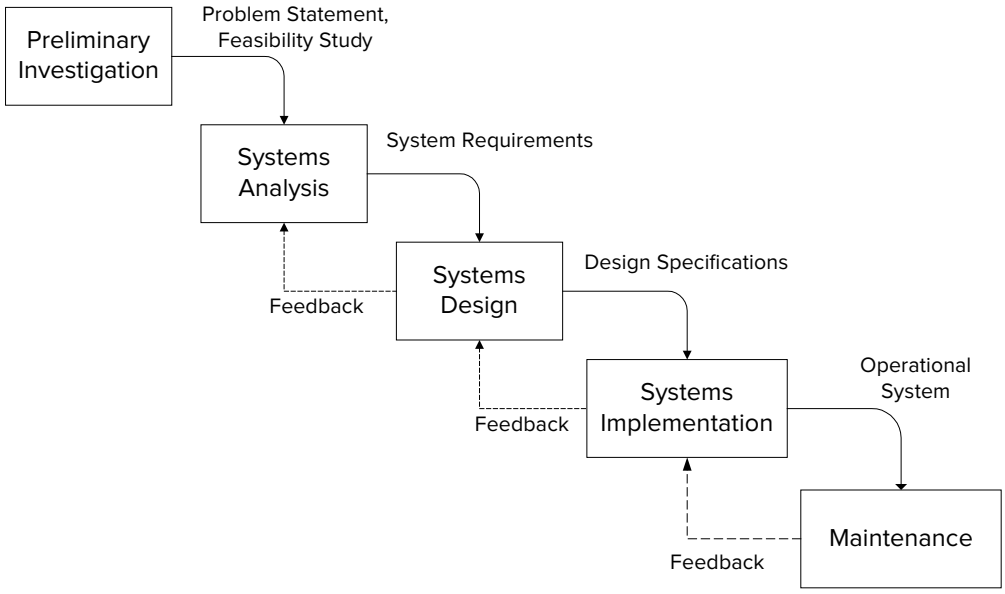


FIGURE 2.2
Traditional Systems Development Life Cycle

have proposed from 3 to 20 phases. The traditional life cycle, known as the waterfall model, contains sequential flow in which the result of each phase flows to the next phase. The traditional life cycle is mostly a reference framework. For most systems, the boundary between phases is blurred and there is considerable backtracking between phases. However, the traditional life cycle is still useful because it describes the kind of activities and shows addition of detail until an operational system emerges. The following items describe the activities in each phase:

- *Preliminary Investigation Phase*: Produces a problem statement and feasibility study. The problem statement contains the objectives, constraints, and scope of the system. The feasibility study identifies costs and benefits of the system. If the system is feasible, approval is given to begin systems analysis.
- *Systems Analysis Phase*: Produces requirements describing processes, data, and environment interactions. This phases uses diagramming techniques to document processes, data, and environment interactions. To produce requirements, work in this phase studies the current system and interviews users of the proposed system.
- *Systems Design Phase*: Produces a plan to efficiently implement the requirements. Work in this phase produces design specifications for processes, data, and environment interaction. The design specifications focus on choices to optimize resources given constraints.
- *Systems Implementation Phase*: Produces executable code, databases, and user documentation. To implement the system, work in this phase generates code to implement design specifications. Before making the new system operational, a transition plan from the old system to the new system is devised. To gain confidence and experience with the new system, an organization may run the old system in parallel to the new system for a period of time.
- *Maintenance Phase*: Produces corrections, changes, and enhancements to an operating information system. The maintenance phase commences when an information system becomes operational. The maintenance phase is fundamentally different from other phases because it comprises activities from all of the other phases. The maintenance phase ends after deploying a replacement system and retiring the current system. Due to the high fixed costs of developing new systems, the maintenance phase can last decades.

The traditional life cycle has been criticized for several reasons. First, an operating system is not produced until late in the process. By the time a system is operational, the requirements may have already changed. Second, there is often a rush to begin implementation so that a product is visible. In this rush, appropriate time may not be devoted to analysis and design.

A number of alternative methodologies have been proposed to alleviate these difficulties. Spiral development methodologies perform life cycle phases for subsets of a system, progressively producing a larger system until the complete system emerges. Rapid application development methodologies delay producing design documents until requirements are clear. Scaled-down versions of a system, known as prototypes, clarify requirements. Prototypes can be implemented rapidly using graphical development tools for generating menus, forms, reports, and other code. Implementing a prototype allows users to provide meaningful feedback to developers. Often, users may not understand the requirements unless they can experience a prototype. Thus, prototyping can reduce the risk of developing an information system because it allows earlier and more direct feedback about the system.

Agile development methodologies are another variation to traditional information systems development. Agile development methodologies promote active user involvement and team empowerment, viewing software development as an empirical process. Requirements evolve in agile development but the timescale of development is fixed. Agile development involves iteration through small incremental releases with testing integrated throughout the project lifecycle. Extreme programming, a prominent Agile development approach, features a set of primary technical practices and a set of corollary technical practices.

All development methodologies produce graphical models of the data, processes, and environment interactions. The data model describes the kinds of data and relationships. The process model describes relationships among processes. A process can provide input data used by other processes and use the output data of other processes. The environment interaction model describes relationships between events and processes. An event such as the passage of time or an action from the environment can trigger a process to start or stop. The systems analysis phase produces an initial version of these models. The systems design phase adds more details to efficiently implement the models.

Even though models of data, processes, and environment interactions are necessary to develop an information system, this book emphasizes data models only. In many information systems development efforts, the data model is the most important. For business information systems, development processes usually product the process and environment interaction models after the data model. Rather than present notation for the process and environment interaction models, this book emphasizes form and report development to depict connections among data, processes, and the environment.

2.2 GOALS OF DATABASE DEVELOPMENT

Broadly, the goal of database development is to create a database that provides an important resource for an organization. To fulfill this broad goal, the database should serve a large community of users, support organizational policies, contain high quality data, and provide efficient access. The remainder of this section describes the goals of database development in more detail.

2.2.1 Develop a Common Vocabulary

A database provides a common vocabulary for an organization. Before implementing a common database, different parts of an organization may have different terminology. For example, there may be multiple formats for addresses, multiple ways to identify customers, and different ways to calculate interest rates. After implementing a

database, communication can improve among different parts of an organization. Thus, a database can unify an organization by establishing a common vocabulary.

Achieving a common vocabulary is not easy. Developing a database requires compromise to satisfy a large community of users. In some sense, a good database designer shares some characteristics with a good politician. A good politician often finds solutions with which everyone finds something to agree and disagree. In establishing a common vocabulary, a good database designer also finds similar imperfect solutions. Forging compromises can be difficult, but the results can improve productivity, customer satisfaction, and other measures of organizational performance.

2.2.2 Define the Meaning of Data

A database contains business rules to support organizational policies. Defining business rules is the essence of defining the semantics or meaning of a database. For example, in an order entry system, an important rule is that an order must precede a shipment. The database can contain an integrity constraint to support this rule. Defining business rules enables a database to actively support organizational policies. This active role contrasts with the more passive role that databases have in establishing a common vocabulary.

In establishing the meaning of data, a database designer must choose appropriate constraint levels. Selecting appropriate constraint levels may require compromise to balance the needs of different groups. Overly strict constraints may force work-around solutions to handle exceptions. In contrast, loose constraints may allow incorrect data in a database. For example, in a university database, a designer must decide if a course offering can be stored without knowing the instructor. Some user groups may want initial entry of the instructor to ensure that course commitments can be met. Other user groups may want more flexibility because course catalogs are typically released well in advance of the beginning of the academic period. Forcing entry of the instructor at the time a course offering is stored may be too strict. If a database contains this constraint, users may use workarounds by using a default value such as TBA (to be announced). The appropriate constraint (forcing entry of the instructor or allowing later entry) depends on the importance of the needs of the user groups to the goals of the organization.

2.2.3 Ensure Data Quality

The importance of data quality is analogous to the importance of product quality in manufacturing. Poor product quality can lead to loss of sales, lawsuits, and customer dissatisfaction. Because data are the product of an information system, data quality is equally important. Poor data quality can lead to poor decision making about communicating with customers, identifying repeat customers, tracking sales, and resolving customer problems. For example, communicating with customers can be difficult if addresses are outdated or customer names are inconsistently spelled on different orders.

Data quality has many dimensions or characteristics, as depicted in Table 2-1. The importance of data quality characteristics can depend on the part of the database in which they are applied. For example, in the product part of a retail grocery database, important characteristics of data quality may be the timeliness and consistency of prices. For other parts of the database, other characteristics may be more important.

A database design should help achieve adequate data quality. When evaluating alternatives, a database designer should consider data quality characteristics. For example, in a customer database, a database designer should consider the possibility that some customers may not have U.S. addresses. Therefore, the database design may be incomplete if it fails to support non-U.S. addresses.

Achieving adequate data quality may require a cost-benefit trade-off. For example, in a grocery store database, the benefits of timely price updates are reduced consumer complaints and less loss in fines from government agencies. Achieving data quality can be costly both in preventative and monitoring activities. For example, to improve

TABLE 2-1
Common Characteristics of Data Quality

Characteristic	Meaning
Completeness	Database represents all important parts of the information system.
Lack of ambiguity	Each part of the database has only one meaning.
Correctness	Database contains values perceived by the user.
Timeliness	Business changes are posted to the database without excessive delays.
Reliability	Failures or interference do not corrupt database.
Consistency	Different parts of the database do not conflict.

the timeliness and accuracy of price updates, automated data entry may be used (preventative activity) as well as sampling the accuracy of the prices charged to consumers (monitoring activity).

The cost-benefit trade-off for data quality should consider long-term as well as short-term costs and benefits. Often the benefits of data quality are long-term, especially data quality issues that cross individual databases. For example, consistency of customer identification across databases can be a crucial issue for strategic decision making. The issue may not be important for individual databases. Chapter 14 on data integration addresses issues of data quality related to strategic decision making.

Organizations increasingly recognize that poor data quality can bring extra risks to an organization especially related to litigation and government regulations. Many businesses and government agencies have data governance organizations that deal with data quality, privacy, and security issues in a broad context. For data quality improvements, data governance initiatives typically focus on development of data quality measures, reporting status of data quality, and establishing decision rights and accountabilities. Chapter 16 provides details about data governance processes and tools covering data quality issues.

2.2.4 Find an Efficient Implementation

Even if the other design goals are met, a slow-performing database will not be used. Thus, finding an efficient implementation is paramount. However, an efficient implementation should respect the other goals as much as possible. An efficient implementation that compromises the meaning of the database or database quality may be rejected by database users.

Finding an efficient implementation is an optimization problem with an objective and constraints. Informally, the objective is to maximize performance subject to constraints about resource usage, data quality, and data meaning. Finding an efficient implementation can be difficult because of the number of choices available, the interaction among choices, and the difficulty of describing inputs. In addition, finding an efficient implementation is a continuing effort. Performance should be monitored and design changes should be made if warranted.

2.3 DATABASE DEVELOPMENT PROCESS

This section describes the phases of the database development process and discusses relationships to the information systems development process. The chapters in Parts 3 and 4 elaborate on the framework provided here.

2.3.1 Phases of Database Development

The goal of the database development process is to produce an operational database for an information system. To produce an operational database, you need to define

the three schemas (external, conceptual, and internal) and populate (supply with data) the database. To create these schemas, you can follow the process depicted in Figure 2.3. The first two phases are concerned with the information content of the database while the last two phases are concerned with efficient implementation. These phases are described in more detail in the remainder of this section.

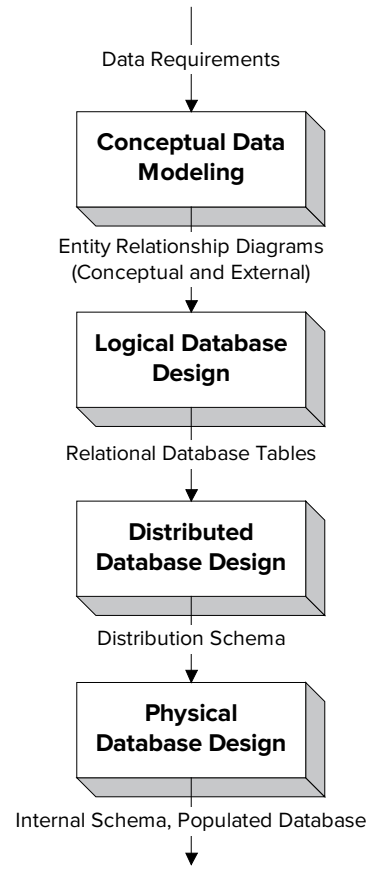


FIGURE 2.3
Phases of Database Development

Conceptual Data Modeling The conceptual data modeling phase uses data requirements and produces entity relationship diagrams (ERDs) for the conceptual schema and each external schema. Data requirements can have many formats such as interviews with users, documentation of existing systems, and proposed forms and reports. The conceptual schema should represent all the requirements and formats. In contrast, the external schemas (or views) represent the requirements of a particular usage of the database such as a form or report rather than all requirements. Thus, external schemas are generally much smaller than the conceptual schema.

The conceptual and external schemas follow the rules of the Entity Relationship Model, a graphical representation that depicts things of interest (entities) and relationships among entities. Figure 2.4 depicts an entity relationship diagram (ERD) for part of a student loan system. The rectangles (*Student* and *Loan*) represent entity types and labeled lines (*Receives*) represent relationships. Attributes or properties of entities are listed inside the rectangle. The underlined attribute, known as the primary key, provides unique identification for the entity type. Chapter 3 provides a precise definition of primary keys. Chapters 5 and 6 present more details about the Entity Relationship Model. Because the Entity Relationship Model is not fully supported by any DBMS, the conceptual schema is not biased toward any specific DBMS.

Logical Database Design The logical database design phase transforms the conceptual data model into a format understandable by a commercial DBMS. The logical design phase is not concerned with efficient implementation. Rather, the logical design phase is concerned with refinements to the conceptual data model. The refinements preserve the information content of the conceptual data model while enabling implementation on a commercial DBMS. Because most business databases are implemented on relational DBMSs, the logical design phase usually produces a table design.

The logical database design phase consists of two refinement activities: conversion and normalization. The conversion activity transforms ERDs into table designs using conversion rules. As you will learn in Chapter 3, a table design includes tables, columns, primary keys, foreign keys (links to other related tables), and other constraints. For example, the ERD in Figure 2.4 is converted into two tables as depicted in Figure 2.5. The normalization activity removes redundancies in a table design using constraints or dependencies among columns. Chapter 6 presents conversion rules while Chapter 7 presents normalization techniques.

Distributed Database Design The distributed database design phase marks a departure from the first two phases. The distributed database design and physical database design phases are both concerned with an efficient implementation. In contrast, the first two phases (conceptual data modeling and logical database design) are concerned with the information content of the database.

FIGURE 2.4
Partial ERD for the Student Loan System

FIGURE 2.5
Conversion of Figure 2.4

```
CREATE TABLE Student
(  StdNo    INTEGER     NOT NULL,
   StdName  CHAR(50),
   …
PRIMARY KEY (StdNo)     )
CREATE TABLE Loan
( LoanNo    INTEGER     NOT NULL,
  LoanAmt   DECIMAL(10,2),
  StdNo     INTEGER     NOT NULL,
  …
PRIMARY KEY (LoanNo),
FOREIGN KEY (StdNo) REFERENCES Student  )
```

Distributed database design involves choices about the location of data and processes to improve performance and provide local control of data. Performance can be measured in many ways such as reduced response time, improved availability of data, and improved control. For data location decisions, the database can be split in many ways to distribute it among computer sites. For example, a loan table can be distributed according to the location of the bank granting the loan. Another technique to improve performance is to replicate or make copies of parts of the database. Replication improves availability of the database but makes updating more difficult because multiple copies must be kept consistent.

Data location decisions should respect data ownership. An organization that controls some part of a database should control access to its data. For example, a franchise store should have control over access to its locally generated data. Distributed database technology presented in Chapter 18 enables an organization to align data location with data control.

For process location decisions, some of the work is typically performed on a server and some of the work is performed by a client. For example, the server often retrieves data and sends them to the client. The client displays the results in an appealing manner. There are many other options about the location of data and processing that are explored in Chapter 18.

Physical Database Design The physical database design phase, like the distributed database design phase, is concerned with an efficient implementation. Unlike distributed database design, physical database design involves performance at one computer location only. If a database is distributed, physical design decisions must be made for each location. An efficient implementation minimizes response time without using excessive resources such as disk space and main memory. Because response time is difficult to directly measure, other measures such as the amount of disk input-output activity is often used as a substitute.

In the physical database design phase, two important choices involve indexes and data placement. An index is an auxiliary file that can improve performance. For each column of a table, the designer decides whether an index can improve performance. An index can improve performance on retrievals but reduce performance on updates. For example, indexes on the primary keys (*StdNo* and *LoanNo* in Figure 2.5) can usually improve performance. For data placement, a designer makes decisions about clustering to locate data close together on a disk. For example, performance might improve by placing student rows near the rows of associated loans. Chapter 8 describes details of physical database design including index selection and data placement.

Splitting Conceptual Design for Large Projects The database development process shown in Figure 2.3 works well for moderate-size databases. For large databases, the conceptual modeling phase is usually modified. Designing large databases is a time-consuming and labor-intensive process often involving a team of designers. The

development effort can involve requirements from many different groups of users. To manage complexity, a divide and conquer strategy is used in many areas of computing. Dividing a large problem into smaller problems allows the smaller problems to be solved independently. The solutions to the smaller problems are then combined into a solution for the entire problem.

View design and integration (Figure 2.6) is an approach to managing the complexity of large database development efforts. In view design, an ERD is constructed for each group of users. A view is typically small enough for a single person to design. Multiple designers can work on views covering different parts of the database. The view integration process merges the views into a complete and consistent conceptual schema. Integration involves recognizing and resolving conflicts. To resolve conflicts, it is sometimes necessary to revise the conflicting views. Compromise is an important part of conflict resolution in the view integration process.

Cross-Checking with Application Development The database development process does not exist in isolation. Database development occurs sometimes concurrently with activities in the systems analysis, systems design, and systems implementation phases. The conceptual data modeling phase is part of the systems analysis phase. The logical database design phase is performed during systems design. The distributed database design and physical database design phases are usually divided between systems design and systems implementation. Most of the preliminary decisions for the last two phases can be made in systems design. However, many physical design and distributed design decisions must be tested on a populated database. Thus, some activities in the last two phases occur in systems implementation.

To fulfill the goals of database development, the database development process must be tightly integrated with other parts of information systems development. To produce data, process, and interaction models that are consistent and complete, cross-checking can be performed, as depicted in Figure 2.7. The information systems development process can be split between database development and applications development. The database development process produces ERDs, table designs, and so on as described in this section. The applications development process produces process models, interaction models, and prototypes. Prototypes are especially important for cross-checking. A database has no value unless it supports intended applications such as forms and reports. Prototypes can help reveal mismatches between the database and applications using the database.

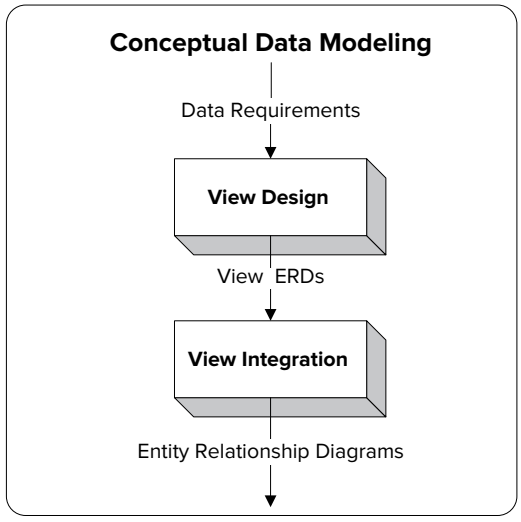


FIGURE 2.6
Splitting of Conceptual Data Modeling into View Design and View Integration

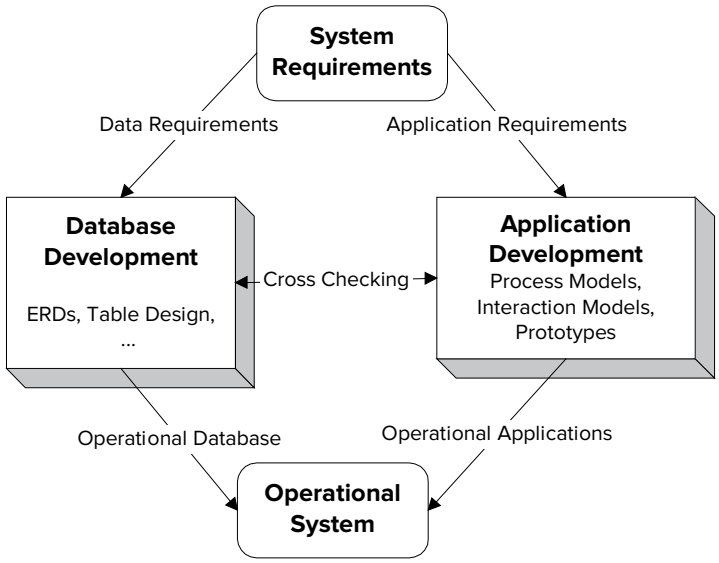


FIGURE 2.7
Interaction between Database and Application Development

2.3.2 Skills in Database Development

As a database designer, you need two different kinds of skills as depicted in Figure 2.8. The conceptual data modeling and logical database design phases involve mostly soft skills. Soft skills are qualitative, subjective, and people-oriented. Qualitative skills emphasize the generation of feasible alternatives rather than the best alternatives. As a database designer, you want to generate a range of feasible alternatives. The choice among feasible alternatives can be subjective. You should note the assumptions in

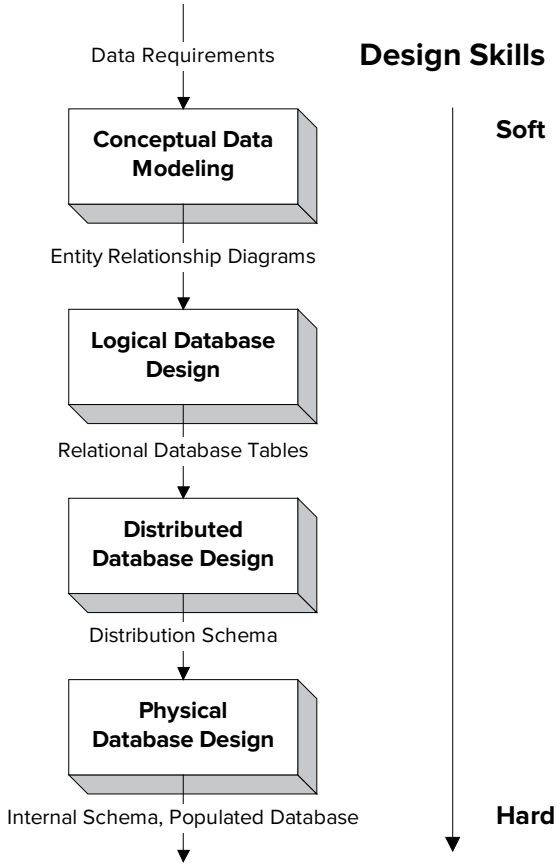


FIGURE 2.8
Design Skills Used in Database Development

which each feasible alternative is preferred. The alternative chosen is often subjective based on the designer's assessment of the most reasonable assumptions. Conceptual data modeling is especially people-oriented. In performing data modeling, you need to obtain requirements from diverse groups of users. Compromise and effective listening are essential skills in data modeling.

Distributed database design and physical database design involve mostly hard skills. Hard skills are quantitative, objective, and data intensive. A background in quantitative disciplines such as statistics and operations management can be useful to understand mathematical models used in these phases. Many of the decisions in these phases can be modeled mathematically using an objective function and constraints. For example, the objective function for index selection is to minimize disk reads and writes with constraints about the amount of disk space and response time limitations. Many decisions cannot be based on objective criteria alone because of uncertainty about database usage. To resolve uncertainty, intensive data analysis can be useful. The database designer should collect and analyze data to understand patterns of database usage and database performance.

Because of the diverse skills and background knowledge required in different phases of database development, role specialization can occur. Large organizations typically provide specialization in database design roles between data modelers and database performance experts. Data modelers perform conceptual data modeling and logical database design phases. Database performance experts mostly perform tasks in the distributed and physical database design phases. Because the skills are different in these roles, the same person will not perform both roles in large organizations. Small organizations typically lack role diversification with the same person fulfilling multiple roles.

2.4 TOOLS FOR DATABASE DEVELOPMENT

To improve productivity in developing information systems, computer-aided software engineering (CASE) tools have been created. CASE tools can help improve the productivity of information systems professionals working on large projects as well as end users working on small projects. A number of studies have provided evidence that CASE tools facilitate improvements in the early phases of systems development leading to lower cost, higher quality, and faster implementations.

Most CASE tools support the database development process. Some CASE tools support database development as a part of information systems development. Other CASE tools target various phases of database development without supporting other aspects of information systems development.

CASE tools can be classified as front-end or back-end tools. Front-end CASE tools help designers diagram, analyze, and document models used in the database development process. Back-end CASE tools create prototypes and generate code that can be used to cross-check a database with other components of an information system. This section presents features of CASE tools for database development and demonstrates a commercial CASE tool, Aqua Data Studio, with a focus on database development.

2.4.1 Diagramming

Diagramming is the most important and widely used function in CASE tools. Most CASE tools provide predefined shapes and connections among the shapes. The connection tools typically allow shapes to be moved while remaining connected as though glued. This glue feature provides important flexibility because symbols on a diagram typically are rearranged many times.

For large drawings, CASE tools provide several features. Most CASE tools allow diagrams to span multiple pages. Multiple-page drawings can be printed so that the

pages can be pasted together to make a wall display. Layout can be difficult for large drawings. Some CASE tools try to improve the visual appeal of a diagram by performing automatic layout. The automatic layout feature may minimize the number of crossing connections in a diagram. Although automated layout is not typically sufficient by itself, a designer can use it as a first step to improve the visual appearance of a large diagram.

2.4.2 Documentation

Documentation is one of the oldest and most valuable functions of CASE tools. CASE tools store various properties of a data model and link the properties to symbols on the diagram. Example properties stored in a CASE tool include alias names, integrity rules, data types, and owners. In addition to properties, CASE tools store text describing assumptions, alternatives, and notes. Both the properties and text are stored in the data dictionary, the database of the CASE tool. The data dictionary is also known as the repository or encyclopedia.

To support system evolution, many CASE tools can document versions. A version is a group of changes and enhancements to a system that is released together. Because of the volume of changes, groups of changes rather than individual changes are typically released together. In the life of an information system, many versions can be made. To aid in understanding relationships among versions, many CASE tools support documentation for individual changes and entire versions.

2.4.3 Analysis

CASE tools can provide active assistance to database designers through analysis functions. In documentation and diagramming, CASE tools help designers become more proficient. In analysis functions, CASE tools can perform the work of a database designer. An analysis function is any form of reasoning applied to specifications produced in the database development process. For example, an important analysis function is to convert between an ERD and a table design. Converting from an ERD to a table design is known as forward engineering and converting in the reverse direction is known as reverse engineering.

Analysis functions can be provided in each phase of database development. In the conceptual data modeling phase, analysis functions can reveal conflicts in an ERD. In the logical database design phase, conversion and normalization are common analysis functions. Conversion produces a table design from an ERD. Normalization removes redundancy in a table design. In the distributed database design and physical database design phases, analysis functions can suggest decisions about data location and index selection. In addition, analysis functions for version control can cross database development phases. Analysis functions can convert between versions and show a list of differences between versions.

Because analysis functions are advanced features in CASE tools, availability of analysis functions varies widely. Some CASE tools support little or no analysis functions while others support extensive analysis functions. Because analysis functions can be useful in each phase of database development, no single CASE tool provides a complete range of analysis functions. CASE tools tend to specialize by the phases supported. CASE tools independent of a DBMS typically specialize in analysis functions in the conceptual data modeling phase. In contrast, CASE tools offered by a DBMS vendor often specialize in physical database design phases.

2.4.4 Prototyping Tools

Prototyping tools provide a link between database development and application development. Prototyping tools can be used to create forms and reports that use a database. Because prototyping tools may generate code (SQL statements and programming language code), they are sometimes known as code generation tools. Prototyping tools

are often provided as part of a DBMS. The prototyping tools may provide wizards to aid a developer in quickly creating applications that can be tested by users. Prototyping tools can also create an initial database design by retrieving existing designs from a library of designs. This kind of prototyping tool can be very useful to end users and novice database designers.

2.4.5 Commercial CASE Tools

Table 2-2 summarizes major CASE tools that provide extensive features for database development. Each product in Table 2-2 supports multiple steps in database development although the quality, depth, and breadth of features varies across products. In addition, most of the products in Table 2-2 provide several versions that vary in price and features. All of the products are relatively neutral to a particular DBMS even though two products are offered by organizations with major DBMS products. Besides the full featured products listed in Table 2-2, other companies offer more drawing tools for database diagrams.

ER Modeler in Aqua Data Studio To depict features of commercial CASE tools, this section concludes with an overview of the ER Modeler component of Aqua Data Studio. The ER Modeler provides excellent drawing capabilities, forward

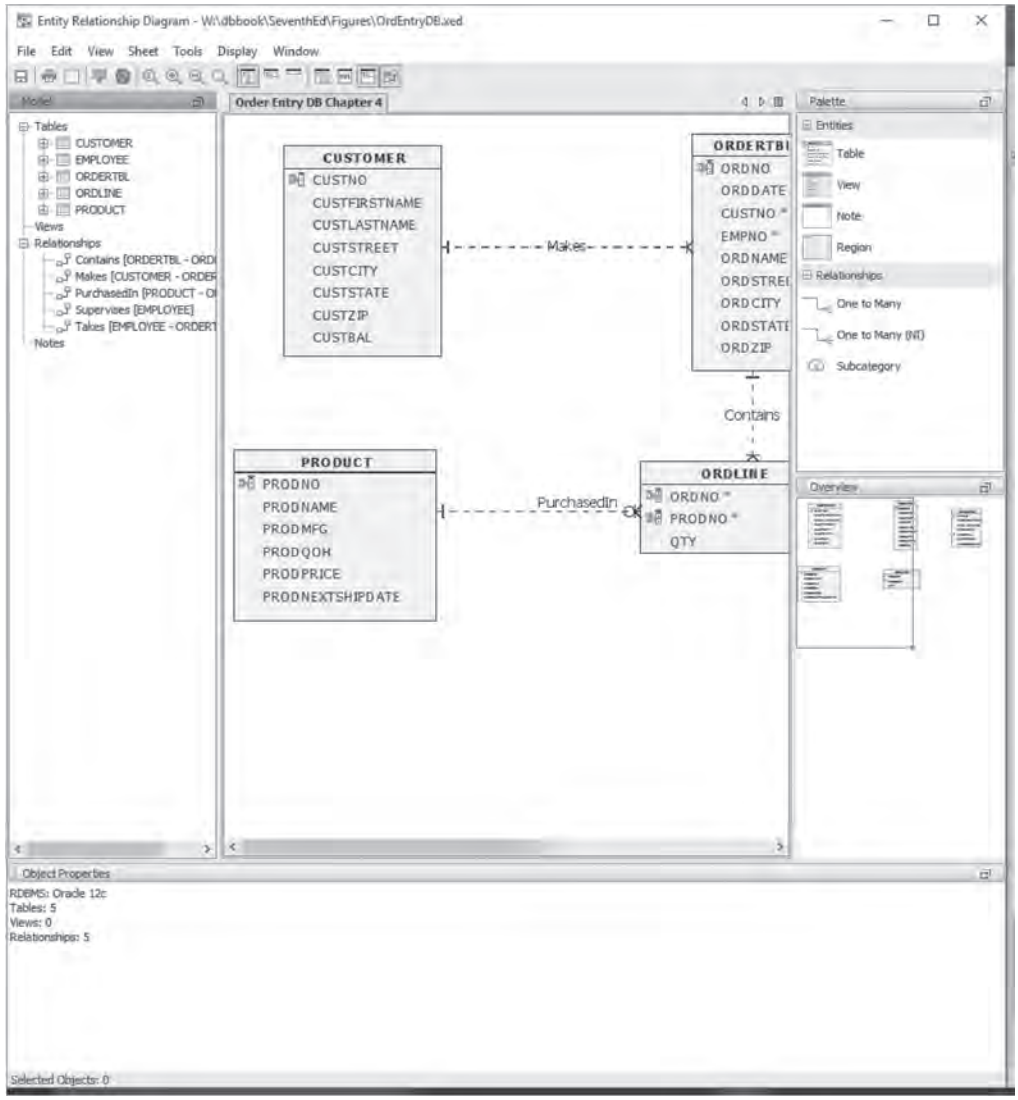
TABLE 2-2
Prominent CASE Tools for Database Development

Tool	Vendor	Innovative Features
SAP PowerDesigner	SAP	Forward and reverse engineering for relational databases and many programming languages; model management support for comparing and merging models; application code generation; UML support; business process modeling; XML code generation; version control; data integration support; physical design support; support for industry standard enterprise architecture frameworks
Oracle SQL Developer Data Modeler	Oracle	Forward and reverse engineering for relational databases; data warehouse modeling; code generation for other DBMSs; compare and merge models; version control; name standardization; design rules; impact analysis; wizards for view creation, view discovery, and foreign key discovery
erwin Data Modeler	ERWin	Forward and reverse engineering for relational databases; model reuse tools; bi-directional compare; model change impact analysis; schema and design analysis; version control; sub modeling support; workgroup support
ER/Studio Data Architect	IDERA	Forward and reverse engineering for relational databases; automated diagram layout; visual data lineage; model management support for comparing and merging models; UML support; version control; schema patterns for model reuse; workgroup support; data integration support
Visible Analyst	Visible Systems	Forward and reverse engineering for relational databases and XML; model management support for comparing and merging models; version control; database view design; data warehouse design diagrams; business requirements traceability; process integration with data; Enterprise Edition supports Zachman Framework for enterprise architecture design
Aqua Data Studio	AquaFold	Forward and reverse engineering, schema comparison, version control, DBA tools, query builder, schema object management
Database Engineering	Visual Paradigm	Forward and reverse engineering, editors for tables and views, generation of database patch scripts, trigger and stored procedure support, support for project management, enterprise architecture, system modeling, business modeling, user interface requirements, and software requirements in Visual Paradigm tool

and reverse engineering tools, and schema comparison tools. In addition to the ER Modeler component, Aqua Data Studio provides DBA tools for managing databases in a variety of DBMSs, a query builder, and code generation. Thus, Aqua Data Studio supports traditional CASE tool features as well as features to manage operations of databases.

The ER Modeler window contains panes for a drawing area, model objects, a palette of diagram shapes, an overview pane for managing large drawings, and an object summary as shown in Figure 2.9. The drawing pane contains a number of drawing sheets, each containing a database diagram. In Figure 2.9, the drawing pane contains one sheet showing a database diagram for an order entry database. The Palette pane shows entities (table, view, note, and region) and relationships (One to Many, One to Many (NI), and Subcategory) that can be placed in a drawing sheet. The Overview pane compresses the entire diagram with a red rectangle surrounding the visible part of the diagram. The Model pane displays the objects in a diagram (tables and relationships) with expansion to display details. In Figure 2.9, the Model pane expands tables and relationships to show the objects in the diagram. The Object Properties pane lists properties of the object selected in the drawing sheet. In Figure 2.9, the Object Properties pane lists properties of the entire diagram because no object in the diagram is selected.

FIGURE 2.9
ER Modeler Window



The ER Modeler provides multiple levels of detail in the drawing pane. Figure 2.9 shows the attribute level with table and column names. Relationship names can be added to the attribute level display as shown in Figure 2.9. The ER Modeler supports less detail with the primary key level (table and primary key names) and the entity level (just table names) and more detail with the physical schema level (data types added to the attribute level), nullable columns (attribute level and null constraints), and the comment level (attribute level and comments).

The ER Modeler provides a data dictionary with details of each object in a diagram. To edit properties in the data dictionary, you use the properties window for a specified object. Figure 2.10 displays the properties window for the *Product* table with tabs separating different collections of properties. The General tab shows the column names, data types, lengths and nulls allowed values for each column. Figure 2.11 displays properties for the *PurchasedIn* relationship with tabs for several collection of properties. The General tab contains the most prominent properties including cardinality, type, and nulls.

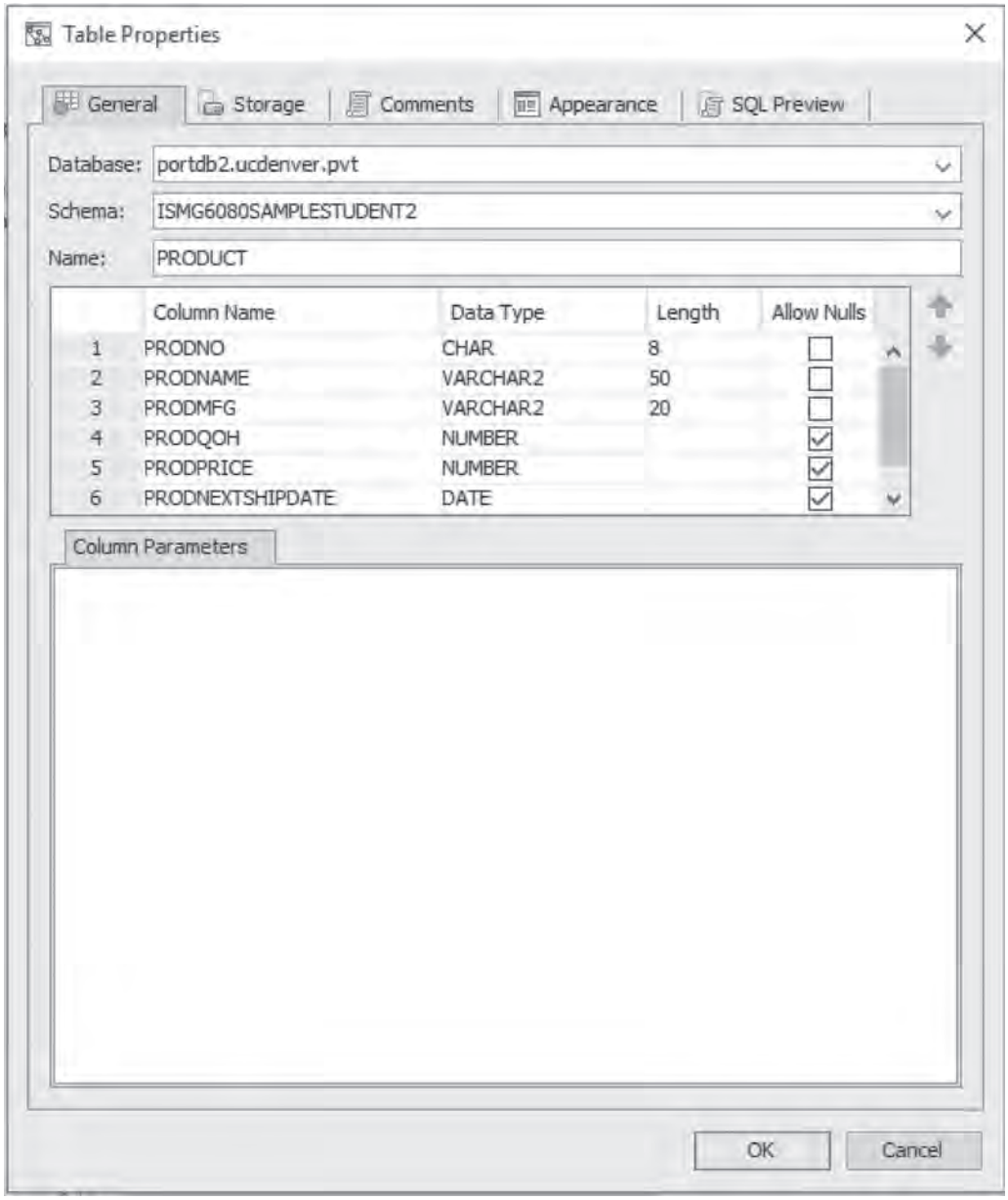
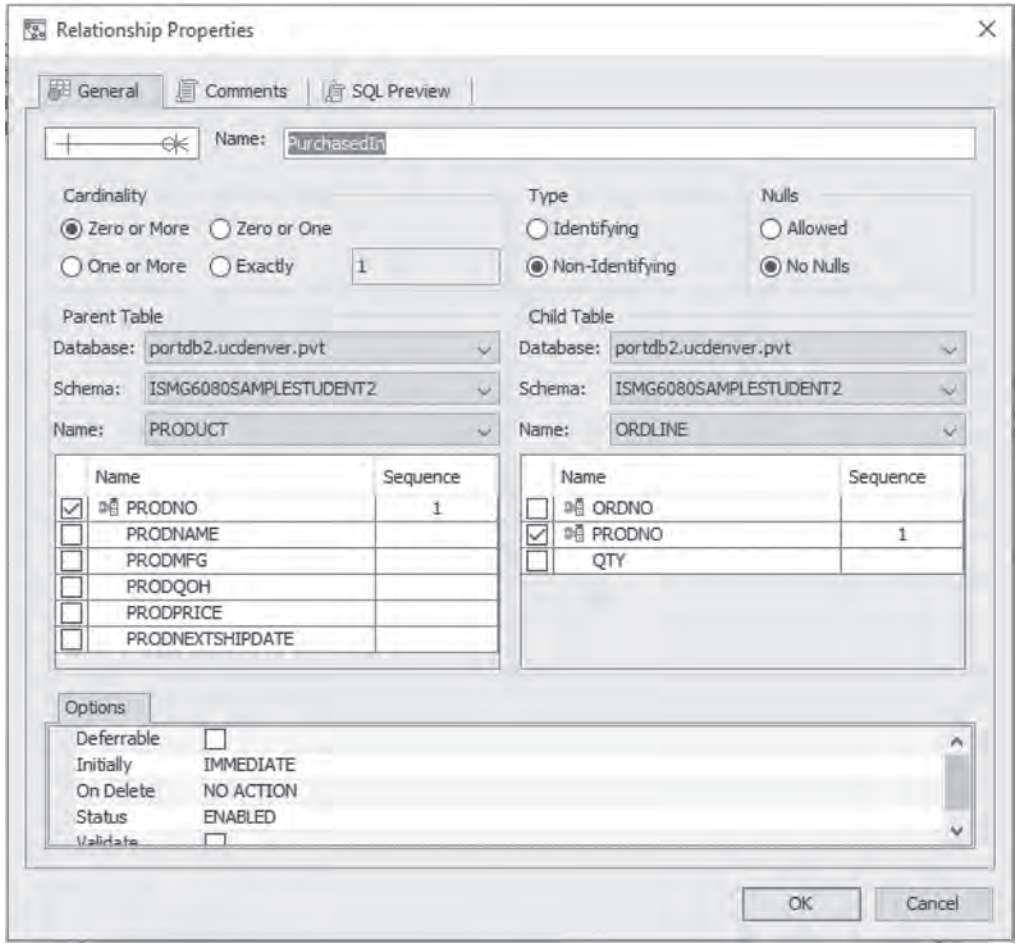


FIGURE 2.10
Table Properties Window for the *Product* Table

FIGURE 2.11
Relationships Properties Window for the *PurchasedIn* Relationship



CLOSING THOUGHTS

This chapter initially described the role of databases in information systems and the nature of the database development process. Information systems are collections of related components that produce data for decision making. A database provides the permanent memory for an information system. Development of an information system involves a repetitive process of analysis, design, and implementation. Database development occurs in all phases of systems development. Because a database is often a crucial part of an information system, database development can be the dominant part of information systems development. Development of the processing and environment interaction components are often performed after the database development. Cross-checking between a database and applications connects the database development process to the information systems development process.

After presenting the role of databases and the nature of database development, this chapter described the goals, phases, and tools of database development. The goals emphasize both the information content of the database as well as efficient implementation. The phases of database development first establish the information content of the database and then find an efficient implementation. The conceptual data modeling and logical database design phases involve the information content of the database. The distributed database design and physical database design phases involve

efficient implementation. Because developing databases can be a challenging process, computer-aided software engineering (CASE) tools have been created to improve productivity. CASE tools can be essential in helping the database designer to draw, document, and prototype the database. In addition, some CASE tools provide active assistance with analyzing a database design.

This chapter provides a context for the chapters in Parts 3 and 4. You might want to reread this chapter after completing the chapters in Parts 3 and 4. The chapters in Parts 3 and 4 provide details about the phases of database development. Chapters 5 and 6 present details of the Entity Relationship Model, data modeling practice using the Entity Relationship Model, and conversion from the Entity Relationship Model to the Relational Model. Chapter 7 presents normalization techniques for relational tables. Chapter 8 presents physical database design techniques.

REVIEW CONCEPTS

- System: related components that work together to accomplish objectives
- Information system: system that accepts, processes, and produces data
- Waterfall model of information systems development: reference framework for activities in the information systems development process
- Spiral development methodologies, rapid application development methodologies, and Agile development methodologies to alleviate the problems in the traditional waterfall development approach
- Role of databases in information systems: provide permanent memory
- Define a common vocabulary to unify an organization
- Define business rules to support organizational processes
- Ensure data quality to improve the quality of decision making
- Evaluate investment in data quality using a cost-benefit approach
- Find an efficient implementation to ensure adequate performance while not compromising other design goals
- Conceptual data modeling to represent the information content independent of a target DBMS
- View design and view integration to manage the complexity of large data modeling efforts
- Logical database design to refine a conceptual data model to a target DBMS
- Distributed database design to determine locations of data and processing to achieve an efficient and reliable implementation
- Physical database design to achieve efficient implementations on each computer site
- Develop prototype forms and reports to cross check among the database and applications using the database
- Soft skills for conceptual data modeling: qualitative, subjective, and people-oriented
- Hard skills for finding an efficient implementation: quantitative, objective, and data intensive
- Computer-aided software engineering (CASE) tools to improve productivity in the database development process
- Fundamental assistance of CASE tools: drawing and documenting
- Active assistance of CASE tools: analysis and prototyping

QUESTIONS

1. What is the relationship between a system and an infomation system?
2. Provide an example of a system that is not an information system.
3. For an information system of which you are aware, describe some of the components (input data, output data, people, software, hardware, and procedures).
4. Briefly describe some of the kinds of data in the database for the information system in question 3.
5. Describe the phases of the waterfall model.
6. Why is the waterfall model considered only a reference framework?
7. What are the shortcomings in the waterfall model?
8. What alternative methodologies have been proposed to alleviate the difficulties of the waterfall model?
9. What is the relationship of the database development process to the information systems development process?
10. What is a data model? Process model? Environment interaction model?
11. What is the purpose of prototyping in the information systems development process?
12. How is a database designer like a politician in establishing a common vocabulary?
13. Why should a database designer establish the meaning of data?
14. What factors should a database designer consider when choosing database constraints?
15. Why is data quality important?
16. Provide examples of data quality problems according to two characteristics mentioned in Section 2.2.3.
17. How does a database designer decide on the appropriate level of data quality?
18. Why is it important to find an efficient implementation?
19. What are the inputs and the outputs of the conceptual data modeling phase?
20. What are the inputs and the outputs of the logical database design phase?
21. What are the inputs and the outputs of the distributed database design phase?
22. What are the inputs and the outputs of the physical database design phase?
23. What does it mean to say that the conceptual data modeling phase and the logical database design phase are concerned with the information content of the database?
24. Why are there two phases (conceptual data modeling and logical database design) that involve the information content of the database?
25. What is the relationship of view design and view integration to conceptual data modeling?
26. What is a soft skill?
27. What phases of database development primarily involve soft skills?
28. What is a hard skill?
29. What phases of database development primarily involve hard skills?
30. What kind of background is appropriate for hard skills?
31. Why do large organizations sometimes have different people performing design phases dealing with information content and efficient implementation?

32. Why are CASE tools useful in the database development process?
33. What is the difference between front-end and back-end CASE tools?
34. What kinds of support can a CASE tool provide for drawing a database diagram?
35. What kinds of support can a CASE tool provide for documenting a database design?
36. What kinds of support can a CASE tool provide for analyzing a database design?
37. What kinds of support can a CASE tool provide for prototyping?
38. Should you expect to find one software vendor providing a full range of functions (drawing, documenting, analyzing, and prototyping) for the database development process? Why or why not?
39. How has data quality moved beyond an issue just for the design of individual databases and data integration efforts?

PROBLEMS

Because of the introductory nature of this chapter, there are no problems in this chapter. Problems appear at the end of chapters in Parts 3 and 4.

REFERENCES FOR FURTHER STUDY

For a more detailed description of the database development process, you can consult specialized books on database design such as Batini, Ceri, and Navathe (1992) and Teorey et al. (2005). For more details about data quality, consult specialized books about data quality including Loshin (2011), Olson (2002), Redman (2001) along with the International Conference on Information Quality (mitiq.mit.edu/ICIQ/2015).

part two

Understanding Relational Databases

The chapters in Part 2 provide a detailed introduction to the Relational Data Model to instill a foundation for database design and application development with relational databases. Chapter 3 presents data definition concepts and retrieval operators for relational databases. Chapter 4 demonstrates SQL retrieval and modification statements for problems of basic and intermediate complexity and emphasizes problem solving guidelines to develop query formulation skills.

chapter

3

The Relational Data Model

Learning Objectives

This chapter provides the foundation for using relational databases. After this chapter, the student should have acquired the following knowledge and skills:

- Recognize relational database terminology
- Understand the meaning of the integrity rules for relational databases
- Understand the impact of referenced rows on maintaining relational databases
- Understand the meaning of each relational algebra operator
- List tables that must be combined to obtain desired results for simple retrieval requests

OVERVIEW

The chapters in Part 1 provided a starting point for your exploration of database technology and your understanding of the database development process. You broadly learned about database characteristics, DBMS features, the goals of database development, and the phases of the database development process. This chapter narrows your focus to the relational data model. Relational DBMSs dominate the market for business DBMSs. You will undoubtedly use relational DBMSs throughout your career as an information systems professional. This chapter provides background so that you will become proficient in designing databases and developing applications for relational databases in later chapters.

To use a relational database effectively, you need two kinds of knowledge. First, you need to understand the structure and contents of database tables. Understanding the connections among tables is especially critical because most database retrievals involve multiple tables. To help you understand relational databases, this chapter presents the basic terminology, the integrity rules, and a notation to visualize connections among tables. Second, you need to understand the operators of relational algebra as they are the building blocks of most commercial query languages. Understanding the operators will improve your knowledge of the SQL SELECT statement, the most widely used query language statement. You will learn the details for the SQL SELECT statement in Chapter 4. To help you understand the meaning of each operator, this chapter provides a visual representation of each operator and several convenient summaries.

3.1 BASIC ELEMENTS

Relational databases promised familiarity, simplicity, and mathematical rigor to manage data. Because many disciplines use tables to communicate complex ideas, the terminology of tables, rows, and columns is familiar to most users. During the early years of relational databases (1970s), the simplicity and familiarity of relational databases had strong appeal especially as compared to the procedural orientation of other data models. In addition to the familiarity and simplicity of relational databases, a strong mathematical basis also exists. The mathematics of relational databases involve conceptualizing tables as sets. The combination of familiarity and simplicity with a mathematical foundation provided a powerful combination enabling commercial dominance of relational DBMSs.

This section presents the basic terminology of relational databases and introduces the CREATE TABLE statement of the Structured Query Language (SQL). Sections 3.2 through 3.4 provide more detail about the elements defined in this section.

3.1.1 Tables

A relational database consists of a collection of tables. Each table has a heading or definition part and a body or content part. The heading part consists of the table name and the column names. For example, a student table may have columns for student number, name, street address, city, state, zip, class (freshman, sophomore, etc.), major, and cumulative grade point average (GPA). The body shows the rows of the **table**. Each row in a student table represents a student enrolled at a university. A student table for a major university may have more than 30,000 rows, too many to view at one time.

Table
a two dimensional arrangement of data. A table consists of a heading defining the table name and column names and a body containing rows of data.

To understand a table, it is also useful to view some of its rows. A table listing or datasheet shows the column names in the first row and the body in the other rows. Table 3-1 shows a table listing for the *Student* table. Three sample rows representing university students are displayed. In this book, the naming convention for column names uses a table abbreviation (*Std*) followed by a descriptive name. Because column names are often used without identifying the associated tables, the abbreviation supports easy table association. Mixed case highlights the different parts of a column name.

A CREATE TABLE statement can be used to define the heading part of a table. CREATE TABLE is a statement in the Structured Query Language (SQL). Because SQL is an industry standard language, the CREATE TABLE statement can be used to create tables in most DBMSs. The CREATE TABLE statement that follows[1] creates the *Student*table. For each column, the column name and data type are specified. **Data types** indicate the kind of data (character, numeric, Yes/No, etc.) and permissible operations (numeric operations, text operations, etc.) for the column. Each data type has a name (for example, CHAR for character) and usually a length specification. Table 3-2 lists common data types[2] used in relational DBMSs.

Data Type
defines a set of values and permissible operations on the values. Each column of a table is associated with a data type.

TABLE 3-1
Sample Table Listing of the *Student* Table

StdNo	StdFirstName	StdLastName	StdCity	StdState	StdZip	StdMajor	StdClass	StdGPA
123-45-6789	HOMER	WELLS	SEATTLE	WA	98121-1111	IS	FR	3.00
124-56-7890	BOB	NORBERT	BOTHELL	WA	98011-2121	FIN	JR	2.70
234-56-7890	CANDY	KENDALL	TACOMA	WA	99042-3321	ACCT	JR	3.50

[1] The CREATE TABLE statements in this chapter conform to the standard SQL syntax. There are slight syntax differences for most commercial DBMSs.

[2] Data types are not standard across relational DBMSs. The data types used in this chapter are specified in the latest SQL standard. Most DBMSs support these data types although the data type names may differ.

```
CREATE TABLE Student
( StdNo           CHAR(11),
  StdFirstName    VARCHAR(50),
  StdLastName     VARCHAR(50),
  StdCity         VARCHAR(50),
  StdState        CHAR(2),
  StdZip          CHAR(10),
  StdMajor        CHAR(6),
  StdClass        CHAR(6),
  StdGPA          DECIMAL(3,2))
```

3.1.2 Connections among Tables

It is not enough to understand each table individually. To understand a relational database, connections or **relationships** among tables also must be understood. The rows in a table are usually related to rows in other tables. Matching (identical) values indicate relationships between tables. Consider the sample *Enrollment* table (Table 3-3) in which each row represents a student enrolled in an offering of a course. The values in the *StdNo* column of the *Enrollment* table match the *StdNo* values in the sample *Student* table (Table 3-1). For example, the first and third rows of the *Enrollment* table have the same *StdNo* value (123-45-6789) as the first row of the S*tudent* table. Likewise, the values in the *OfferNo* column of the *Enrollment* table match the *OfferNo* column in the *Offering* table (Table 3-4). Figure 3.1 shows a graphical depiction of the matching values.

Relationship
connection between rows in two tables. Relationships are shown by column values in one table that match column values in another table.

TABLE 3-2
Brief Description of Common SQL Data Types

Data Type	Description
CHAR(L)	For fixed length text entries such as state abbreviations and fixed length postal codes. Each column value using CHAR contains the maximum number of characters (*L*) even if the actual length is shorter. Most DBMSs have an upper limit on the length (*L*) such as 255.
VARCHAR(L)	For variable length text such as names and street addresses. Column values using VARCHAR contain only the actual number of characters, not the maximum length for CHAR columns. Most DBMSs have an upper limit on the length such as 255.
FLOAT(P)	For columns containing numeric data with a floating precision such as interest rate calculations and scientific calculations. The precision parameter *P* indicates the number of significant digits. Most DBMSs have an upper limit on *P* such as 38. Some DBMSs have two data types, REAL and DOUBLE PRECISION, for low- and high-precision floating point numbers instead of the variable precision with the FLOAT data type.
DATE/TIME	For columns containing dates and times such as an order date. These data types are not standard across DBMSs. Some systems support three data types (DATE, TIME, and TIMESTAMP) while other systems support a combined data type (DATE) storing both the date and time.
DECIMAL(W,R)	For columns containing numeric data with a fixed precision such as monetary amounts. The *W* value indicates the total number of digits and the *R* value indicates the number of digits to the right of the decimal point. This data type is also called NUMERIC in some DBMSs.
INTEGER	For columns containing whole numbers (numbers without a decimal point). Some DBMSs have the SMALLINT data type for very small whole numbers and the LONG data type for very large integers.
BOOLEAN	For columns containing data with two values such as true/false or yes/no.

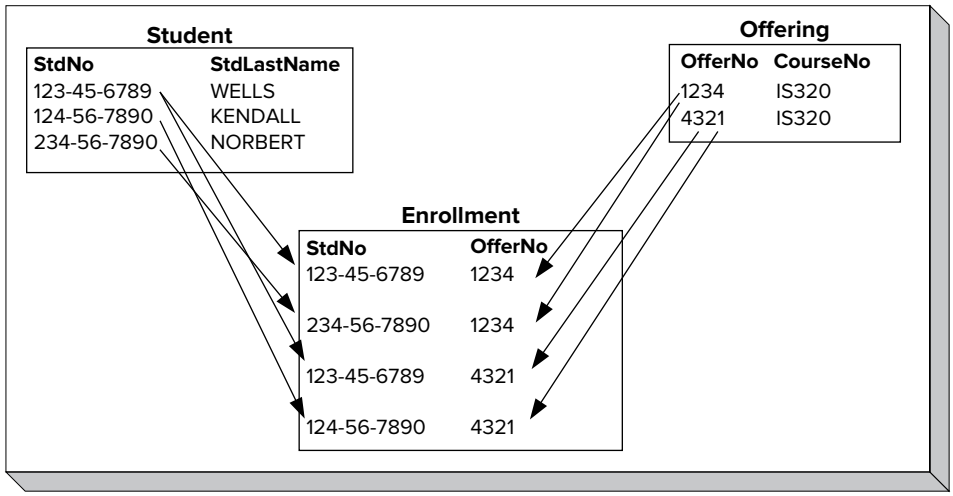
TABLE 3-3
Sample *Enrollment* Table

OfferNo	StdNo	EnrGrade
1234	123-45-6789	3.3
1234	234-56-7890	3.5
4321	123-45-6789	3.5
4321	124-56-7890	3.2

TABLE 3-4
Sample *Offering* Table

OfferNo	CourseNo	OffTerm	OffYear	OffLocation	OffTime	FacNo	OffDays
1111	IS320	SUMMER	2017	BLM302	10:30 AM		MW
1234	IS320	FALL	2016	BLM302	10:30 AM	098-76-5432	MW
2222	IS460	SUMMER	2016	BLM412	1:30 PM		TTH
3333	IS320	SPRING	2017	BLM214	8:30 AM	098-76-5432	MW
4321	IS320	FALL	2016	BLM214	3:30 PM	098-76-5432	TTH
4444	IS320	SPRING	2017	BLM302	3:30 PM	543-21-0987	TTH
5678	IS480	SPRING	2017	BLM302	10:30 AM	987-65-4321	MW
5679	IS480	SPRING	2017	BLM412	3:30 PM	876-54-3210	TTH
9876	IS460	SPRING	2017	BLM307	1:30 PM	654-32-1098	TTH

FIGURE 3.1
Matching Values among the Enrollment, Offering, and Student Tables



The concept of matching values is crucial in relational databases. As you will see, relational databases typically contain many tables. Even a modest-size database can have 10 to 15 tables. Major databases for business and government organizations contain hundreds of tables. To extract meaningful information, you must combine multiple tables using matching values. By matching on *Student.StdNo* and *Enrollment .StdNo*, you could combine the *Student* and *Enrollment* tables[3]. Similarly, by matching on *Enrollment.OfferNo* and *Offering.OfferNo*, you could combine the *Enrollment* and *Offering* tables. As you will see later in this chapter, the join operator combines tables on matching values. Understanding the connections between tables (or columns on which tables can be combined) is crucial for extracting useful data.

[3] When columns have identical names in two tables, it is customary to precede the column name with the table name and a period as *Student.StdNo* and *Enrollment.StdNo*.

3.1.3 Alternative Terminology

You should be aware about other commonly used terminology besides table, row, and column. Table 3-5 shows three roughly equivalent terminologies. The divergence in terminology is due to different groups that use databases. The table-oriented terminology appeals to end users; the set-oriented terminology appeals to academic researchers; and the record-oriented terminology appeals to information systems professionals. In practice, these terms may be mixed. For example, in the same sentence you might hear both "tables" and "fields." You should expect to see a mix of terminology in your career.

TABLE 3-5
Alternative Terminology for Relational Databases

Table-Oriented	Set-Oriented	Record-Oriented
Table	Relation	Record type, file
Row	Tuple	Record
Column	Attribute	Field

3.2 INTEGRITY RULES

In the previous section, you learned that a relational database consists of a collection of interrelated tables. To ensure that a database provides meaningful information, integrity rules are necessary. This section describes two important integrity rules (entity integrity and referential integrity), examples of their usage, and a notation to visualize referential integrity.

3.2.1 Definition of the Integrity Rules

Entity integrity[4] means that each table must have a column or combination of columns with unique values. Unique means that no two rows of a table have the same value. For example, *StdNo* in *Student* is unique and the combination of *StdNo* and *OfferNo* is unique in *Enrollment*. Entity integrity ensures unique identification of entities (people, things, places, and events) in a database. For auditing, security, and communication reasons, business entities must be easily traceable.

Referential integrity means that the column values in one table must match column values in a related table. For example, the value of *StdNo* in each row of the *Enrollment* table must match the value of *StdNo* in some row of the *Student* table. Referential integrity ensures that a database contains valid connections. For example, it is critical that each row of the *Enrollment* table contains a student number of a valid student. Otherwise, some enrollments can be meaningless, possibly resulting in students denied enrollment because non existing students took their places.

For more precise definitions of entity integrity and referential integrity, some other definitions are necessary. These prerequisite definitions and the more precise definitions follow.

Definitions

- Superkey: a column or combination of columns containing unique values for each row. The combination of every column in a table is always a superkey because rows in a table must be unique[5].
- Candidate key: a minimal superkey. A superkey is minimal if removing any column makes it no longer unique. A single column superkey is minimal because no columns can be removed.
- Null value: a special value that represents the absence of an actual value. A null value can mean that the actual value is unknown or does not apply to a specified row.

[4] Entity integrity is also known as uniqueness integrity.

[5] The SQL standard does not require uniqueness of rows although uniqueness is a basic tenet of the relational model.

- Primary key: a specially designated candidate key. The primary key of a table cannot contain null values. Each table contains one primary key.
- Foreign key: a column or combination of columns in which the values must match those of a candidate key. A foreign key must have the same data type as its associated candidate key. In the CREATE TABLE statement of SQL, a foreign key must be associated with a primary key rather than merely a candidate key.

Integrity Rules

- Entity integrity rule: No two rows of a table can contain the same value for the primary key. In addition, no row can contain a null value for any column of a primary key.
- Referential integrity rule: Only two kinds of values can be stored in a foreign key:
  - a value matching a candidate key value in some row of the table containing the associated candidate key or
  - a null value.

3.2.2 Application of the Integrity Rules

To extend your understanding, let us apply the integrity rules to several tables in the university database. The primary key of *Student* is *StdNo*. You specify a primary key as part of the CREATE TABLE statement. To designate *StdNo* as the primary key of *Student*, you use a CONSTRAINT clause for the primary key at the end of the CREATE TABLE statement. The constraint name (*PKStudent*) following the CONSTRAINT keyword facilitates identification of the constraint if a violation occurs when a row is inserted or updated.

```
CREATE TABLE Student
( StdNo         CHAR(11),
  StdFirstName  VARCHAR(50),
  StdLastName   VARCHAR(50),
  StdCity       VARCHAR(50),
  StdState      CHAR(2),
  StdZip        CHAR(10),
  StdMajor      CHAR(6),
  StdClass      CHAR(2),
  StdGPA        DECIMAL(3,2),
CONSTRAINT PKStudent PRIMARY KEY (StdNo) )
```

Many organizations including universities in the U.S.A. previously used Social Security numbers as unique identifiers. Because of the increase in identity theft, most organizations have eliminated the usage of government identifiers such as Social Security numbers as primary keys. Instead, organizations typically use unique identifiers specific to an organization. For example, an organization may generate unique customer numbers, product numbers, and employee numbers. In these cases, automatic generation of unique values is required. Most DBMSs support automatic generation of unique values as explained in Appendix 3.C. In some situations, an organization uses an external identifier already possessed by an individual such as an email address.

Entity Integrity Variations The UNIQUE keyword designates candidate keys that are not primary keys. The *Course* table (see Table 3-6) contains two candidate keys: *CourseNo* (primary key) and *CrsDesc* (course description). The *CourseNo* column is the primary key because it is more stable than the *CrsDesc* column. Course descriptions may change over time, but course numbers remain the same. In addition, course numbers are shorter requiring less space to store in related tables.

```
CREATE TABLE Course
( CourseNo    CHAR(6),
  CrsDesc     VARCHAR(250),
  CrsUnits    SMALLINT,
CONSTRAINT PKCourse PRIMARY KEY(CourseNo),
CONSTRAINT UniqueCrsDesc UNIQUE (CrsDesc))
```

Some tables need more than one column in the primary key. In the *Enrollment* table, the primary key consists of the combination of *StdNo* and *OfferNo*. You must provide values for both columns to identify a row. A composite or combined primary contains more than one column. In the CREATE TABLE statement for the *Enrollment* table, you should note that both *OfferNo* and *StdNo* appear inside the parentheses following the PRIMARY KEY keywords.

```
CREATE TABLE Enrollment
( OfferNo     INTEGER,
  StdNo       CHAR(11),
  EnrGrade    DECIMAL(3,2),
CONSTRAINT PKEnrollment PRIMARY KEY(OfferNo, StdNo) )
```

Superkeys that are not candidate keys are not important. Recall that a candidate key is a minimal superkey. Nonminimal superkeys are usually ignored because they are common and contain columns that do not contribute to the uniqueness property. For example, the combination of *StdNo* and *StdLastName* is unique so it is a superkey. However, if *StdLastName* is removed, *StdNo* is still unique so the combination of *StdNo* and *StdLastName* is not minimal and hence not a candidate key. Thus, the superkey with *StdNo* and *StdLastName* is not important.

Referential Integrity For referential integrity, the columns *StdNo* and *OfferNo* are foreign keys in the *Enrollment* table. The *StdNo* column refers to the *Student* table and the *OfferNo* column refers to the *Offering* table (Table 3-4). An *Offering* row represents a course given in an academic period (summer, winter, etc.), year, time, location, and

TABLE 3-6
Sample *Course* Table

CourseNo	CrsDesc	CrsUnits
IS320	FUNDAMENTALS OF BUSINESS PROGRAMMING	4
IS460	SYSTEMS ANALYSIS	4
IS470	BUSINESS DATA COMMUNICATIONS	4
IS480	FUNDAMENTALS OF DATABASE MANAGEMENT	4

days of the week. The primary key of *Offering* is *OfferNo*. A course such as IS480 will have different offer numbers each time it is taught.

You can define referential integrity constraints similarly to the way of defining primary keys. For example, to define the foreign keys in *Enrollment*, you should use CONSTRAINT clauses for foreign keys at the end of the CREATE TABLE statement as shown in the revised CREATE TABLE statement for the *Enrollment* table.

```
CREATE TABLE Enrollment
( OfferNo     INTEGER,
  StdNo       CHAR(11),
  EnrGrade    DECIMAL(3,2),
CONSTRAINT PKEnrollment PRIMARY KEY(OfferNo, StdNo),
CONSTRAINT FKOfferNo FOREIGN KEY (OfferNo) REFERENCES Offering,
CONSTRAINT FKStdNo FOREIGN KEY (StdNo) REFERENCES Student )
```

Although referential integrity permits foreign keys to have null values, it is not common for foreign keys to have null values. When a foreign key is part of a primary key, null values are not permitted because of the entity integrity rule. For example, null values are not permitted for either *Enrollment.StdNo* or *Enrollment.OfferNo* because each column is part of the primary key.

When a foreign key is not part of a primary key, organizational practice dictates if null values should be permitted. For example, *Offering.CourseNo*, a foreign key referring to *Course* (Table 3-4), is not part of a primary key, yet null values are not permitted. In most universities, a course cannot be offered before it is approved. Thus, an offering should not be inserted without a related course.

The NOT NULL keywords indicate that a column cannot have null values as shown in the CREATE TABLE statement for the *Offering* table. The NOT NULL constraints are inline constraints associated with a specific column. In contrast, the primary and foreign key constraints in the CREATE TABLE statement for the *Offering* table are table constraints in which the associated columns must be specified in the constraint. Constraint names should be used with both table and inline constraints to facilitate identification when a violation occurs. Without using a meaningful constraint name, it is difficult to identify the constraint and understand the constraint violation.

```
CREATE TABLE Offering
( OfferNo      INTEGER,
  CourseNo     CHAR(6)      CONSTRAINT OffCourseNoRequired NOT
                            NULL,
  OffLocation  VARCHAR(50),
  OffDays      CHAR(6),
  OffTerm      CHAR(6)      CONSTRAINT OffTermRequired NOT NULL,
  OffYear      INTEGER      CONSTRAINT OffYearRequired NOT NULL,
  FacNo        CHAR(11),
  OffTime      DATE,
```

```
CONSTRAINT PKOffering   PRIMARY KEY (OfferNo),
CONSTRAINT FKCourseNo   FOREIGN KEY(CourseNo) REFERENCES Course,
CONSTRAINT FKFacNo      FOREIGN KEY(FacNo) REFERENCES Faculty )
```

In contrast, *Offering.FacNo* referring to the faculty member teaching the offering, may be null. A null value for *Offering.FacNo* means that a faculty member is not yet assigned to teach the offering. For example, an instructor is not assigned in the first and third rows of Table 3-4. Because offerings must be scheduled perhaps a year in advance, it is likely that instructors for some offerings will not be known until after the offering row is initially stored. Therefore, permitting null values in the *Offering* table is prudent.

Referential Integrity for Self-Referencing (Unary) Relationships A referential integrity constraint involving a single table is known as a **self-referencing relationship** or unary relationship. Self-referencing relationships are not common, but they are important in certain business situations. In the university database, a faculty member can supervise other faculty members and be supervised by a faculty member. For example, Victoria Emmanuel (second row) supervises Leonard Fibon (third row) in the sample *Faculty* table found in Table 3-7. The *FacSupervisor* column shows this relationship: the *FacSupervisor* value in the third row (543-21-0987) matches the *FacNo* value in the second row. Note that null values for *FacSupervisor* represent faculty without supervisors. The second row (Victoria Emmanuel) and fourth row (Nicki Macon) are faculty without supervisors.

Self-Referencing Relationship
a relationship in which a foreign key refers to the same table. Self-referencing relationships represent associations among members of the same set.

A referential integrity constraint involving the *FacSupervisor* column represents the self-referencing relationship. In the CREATE TABLE statement, the referential integrity constraint for a self-referencing relationship uses the same table (*Faculty*) following the REFERENCES keyword.

```
CREATE TABLE Faculty
( FacNo            CHAR(11),
  FacFirstName     VARCHAR(50)  CONSTRAINT FacFirstNameRequired NOT NULL,
  FacLastName      VARCHAR(50)  CONSTRAINT FacLastNameRequired NOT NULL,
  FacCity          VARCHAR(50)  CONSTRAINT FacCityRequired NOT NULL,
  FacState         CHAR(2)      CONSTRAINT FacStateRequired NOT NULL,
  FacZipCode       CHAR(10)     CONSTRAINT FacZipRequired NOT NULL,
  FacHireDate      DATE,
  FacDept          CHAR(6),
  FacRank          CHAR(4),
  FacSalary        DECIMAL(10,2),
  FacSupervisor    CHAR(11),
CONSTRAINT PKFaculty PRIMARY KEY (FacNo),
CONSTRAINT FKFacSupervisor FOREIGN KEY (FacSupervisor) REFERENCES Faculty )
```

TABLE 3-7
Sample *Faculty* Table

FacNo	FacFirstName	FacLastName	FacCity	FacState	FacDept	FacRank	FacSalary	FacSupervisor	FacHireDate	FacZipCode
098-76-5432	LEONARD	VINCE	SEATTLE	WA	MS	ASST	$35,000	654-32-1098	10-Apr-2004	98111-9921
543-21-0987	VICTORIA	EMMANUEL	BOTHELL	WA	MS	PROF	$120,000		15-Apr-2005	98011-2242
654-32-1098	LEONARD	FIBON	SEATTLE	WA	MS	ASSC	$70,000	543-21-0987	01-May-2003	98121-0094
765-43-2109	NICKI	MACON	BELLEVUE	WA	FIN	PROF	$65,000		11-Apr-2006	98015-9945
876-54-3210	CRISTOPHER	COLAN	SEATTLE	WA	MS	ASST	$40,000	654-32-1098	01-Mar-2008	98114-1332
987-65-4321	JULIA	MILLS	SEATTLE	WA	FIN	ASSC	$75,000	765-43-2109	15-Mar-2009	98114-9954

3.2.3 Graphical Representation of Referential Integrity

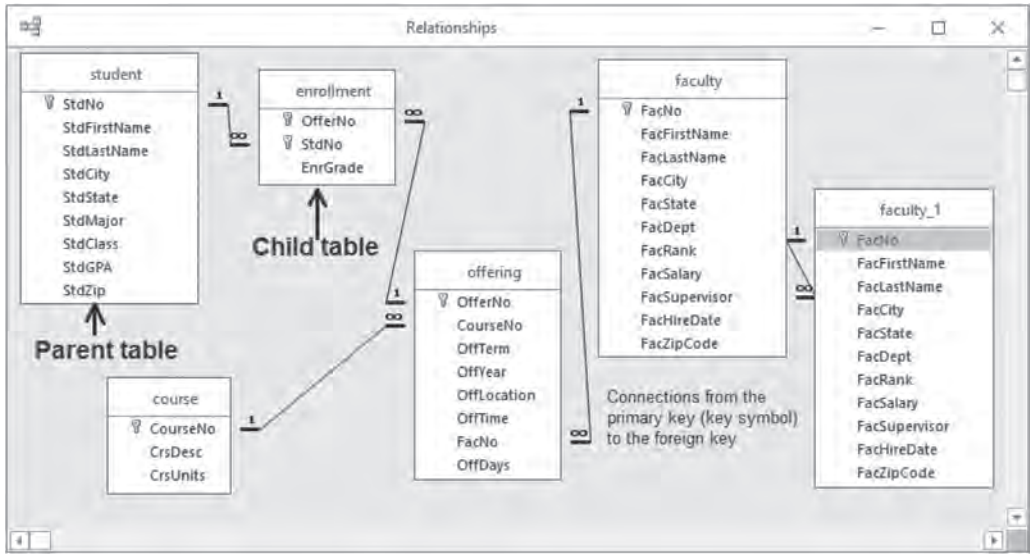
In recent years, commercial DBMSs have provided graphical representations for referential integrity constraints. The graphical representation makes referential integrity easier to define and understand than the text representation in the CREATE TABLE statement. In addition, a graphical representation supports nonprocedural data access.

To depict a graphical representation, let us study the Relationship window in Microsoft Access. Access provides the Relationship window to visually define and display referential integrity constraints. Figure 3.2 shows the Relationship window for the tables of the university database. Each line represents a referential integrity constraint or relationship. In a relationship, the primary key table is known as the parent or "1" table (for example, *Student*) and the foreign key table (for example, *Enrollment*) is known as the child or "M" (many) table.

The relationship from *Student* to *Enrollment* is called "1-M" (one-to-many) because a student can be related to many enrollments but an enrollment can be related to only one student. Similarly, the relationship from the *Offering* table to the *Enrollment* table means that an offering can be related to many enrollments but an enrollment can be related to only one offering. You should practice by writing similar sentences for the other relationships in Figure 3.2.

M-N (many-to-many) relationships are not directly represented in the Relational Model. An M-N relationship means that rows from each table can be related to many rows of the other table. For example, a student enrolls in many course offerings and a course offering contains many students. In the Relational Model, a pair of

FIGURE 3.2
Relationship Window for the University Database



1-M relationships and a linking or associative table represents an **M-N relationship**. In Figure 3.2, the linking table *Enrollment* and its relationships with *Offering* and *Student* represent an M-N relationship between the *Student* and *Offering* tables.

Self-referencing relationships are represented indirectly in the Relationship window. The self-referencing relationship involving *Faculty* is represented as a relationship between the *Faculty* and *Faculty_1* tables. *Faculty_1* is not a real table as it is created only inside the Access Relationship window. Access can only indirectly show self-referencing relationships.

A graphical representation such as the Relationship window makes it easy to identify tables that should be combined to answer a retrieval request. For example, assume that you want to find instructors who teach courses with "database" in the course description. Clearly, you need the *Course* table to find "database" courses. You also need the *Faculty* table to display instructor data. Figure 3.2 shows that you also need the *Offering* table because *Course* and *Faculty* are not directly connected. Rather, *Course* and *Faculty* are connected through *Offering*. Thus, visualizing relationships helps to identify tables needed to fulfill retrieval requests. Before attempting the retrieval problems in later chapters, you should carefully study a graphical representation of the relationships. You should construct your own diagram if one is not available.

1-M Relationship
a connection between two tables in which one row of a parent table can be referenced by many rows of a child table. 1-M relationships are the most common kind of relationship.

M-N Relationship
a connection between two tables in which rows of one table can be related to many rows of the other table. M-N relationships cannot be directly represented in the Relational Model. Two 1-M relationships and a linking or associative table represent an M-N relationship.

3.3 DELETE AND UPDATE ACTIONS FOR REFERENCED ROWS

For each referential integrity constraint, you should carefully consider actions on referenced rows in parent tables of 1-M relationships. A parent row is referenced if there are rows in a child table with foreign key values identical to the primary key value of the parent table row. For example, the first row of the *Course* table (Table 3-6) with *CourseNo* "IS320" is referenced by the first row of the *Offering* table (Table 3-4). It is natural to consider the impact on related *Offering* rows when deleting the referenced *Course* row or updating the *CourseNo* value. More generally, these concerns can be stated as

Deleting a referenced row: What happens to related rows (that is, rows in the child table with the identical foreign key value) when deleting the referenced row in the parent table?
Updating the primary key of a referenced row: What happens to related rows when updating the primary key of the referenced row in the parent table?

Actions on referenced rows are important when changing the rows of a database. When developing data entry forms (discussed in Chapter 10), actions on referenced rows can be especially important. For example, if a data entry form permits deletion of rows in the *Course* table, actions on related rows in the *Offering* table must be carefully planned. Otherwise, the database can become inconsistent or difficult to use.

Possible Actions

There are several possible actions in response to the deletion of a referenced row or the update of the primary key of a referenced row. The appropriate action depends on organizational practices and tables involved. The following list describes the actions and provides examples of usage.

- **Restrict**[6]: Do not allow the action on the referenced row. For example, do not permit a *Student* row to be deleted if there are any related *Enrollment* rows.

[6] There is a related action designated by the keywords NO ACTION. The difference between RESTRICT and NO ACTION involves the concept of deferred integrity constraints, discussed in Chapter 17.

Similarly, do not allow *Student.StdNo* to be updated if there are related *Enrollment* rows.

- **Cascade**: Perform the same action (cascade the action) on related rows. For example, if a *Student* is deleted, then delete the related *Enrollment* rows. Likewise, if *Student.StdNo* is changed in some row, update *StdNo* in the related *Enrollment* rows.
- **Nullify**: Set the foreign key of related rows to null. For example, if a *Faculty* row is deleted, then set *FacNo* to NULL in related *Offering* rows. Likewise, if *Faculty.FacNo* is updated, then set *FacNo* to NULL in related *Offering* rows. The nullify action is valid only if the foreign key allows null values. For example, the nullify option is not valid when deleting rows of the *Student* table because *Enrollment.StdNo* is part of the primary key of *Enrollment*.
- **Default**: Set the foreign key of related rows to its default value. For example, if a *Faculty* row is deleted, then set *FacNo* to a default faculty number in related *Offering* rows. The default faculty number might have an interpretation such as "to be announced". Likewise, if *Faculty.FacNo* is updated, then set *FacNo* to its default value in related *Offering* rows. The default action is an alternative to the nullify action as the default action avoids null values.

The delete and update actions can be specified in the CREATE TABLE statement using the ON DELETE and ON UPDATE clauses. These clauses are part of foreign key constraints. For example, the revised CREATE TABLE statement for the *Enrollment* table shows ON UPDATE clauses for the *Enrollment* table. The ON DELETE clause is not used because the default is to restrict deletions with referenced rows[7]. The keywords CASCADE, SET NULL, and SET DEFAULT can be used to specify the second through fourth options, respectively.

```
CREATE TABLE Enrollment
(  OfferNo          INTEGER,
   StdNo            CHAR(11),
   EnrGrade         DECIMAL(3,2),
CONSTRAINT PKEnrollment PRIMARY KEY(OfferNo, StdNo),
CONSTRAINT FKOfferNo FOREIGN KEY (OfferNo) REFERENCES Offering
   ON UPDATE CASCADE,
CONSTRAINT FKStdNo FOREIGN KEY (StdNo) REFERENCES Student
   ON UPDATE CASCADE )
-- ON UPDATE is not valid Oracle SQL syntax but valid SQL:2016 syntax
```

Before finishing this section, you should understand the impact of referenced rows on insert operations. A referenced row must be inserted before its related rows. For example, before inserting a row in the *Enrollment* table, the referenced rows in the *Student* and *Offering* tables must exist. Referential integrity places an ordering on insertion of rows from different tables. When designing data entry forms, you should carefully consider the impact of referential integrity on the order that users complete forms.

[7] Note that the ON UPDATE and RESTRICT keywords are not valid syntax in Oracle. Oracle does not provide syntax for the restrict action as the restrict action is default. The *Enrollment* table example is not valid Oracle syntax because of the ON UPDATE clauses, but it is valid SQL:2016 syntax.

3.4 OPERATORS OF RELATIONAL ALGEBRA

In previous sections of this chapter, you studied the terminology and integrity rules of relational databases with the goal of understanding existing relational databases. In particular, understanding connections among tables was emphasized as a prerequisite to retrieving useful information. This section describes some fundamental operators that can be used to retrieve useful data from a relational database.

You can think of relational algebra similarly to the algebra of numbers except that the objects are different: algebra applies to numbers and relational algebra applies to tables. In algebra, each operator transforms one or more numbers into another number. Similarly, each operator of relational algebra transforms a table (or two tables) into a new table.

This section emphasizes the study of each relational algebra operator in isolation. For each operator, you should understand its purpose and inputs. While it is possible to combine operators to make complicated formulas, this level of understanding is not important for developing query formulation skills. Using relational algebra by itself to write queries can be awkward because of details such as ordering of operations and parentheses. Therefore, you should seek only to understand the meaning of each operator, not how to combine operators to write expressions. In Chapter 4, you will learn the SELECT statement of SQL to perform retrievals that would involve complex combinations of relational algebra operators.

The coverage of relational algebra groups the operators into three categories. The most widely used operators (restrict, project, and join) are presented first. The extended cross product operator is also presented to provide background for the join operator. Knowledge of these operators will help you to formulate a large percentage of queries. More specialized operators are covered in latter parts of the section. The more specialized operators include the traditional set operators (union, intersection, and difference) and advanced operators (summarize and divide). Knowledge of these operators will help you formulate more difficult queries.

3.4.1 Restrict (Select) and Project Operators

The restrict[8] (also known as select) and project operators produce subsets of a table. Because users often want to see a subset rather than an entire table, these operators are widely used. These operators are also popular because they are easy to understand.

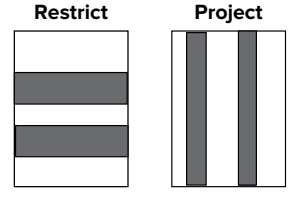
Restrict
an operator that retrieves a subset of the rows of the input table that satisfy a given condition.

Project
an operator that retrieves a specified subset of the columns of the input table.

The restrict and project operators produce an output table that is a subset of an input table (Figure 3.3). **Restrict** produces a subset of the rows, while **project** produces a subset of the columns. Restrict uses a condition or logical expression to indicate the rows to retain in the output. Project uses a list of column names to indicate the columns to retain in the output. Restrict and project are often used together because tables can have many rows and columns. It is rare that a user wants to see all rows and columns.

The logical expression used in the restrict operator can include comparisons involving columns and constants. Complex logical expressions can be formed using the logical operators AND, OR, and NOT. For example, Table 3-8 shows the result of a restrict operation on Table 3-4 where the logical expression is: OffDays = 'MW' AND OffTerm = 'SPRING' AND OffYear = 2017.

FIGURE 3.3
Graphical Representation of Restrict and Project Operators



A project operation can have a side effect. Sometimes after retrieving a subset of columns, duplicate rows exist. When this occurs, the project operator removes the duplicate rows. For example, if *Offering.CourseNo* is the only column used in a project operation, only three rows are in the result (Table 3-9) even though the *Offering* table (Table 3-4) has nine rows. The column *Offering.CourseNo* contains only three unique values in Table 3-4. Note that if the primary key or a candidate key is included in the list of columns, the resulting table has no duplicates. For example, if *OfferNo* was

[8]In this book, the operator name restrict is used to avoid confusion with the SQL SELECT statement. The operator is more widely known as select.

TABLE 3-8
Result of Restrict Operation on the Sample *Offering* Table (Table 3-4)

OfferNo	CourseNo	OffTerm	OffYear	OffLocation	OffTime	FacNo	OffDays
3333	IS320	SPRING	2017	BLM214	8:30 AM	098-76-5432	MW
5678	IS480	SPRING	2017	BLM302	10:30 AM	987-65-4321	MW

TABLE 3-9
Result of a Project Operation on *Offering.CourseNo*

CourseNo
IS320
IS460
IS480

included in the list of columns, the result table would have nine rows with no duplicate removal necessary.

This side effect is due to the mathematical nature of relational algebra. In relational algebra, tables are considered sets. Because sets do not have duplicates, duplicate removal is a possible side effect of the project operator. Commercial languages such as SQL usually take a more pragmatic view. Because duplicate removal requires a reasonable level of computing resources, a user must explicitly indicate removal of duplicates.

3.4.2 Extended Cross Product Operator

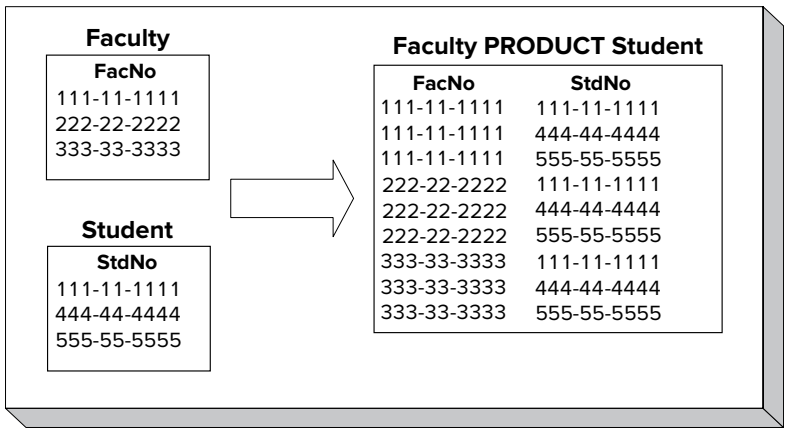
The extended cross product operator can combine any two tables. Other table combining operators have conditions about the tables to combine. Because of its unrestricted nature, the extended cross product operator can produce tables with excessive data. The extended cross product operator is important because it is a building block for the join operator. When you initially learn the join operator, knowledge of the extended cross product operator can be useful. After you gain experience with the join operator, you will not need to rely on the extended cross product operator.

Extended Cross Product an operator that builds a table consisting of all combinations of rows from each of the two input tables.

The **extended cross product**[9] (product for short) operator shows everything possible from two tables. The product of two tables is a new table consisting of all possible combinations of rows from the two input tables. Figure 3.4 depicts a product of two single column tables. Each result row consists of the columns of the *Faculty* table (only *FacNo*) and the columns of the *Student* table (only *StdNo*). The name of the operator (product) derives from the number of rows in the result. The number of rows in the resulting table is the product of the number of rows of the two input tables. In contrast, the number of result columns is the sum of the columns of the two input tables. In Figure 3.4, the result table has nine rows and two columns.

As another example, consider the product of the sample *Student* (Table 3-10) and *Enrollment* (Table 3-11) tables. The resulting table (Table 3-12) has 9 rows (3 × 3) and 7 columns (4 + 3). Note that most rows in the result are not meaningful as only three rows have the same value for *StdNo*.

FIGURE 3.4
Cross Product Example



[9]The extended cross product operator is also known as the Cartesian product after the French mathematician Rene´ Descartes.

TABLE 3-10
Sample *Student* Table

StdNo	StdLastName	StdMajor	StdClass
123-45-6789	WELLS	IS	FR
124-56-7890	NORBERT	FIN	JR
234-56-7890	KENDALL	ACCT	JR

TABLE 3-11
Sample *Enrollment* Table

OfferNo	StdNo	EnrGrade
1234	123-45-6789	3.3
1234	234-56-7890	3.5
4321	124-56-7890	3.2

TABLE 3-12
Student PRODUCT *Enrollment*

Student.StdNo	StdLastName	StdMajor	StdClass	OfferNo	Enrollment.StdNo	EnrGrade
123-45-6789	WELLS	IS	FR	1234	123-45-6789	3.3
123-45-6789	WELLS	IS	FR	1234	234-56-7890	3.5
123-45-6789	WELLS	IS	FR	4321	124-56-7890	3.2
124-56-7890	NORBERT	FIN	JR	1234	123-45-6789	3.3
124-56-7890	NORBERT	FIN	JR	1234	234-56-7890	3.5
124-56-7890	NORBERT	FIN	JR	4321	124-56-7890	3.2
234-56-7890	KENDALL	ACCT	JR	1234	123-45-6789	3.3
234-56-7890	KENDALL	ACCT	JR	1234	234-56-7890	3.5
234-56-7890	KENDALL	ACCT	JR	4321	124-56-7890	3.2

As these examples show, the extended cross product operator often generates excessive data. Excessive data are as bad as lack of data. For example, the product of a student table of 30,000 rows and an enrollment table of 300,000 rows is a table of nine billion rows! Most of these rows would be meaningless combinations. So it is rare that a cross product operation by itself is needed. Rather, the importance of the cross product operator is as a building block for other operators such as the join operator.

3.4.3 Join Operator

Join is the most widely used operator for combining tables. Because most databases have many tables, combining tables is important. Join differs from cross product because join requires a matching condition on rows of two tables. Most tables are combined in this way. To a large extent, your skill in retrieving useful data will depend on your ability to use the join operator.

Join
an operator that produces a table containing rows that match on a condition involving a column from each input table.

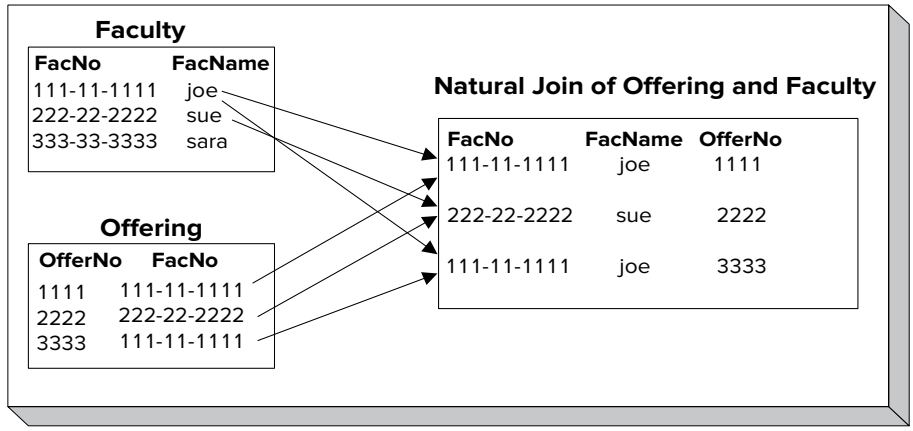
The **join** operator builds a new table by combining rows from two tables that match on a join condition. Typically, the join condition specifies that two rows have an identical value in one or more columns. When the join condition involves equality, the join is known as an equi-join, for equality join. Figure 3.5 shows a join of sample *Faculty* and *Offering* tables where the join condition is that the *FacNo* columns are equal. Note that only a few columns are shown to simplify the illustration. The arrows indicate the manner that rows from the input tables combine to form rows in the result table. For example, the first row of the *Faculty* table combines with the first and third rows of the *Offering* table to yield two rows in the result table.

Natural Join
a commonly used join operator where the matching condition is equality (equi-join), one of the matching columns is discarded in the result table, and the join columns have the same unqualified names.

The **natural join** operator is the most common join operation. In a natural join operation, the join condition is equality (equi-join), one of the join columns is removed, and the join columns have the same unqualified[10] name. In Figure 3.5, the result table contains only three columns because the natural join removes one of the *FacNo* columns. The particular column (*Faculty.FacNo* or *Offering.FacNo*) removed does not matter.

[10] An unqualified name is the column name without the table name. The full name of a column includes the table name. Thus, the full names of the join columns in Figure 3.5 are *Faculty.FacNo* and *Offering.FacNo*.

FIGURE 3.5
Sample Natural Join Operation



As another example, consider the natural join of *Student* (Table 3-13) and *Enrollment* (Table 3-14) shown in Table 3-15. In each row of the result, *Student.StdNo* matches *Enrollment.StdNo*. Only one of the join columns is included in the result. Arbitrarily, *Student.StdNo* is shown although *Enrollment.StdNo* could be shown instead without changing the result.

Derivation of the Natural Join The natural join operator is not primitive because it can be derived from other operators. The natural join operator consists of three steps:

(1) A product operation to combine the rows.
(2) A restrict operation to remove rows not satisfying the join condition.
(3) A project operation to remove one of the join columns.

To depict these steps, the first step to produce the natural join in Table 3-15 is the product result shown in Table 3-16. The second step is to retain only the matching rows (rows 1, 6, and 8 of Table 3-16). A restrict operation is used with Student.StdNo = Enrollment.StdNo as the restriction condition. The final step is to eliminate one of the join columns (*Enrollment.StdNo*). The project operation includes all columns except for *Enrollment.StdNo* (Table 3.-17).

TABLE 3-13
Sample *Student* Table

StdNo	StdLastName	StdMajor	StdClass
123-45-6789	WELLS	IS	FR
124-56-7890	NORBERT	FIN	JR
234-56-7890	KENDALL	ACCT	JR

TABLE 3-14
Sample *Enrollment* Table

OfferNo	StdNo	EnrGrade
1234	123-45-6789	3.3
1234	234-56-7890	3.5
4321	124-56-7890	3.2

TABLE 3-15
Natural Join of *Student* and *Enrollment*

Student.StdNo	StdLastName	StdMajor	StdClass	OfferNo	EnrGrade
123-45-6789	WELLS	IS	FR	1234	3.3
124-56-7890	NORBERT	FIN	JR	4321	3.2
234-56-7890	KENDALL	ACCT	JR	1234	3.5

TABLE 3-16
Student PRODUCT *Enrollment*

Student.StdNo	StdLastName	StdMajor	StdClass	OfferNo	Enrollment.StdNo	EnrGrade
123-45-6789	WELLS	IS	FR	1234	123-45-6789	3.3
123-45-6789	WELLS	IS	FR	1234	234-56-7890	3.5
123-45-6789	WELLS	IS	FR	4321	124-56-7890	3.2
124-56-7890	NORBERT	FIN	JR	1234	123-45-6789	3.3
124-56-7890	NORBERT	FIN	JR	1234	234-56-7890	3.5
124-56-7890	NORBERT	FIN	JR	4321	124-56-7890	3.2
234-56-7890	KENDALL	ACCT	JR	1234	123-45-6789	3.3
234-56-7890	KENDALL	ACCT	JR	1234	234-56-7890	3.5
234-56-7890	KENDALL	ACCT	JR	4321	124-56-7890	3.2

TABLE 3-17
Restrict Operation to Retain Rows Matching on *StdNo*

Student.StdNo	StdLastName	StdMajor	StdClass	OfferNo	Enrollment.StdNo	EnrGrade
123-45-6789	WELLS	IS	FR	1234	123-45-6789	3.3
124-56-7890	NORBERT	FIN	JR	4321	124-56-7890	3.2
234-56-7890	KENDALL	ACCT	JR	1234	234-56-7890	3.5

Although the join operator is not primitive, it can be conceptualized directly without its primitive operations. When you are initially learning the join operator, it can be helpful to derive the results using the underlying operations. As an exercise, you are encouraged to derive the result in Figure 3.5. After learning the join operator, you should not need to use the underlying operations.

Visual Formulation of Join Operations As a query formulation aid, many DBMSs provide a visual way to formulate joins. Microsoft Access provides a visual representation of the join operator using the Query Design window. Figure 3.6 depicts a join between *Student* and *Enrollment* on *StdNo* using the Query Design window. To form this join, you need only to select the tables. Access determines that you should join over the *StdNo* column. Access assumes that most joins involve a primary key and foreign key combination. If Access chooses the join condition incorrectly, you can choose other join columns.

3.4.4 Outer Join Operator

The result of a join operation includes the rows matching on the join condition. Sometimes it is useful to include both matching and nonmatching rows. For example, you may want to know offerings that have an assigned instructor as well as offerings without an assigned instructor. In these situations, the outer join operator is useful.

The outer join operator provides the ability to preserve nonmatching rows in the result as well as to include the matching rows. Figure 3.7 depicts an outer join between sample *Faculty* and *Offering* tables. Note that each table has one row that does not match any row in the other table. The third row of *Faculty* and the fourth row of *Offering* do not have matching rows in the other table. For nonmatching rows, null values are used to complete the column values in the other table. In Figure 3.7, blanks (no values) represent null values. The fourth result row is the nonmatched row of *Faculty* with a null value for the *OfferNo* column. Likewise, the fifth result row contains a null value for the first two columns because it is a nonmatched row of *Offering*.

FIGURE 3.6
Query Design Window Showing a Join between Student and Enrollment

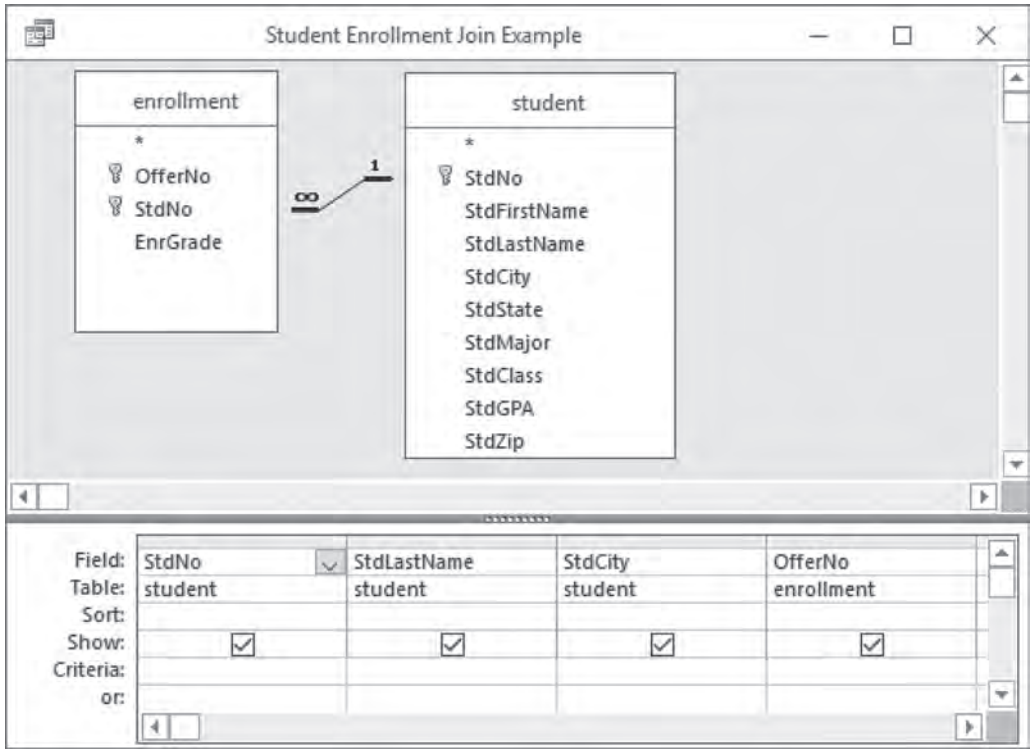
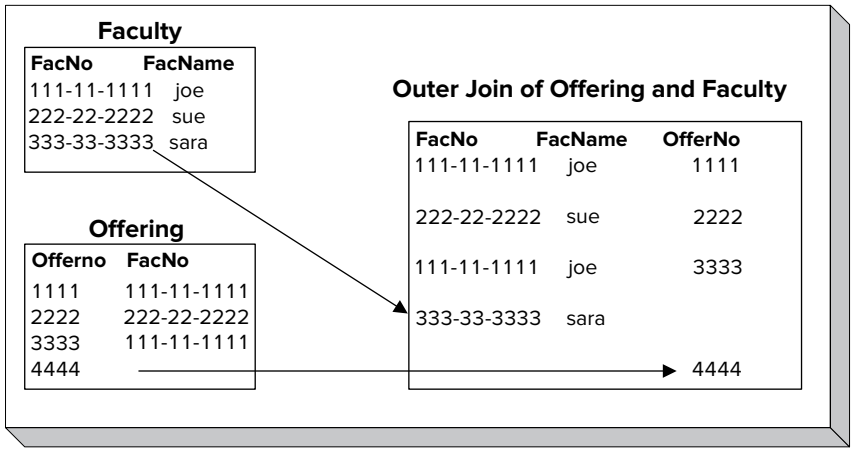


FIGURE 3.7
Sample Outer Join Operation



Full Outer Join
an operator that produces the matching rows (the join part) as well as the nonmatching rows from both input tables.

One-Sided Outer Join
an operator that produces the matching rows (the join part) as well as the nonmatching rows from the designated input table.

Full versus One-Sided Outer Join Operators The outer join operator has two variations. The **full outer join** preserves nonmatching rows from both input tables. Figure 3.7 shows a full outer join because the nonmatching rows from both tables are preserved in the result. Because it is sometimes useful to preserve the nonmatching rows from just one input table, the **one-sided outer join** operator has been devised. In Figure 3.7, only the first four rows of the result would appear for a one-sided outer join that preserves the rows of the *Faculty* table. The last row would not appear in the result because it is an unmatched row of the *Offering* table. Similarly, only the first three rows and the last row would appear in the result for a one-sided outer join that preserves the rows of the *Offering* table.

The outer join is useful in two situations. A full outer join can be used to combine two tables with some common columns and some unique columns. For example, to combine the *Student* and *Faculty* tables on *FacNo* and *StdNo*, a full outer join can be used to show all columns about university people. In Table 3-20, the first two rows are

only from the sample *Student* table (Table 3-18), while the last two rows are only from the sample *Faculty* table (Table 3-19). Note the use of null values for the columns from the other table. The third row in Table 3-20 is the row common to the sample *Faculty* and *Student* tables.

A one-sided outer join can be useful when a table has null values in a foreign key. For example, the *Offering* table (Table 3-21) can have null values in the *FacNo* column representing course offerings without an assigned professor. A one-sided outer join between *Offering* and *Faculty* preserves the rows of *Offering* that do not have an assigned *Faculty* as shown in Table 3-22. With a natural join, the first and third rows of Table 3-22 would not appear. As you will see in Chapter 10, one-sided joins can be useful in data entry forms.

TABLE 3-18
Sample *Student* Table

StdNo	StdLastName	StdMajor	StdClass
123-45-6789	WELLS	IS	FR
124-56-7890	NORBERT	FIN	JR
876-54-3210	COLAN	MS	SR

TABLE 3-19
Sample *Faculty* Table

FacNo	FacLastName	FacDept	FacRank
098-76-5432	VINCE	MS	ASST
543-21-0987	EMMANUEL	MS	PROF
876-54-3210	COLAN	MS	ASST

TABLE 3-20
Result of Full Outer Join of Sample *Student* and *Faculty* Tables on FacNo = StdNo

StdNo	StdLastName	StdMajor	StdClass	FacNo	FacLastName	FacDept	FacRank
123-45-6789	WELLS	IS	FR				
124-56-7890	NORBERT	FIN	JR				
876-54-3210	COLAN	MS	SR	876-54-3210	COLAN	MS	ASST
				098-76-5432	VINCE	MS	ASST
				543-21-0987	EMMANUEL	MS	PROF

TABLE 3-21
Sample *Offering* Table

OfferNo	CourseNo	OffTerm	FacNo
1111	IS320	SUMMER	
1234	IS320	FALL	098-76-5432
2222	IS460	SUMMER	
3333	IS320	SPRING	098-76-5432
4444	IS320	SPRING	543-21-0987

TABLE 3-22
Result of a One-Sided Outer Join between *Offering* (Table 3-21) and *Faculty* (Table 3-19)

OfferNo	CourseNo	OffTerm	Offering.FacNo	Faculty.FacNo	FacLastName	FacDept	FacRank
1111	IS320	SUMMER					
1234	IS320	FALL	098-76-5432	098-76-5432	VINCE	MS	ASST
2222	IS460	SUMMER					
3333	IS320	SPRING	098-76-5432	098-76-5432	VINCE	MS	ASST
4444	IS320	SPRING	543-21-0987	543-21-0987	EMMANUEL	MS	PROF

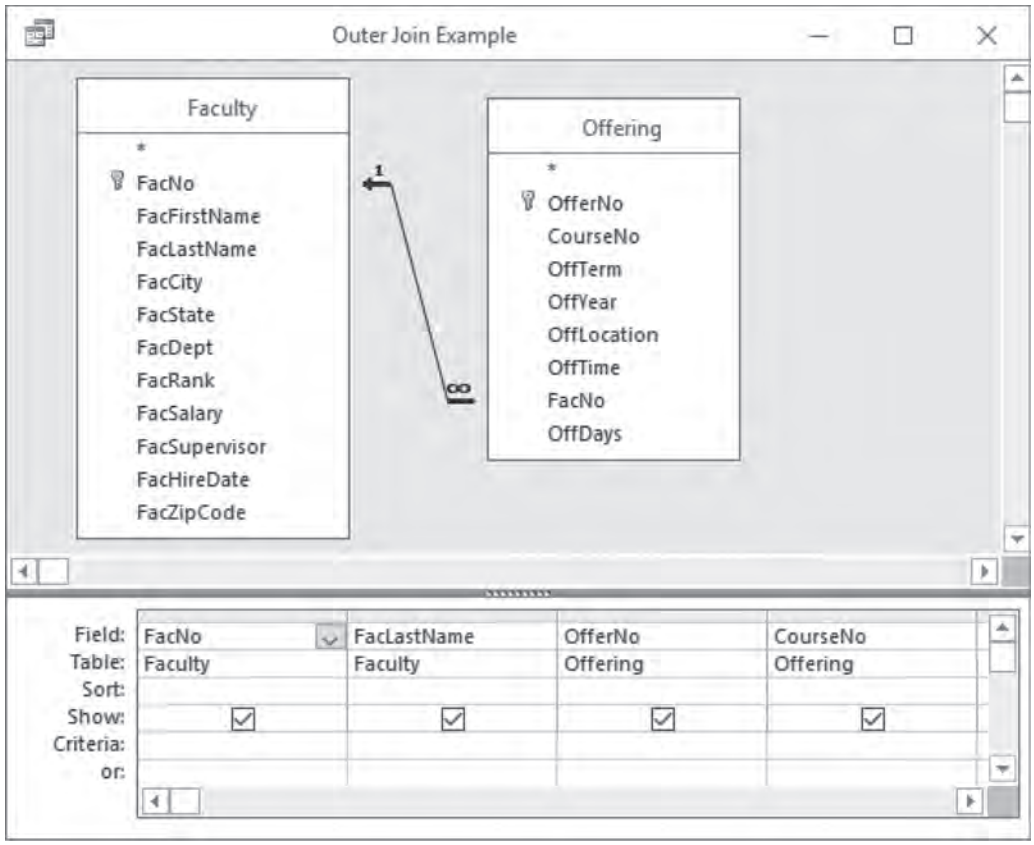


FIGURE 3.8
Query Design Window Showing a One-Sided Outer Join Preserving the *Offering* Table

Visual Formulation of Outer Join Operations As a query formulation aid, many DBMSs provide a visual way to formulate outer joins. Microsoft Access provides a visual representation of the one-sided join operator in the Query Design window. Figure 3.8 depicts a one-sided outer join that preserves the rows of the *Offering*. The arrow from *Offering* to *Faculty* means that the nonmatched rows of *Offering* are preserved in the result. When combining the *Faculty* and *Offering* tables, Microsoft Access provides three choices: (1) show only the matched rows (a join); (2) show matched rows and nonmatched rows of *Faculty*; and (3) show matched rows and nonmatched rows of *Offering*. Choice (3) is shown in Figure 3.8. Choice (1) would appear similar to Figure 3.6. Choice (2) would have the arrow from *Faculty* to *Offering*.

3.4.5 Union, Intersection, and Difference Operators

The union, intersection, and difference table operators are similar to the traditional set operators. The traditional set operators are used to determine all members of two sets (union), common members of two sets (intersection), and members unique to only one set (difference), as depicted in Figure 3.9.

Traditional Set Operators the union operator produces a table containing rows in either input table. The intersection operator produces a table containing rows common to both input tables. The difference operator produces a table containing rows in the first input table but not in the second input table.

The union, intersection, and difference operators for tables apply to rows of a table but otherwise operate in the same way as the **traditional set operators**. A union operation retrieves all the rows in either table. For example, a union operator applied to two student tables at different universities can find all student rows. An intersection operation retrieves just the common rows. For example, an intersection operation can determine the students attending both universities. A difference operation retrieves the rows in the first table but not in the second table. For example, a difference operation can determine the students attending only one university.

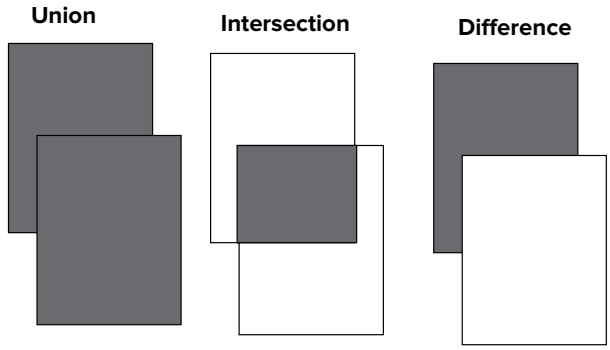


FIGURE 3.9
Venn Diagrams for Traditional Set Operators

Union Compatibility Compatibility is a new concept for the table operators as compared to the traditional set operators. With the table operators, both tables must be union compatible because all columns are compared. **Union compatibility** means that each table must have the same number of columns and each corresponding column must have a compatible data type. Union compatibility can be confusing because it involves positional correspondence of the columns. That is, the first columns of the two tables must have compatible data types, the second columns must have compatible data types, and so on.

Union Compatibility
a requirement on the input tables for the traditional set operators. Each table must have the same number of columns and each corresponding column must have a compatible data type.

To depict the union, intersection, and difference operators, let us apply them to the *Student1* and *Student2* tables (Tables 3-23 and 3-24). These tables are union compatible because they have identical columns listed in the same order. The results of union, intersection, and difference operators are shown in Tables 3-25 through 3-27, respectively. Even though we can determine that two rows are identical from looking only at *StdNo*, all columns are compared due to the design of the operators.

Note that the result of *Student1* DIFFERENCE *Student2* would not be the same as *Student2* DIFFERENCE *Student1*. The result of the latter (*Student2* DIFFERENCE *Student1*) is the second and third rows of *Student2* (rows in *Student2* but not in *Student1*).

TABLE 3-23
Student1 Table

StdNo	StdLastName	StdCity	StdState	StdMajor	StdClass	StdGPA
123-45-6789	WELLS	SEATTLE	WA	IS	FR	3.00
124-56-7890	NORBERT	BOTHELL	WA	FIN	JR	2.70
234-56-7890	KENDALL	TACOMA	WA	ACCT	JR	3.50

TABLE 3-24
Student2 Table

StdNo	StdLastName	StdCity	StdState	StdMajor	StdClass	StdGPA
123-45-6789	WELLS	SEATTLE	WA	IS	FR	3.00
995-56-3490	BAGGINS	AUSTIN	TX	FIN	JR	2.90
111-56-4490	WILLIAMS	SEATTLE	WA	ACCT	JR	3.40

TABLE 3-25
Student1 UNION *Student2*

StdNo	StdLastName	StdCity	StdState	StdMajor	StdClass	StdGPA
123-45-6789	WELLS	SEATTLE	WA	IS	FR	3.00
124-56-7890	NORBERT	BOTHELL	WA	FIN	JR	2.70
234-56-7890	KENDALL	TACOMA	WA	ACCT	JR	3.50
995-56-3490	BAGGINS	AUSTIN	TX	FIN	JR	2.90
111-56-4490	WILLIAMS	SEATTLE	WA	ACCT	JR	3.40

TABLE 3-26
Student1 INTERSECT *Student2*

StdNo	StdLastName	StdCity	StdState	StdMajor	StdClass	StdGPA
123-45-6789	WELLS	SEATTLE	WA	IS	FR	3.00

TABLE 3-27
Student1 DIFFERENCE *Student2*

StdNo	StdLastName	StdCity	StdState	StdMajor	StdClass	StdGPA
124-56-7890	NORBERT	BOTHELL	WA	FIN	JR	2.70
234-56-7890	KENDALL	TACOMA	WA	ACCT	JR	3.50

Because of the union compatibility requirement, the union, intersection, and difference operators are not as widely used as other operators. However, these operators have some important, specialized uses. One use is to combine tables distributed over multiple locations. For example, suppose there are a student table at Big State University (*BSUStudent*) and a student table at University of Big State (*UBSStudent*). Because these tables have identical columns, the traditional set operators are applicable. To find students attending either university, you should use *UBSStudent* UNION *BSUStudent*. To find students only attending Big State, you should use *BSUStudent* DIFFERENCE *UBSStudent*. To find students attending both universities, you should use *UBSStudent* INTERSECT *BSUStudent*. Note that the resulting table in each operation has the same number of columns as the two input tables.

The traditional set operators are also useful if there are tables that are similar but not union compatible. For example, the *Student* and *Faculty* tables have some compatible columns (*StdNo* with *FacNo*, *StdLastName* with *FacLastName*, and *StdCity* with *FacCity*), but other columns are different. The union compatible operators can be used if the *Student* and *Faculty* tables are first made union compatible using the project operator presented in Section 3.4.1.

3.4.6 Summarize Operator

Summarize
an operator that produces a table with groups of rows replaced by row summaries. Aggregate functions are calculated for each summary row in the result table.

Summarize is a powerful operator for decision making. Because tables can contain many rows, it is often useful to see statistics about groups of rows rather than individual rows. The **summarize** operator allows groups of rows to be compressed or summarized by a calculated value. Almost any kind of statistical function can be used to summarize groups of rows. Because this is not a statistics book, we will use only simple functions such as count, min, max, average, and sum.

The summarize operator compresses a table by replacing groups of rows with individual rows containing calculated values. A statistical or aggregate function is used for the calculated values. Figure 3.10 depicts a summarize operation for a sample enrollment table. The summarize operation groups input rows on the *StdNo* column, resulting in three row groups. Then, the summarize operator replaces each group of rows with an individual row containing the *StdNo* value and the average enrollment

FIGURE 3.10
Sample Summarize Operation

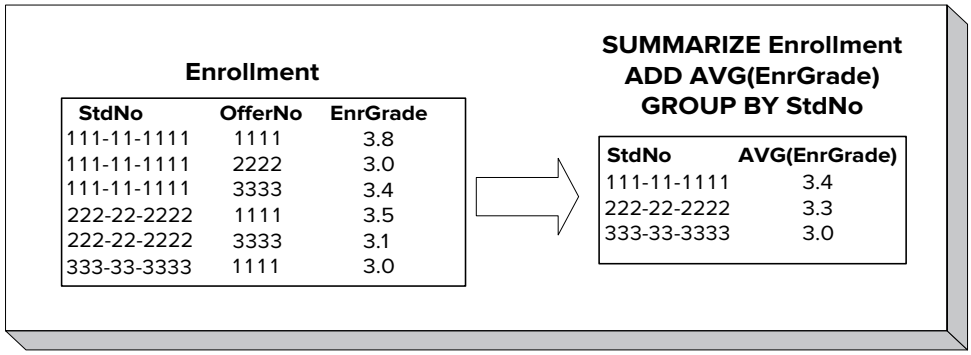


TABLE 3-28
Sample *Faculty* Table

FacNo	FacLastName	FacDept	FacRank	FacSalary	FacSupervisor	FacHireDate
098-76-5432	VINCE	MS	ASST	$35,000	654-32-1098	01-Apr-2004
543-21-0987	EMMANUEL	MS	PROF	$120,000		01-Apr-2005
654-32-1098	FIBON	MS	ASSC	$70,000	543-21-0987	01-Apr-2003
765-43-2109	MACON	FIN	PROF	$65,000		01-Apr-2006
876-54-3210	COLAN	MS	ASST	$40,000	654-32-1098	01-Apr-2008
987-65-4321	MILLS	FIN	ASSC	$75,000	765-43-2109	01-Apr-2009

grade (*EnrGrade*). For example, the summarize operation replaces three rows with *StdNo* value of 111-11-1111 into one row with *StdNo* value 111-11-1111 and the average *EnrGrade* value (3.4) of the three input rows.

As another example, Table 3-29 shows the result of a summarize operation on the sample *Faculty* table in Table 3-28. Note that the result table contains one row per value of the grouping column, *FacDept*.

The summarize operator can include additional calculated values (also showing the minimum salary, for example) and additional grouping columns (also grouping on *FacRank*, for example). When grouping on multiple columns, each result row shows one combination of values for the grouping columns.

TABLE 3-29
Result Table for SUMMARIZE *Faculty* ADD AVG(*FacSalary*) GROUP BY *FacDept*

FacDept	FacSalary
MS	$66,250
FIN	$70,000

3.4.7 Divide Operator

Divide
an operator that produces a table in which the values of a column from one input table are associated with all the values from a column of a second input table.

The **divide** operator is a more specialized and difficult operator than join because the matching requirement in divide is more stringent than join. For example, a join operation retrieves offerings taken by any student. A divide operation retrieves offerings taken by all (or every) students. Because divide has more stringent matching conditions, it is not as widely used as join, and it is more difficult to understand. When appropriate, the divide operator provides a powerful way to combine tables.

The divide operator for tables is somewhat analogous to the divide operator for numbers. In numerical division, the objective is to find the number of times one number contains another number. In table division, the objective is to find values of one column that contain every value in another column. Stated another way, the divide operator finds values of one column that are associated with every value in another column.

To understand the divide operator more concretely, you should consider an example with sample *Part* and *SuppPart* (supplier-part) tables as depicted in Figure 3.11. The divide operator uses two input tables. The first table (*SuppPart*) has two columns (a binary table) and the second table (*Part*) has one column[11] (a unary table). The result table has one column where the values come from the first column of the binary table. The result table in Figure 3.11 shows the suppliers who supply every part. The value s3 appears in the output because it is associated with every value in the *Part* table. Stated another way, the set of values associated with s3 contains the set of values in the *Part* table.

To understand the divide operator in another way, you can rewrite the *SuppPart* table as three rows using the angle brackets <> to surround a row: <s3, {p1, p2, p3}>, <s0, {p1}>, <s1, {p2}>. Rewrite the *Part* table as a set: {p1, p2, p3}. The value s3 is in the result table because its set of second column values {p1, p2, p3} contains the values in the second table {p1, p2, p3}. The other *SuppNo* values (s0 and s1) are not in the result because they are not associated with all values in the *Part* table.

As an example using the university database tables, Table 3-32 shows the result of a divide operation involving the sample *Enrollment* (Table 3-30) and *Student* tables (Table 3-31). The result shows offerings in which every student is enrolled. Only *OfferNo* 4235 has all three students enrolled.

[11] The divide operator can be generalized to work with input tables containing more columns. However, the details are not important in this book.

FIGURE 3.11
Sample Divide Operation

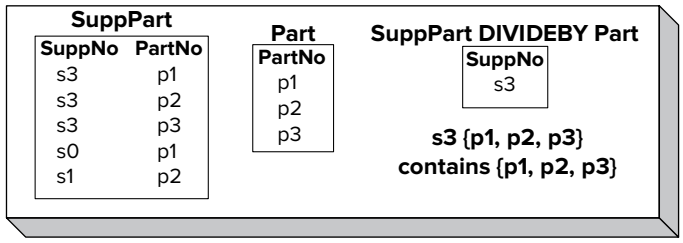


TABLE 3-30
Sample *Enrollment* Table

OfferNo	StdNo
1234	123-45-6789
1234	234-56-7890
4235	123-45-6789
4235	234-56-7890
4235	124-56-7890
6321	124-56-7890

TABLE 3-31
Sample *Student* Table

StdNo
123-45-6789
124-56-7890
234-56-7890

TABLE 3-32
Result of *Enrollment* DIVIDEBY *Student*

OfferNo
4235

3.4.8 Summary of Operators

To help you recall the relational algebra operators, Tables 3-33 and 3-34 provide convenient summaries of the meaning and usage of each operator. You might want to refer to these tables when studying query formulation in later chapters.

TABLE 3-33
Summary of Meanings of the Relational Algebra Operators

Operator	Meaning
Restrict (Select)	Extracts rows that satisfy a specified condition.
Project	Extracts specified columns.
Product	Builds a table from two tables consisting of all possible combinations of rows, one from each of the two tables.
Union	Builds a table consisting of all rows appearing in either of two tables.
Intersect	Builds a table consisting of all rows appearing in both of two tables.
Difference	Builds a table consisting of all rows appearing in the first table but not in the second table.
Join	Extracts rows from a product of two tables such that two input rows contributing to any output row satisfy some specified condition.
Outer Join	Extracts the matching rows (the join part) of two tables and the unmatched rows from one or both tables.
Divide	Builds a table consisting of all values of one column of a binary (two-column) table that match (in the other column) all values in a unary (one-column) table.
Summarize	Organizes a table on specified grouping columns. Specified aggregate computations are made on each value of the grouping columns.

TABLE 3-34
Summary of Usage of the Relational Algebra Operators

Operator	Notes
Union	Input tables must be union compatible.
Difference	Input tables must be union compatible.
Intersection	Input tables must be union compatible.
Product	Conceptually underlies join operator.
Restrict (Select)	Uses a logical expression.
Project	Eliminates duplicate rows if necessary.
Join	Only matched rows are in the result. Natural join eliminates one join column.
Outer Join	Retains both matched and unmatched rows in the result. Uses null values for some columns of the unmatched rows.
Divide	Stronger operator than join, but less frequently used.
Summarize	Specify grouping column(s) if any and aggregate function(s).

CLOSING THOUGHTS

Chapter 3 introduced the Relational Data Model as a prelude to developing queries, forms, and reports with relational databases. As a first step to work with relational databases, you should understand the basic terminology and integrity rules. You should be able to read table definitions in SQL CREATE TABLE statements and visual representations. To effectively query a relational database, you must understand the connections among tables. Most queries involve multiple tables using relationships defined by referential integrity constraints. A graphical representation such as the Relationship window in Microsoft Access provides a powerful tool to conceptualize referential integrity constraints. When developing applications that can change a database, you must respect the rules for referenced rows and understand actions on related rows.

The final part of this chapter described the operators of relational algebra. At this point, you should understand the purpose of each operator, the number of input tables, and other inputs used. You do not need to write complicated formulas that combine operators. Eventually, you should be comfortable understanding statements such as "write an SQL SELECT statement to join three tables." Chapters 4 and 9 present details of the SQL SELECT statement, but you should understand the basic idea of a join now. As you learn to extract data using the SQL SELECT statement in Chapter 4, you may want to review this chapter again. To help you remember the major points about the operators, the last section of this chapter presented several convenient summaries.

Understanding relational algebra operators will improve your knowledge of SQL and your query formulation skills. The meaning of SQL queries can be understood as relational algebra operations. Chapter 4 provides a flowchart demonstrating this correspondence. For this reason, relational algebra provides a yardstick to measure commercial languages: the commercial languages should provide at least the same retrieval ability as the operators of relational algebra.

REVIEW CONCEPTS

- Table with heading (column names) and body (rows)
- Primary keys and entity integrity rule
- Foreign keys, referential integrity rule, and matching values
- Visualizing referential integrity constraints
- Relational Model representation of 1-M relationships, M-N relationships, and self-referencing relationships
- Actions on referenced rows: cascade, nullify, restrict, default
- Subset operators: restrict (select) and project
- Join operator for combining two tables using a matching condition to compare join columns
- Natural join using equality for the matching operator, join columns with the same unqualified name, and elimination of one join column
- Most widely used operator for combining tables: natural join
- Less widely used operators for combining tables: full outer join, one-sided outer join, and divide
- Outer join operator extending the join operator by preserving nonmatching rows
- One-sided outer join preserving the nonmatching rows of one input table
- Full outer join preserving the nonmatching rows of both input tables
- Traditional set operators: union, intersection, difference, extended cross product

- Union compatibility for comparing rows for the union, intersection, and difference operators
- Complex matching operator: divide operator for matching on a subset of rows
- Summarize operator that replaces groups of rows with summary rows

QUESTIONS

1. How is creating a table similar to writing a chapter of a book?
2. With what terminology for relational databases are you most comfortable? Why?
3. What is the difference between a primary key and a candidate key?
4. What is the difference between a candidate key and a superkey?
5. What is a null value?
6. What is the motivation for the entity integrity rule?
7. What is the motivation for the referential integrity rule?
8. What is the relationship between the referential integrity rule and foreign keys?
9. How are candidate keys that are not primary keys indicated in the CREATE TABLE statement?
10. What is the advantage of using constraint names when defining primary key, candidate key, and referential integrity constraints in CREATE TABLE statements?
11. When is it not permissible for foreign keys to store null values?
12. What is the purpose of a database diagram such as the Access Relationship window?
13. How is a 1-M relationship represented in the Relational Model?
14. How is an M-N relationship represented in the Relational Model?
15. What is a self-referencing relationship?
16. How is a self-referencing relationship represented in the Relational Model?
17. What is a referenced row?
18. What two actions on referenced rows can affect related rows in a child table?
19. What are the possible actions on related rows after a referenced row is deleted or its primary key is updated?
20. Why is the restrict action for referenced rows more common than the cascade action?
21. When is the nullify action not allowed?
22. Why study the operators of relational algebra?
23. Why are the restrict and the project operators widely used?
24. Explain how the union, intersection, and difference operators for tables differ from the traditional operators for sets.
25. Why is the join operator so important for retrieving useful information?
26. What is the relationship between the join operator and the extended cross product operator?
27. Why is the extended cross product operator used sparingly?
28. What happens to unmatched rows with the join operator?
29. What happens to unmatched rows with the full outer join operator?
30. What is the difference between the full outer join and the one-sided outer join?

31. Define a decision-making situation that might require the summarize operator.
32. What is an aggregate function?
33. How are grouping columns used in the summarize operator?
34. Why is the divide operator not as widely used as the join operator?
35. What are the requirements of union compatibility?
36. What are the requirements of the natural join operator?
37. Why is the natural join operator widely used for combining tables?
38. How do visual tools such as the Microsoft Access Query Design tool facilitate the formulation of join operations?
39. Why are non-minimal superkeys typically ignored?
40. What are two interpretations for null values?
41. What is an important specialized use of the traditional set operators (union, intersection, and difference)?
42. Provide examples of each traditional set operator for the specialized situation that you provided in your answer to problem 41.
43. If two tables have some common and some unique columns, what operator can be used to make the tables union compatible?

PROBLEMS

The problems use the *Customer*, *OrderTbl*, and *Employee* tables of the simplified Order Entry database. Chapters 4 and 10 extend the database to increase its usefulness. The *Customer* table contains clients who have placed orders. The *OrderTbl* contains basic facts about customer orders. The *Employee* table contains facts about employees who take orders. The primary keys of the tables are *CustNo* for *Customer*, *EmpNo* for *Employee*, and *OrdNo* for *OrderTbl*.

Customer

CustNo	CustFirstName	CustLastName	CustCity	CustState	CustZip	CustBal
C0954327	Sheri	Gordon	Littleton	CO	80129-5543	$230.00
C1010398	Jim	Glussman	Denver	CO	80111-0033	$200.00
C2388597	Beth	Taylor	Seattle	WA	98103-1121	$500.00
C3340959	Betty	Wise	Seattle	WA	98178-3311	$200.00
C3499503	Bob	Mann	Monroe	WA	98013-1095	$0.00
C8543321	Ron	Thompson	Renton	WA	98666-1289	$85.00

Employee

EmpNo	EmpFirstName	EmpLastName	EmpPhone	EmpEmail
E1329594	Landi	Santos	(303) 789-1234	LSantos@bigco.com
E8544399	Joe	Jenkins	(303) 221-9875	JJenkins@bigco.com
E8843211	Amy	Tang	(303) 556-4321	ATang@bigco.com
E9345771	Colin	White	(303) 221-4453	CWhite@bigco.com
E9884325	Thomas	Johnson	(303) 556-9987	TJohnson@bigco.com
E9954302	Mary	Hill	(303) 556-9871	MHill@bigco.com

OrderTbl

OrdNo	OrdDate	CustNo	EmpNo
O1116324	01/23/2017	C0954327	E8544399
O2334661	01/14/2017	C0954327	E1329594
O3331222	01/13/2017	C1010398	
O2233457	01/12/2017	C2388597	E9884325
O4714645	01/11/2017	C2388597	E1329594
O5511365	01/22/2017	C3340959	E9884325
O7989497	01/16/2017	C3499503	E9345771
O1656777	02/11/2017	C8543321	
O7959898	02/19/2017	C8543321	E8544399

1. Write a CREATE TABLE statement for the *Customer* table. Choose data types appropriate for the DBMS used in your course. Note that the *CustBal* column contains numeric data. The currency symbols are not stored in the database. The *CustFirstName* and *CustLastName* columns are required (not null).
2. Write a CREATE TABLE statement for the *Employee* table. Choose data types appropriate for the DBMS used in your course. The *EmpFirstName*, *EmpLastName*, and *EmpEMail* columns are required (not null).
3. Write a CREATE TABLE statement for the *OrderTbl* table. Choose data types appropriate for the DBMS used in your course. The *OrdDate* column is required (not null).
4. Identify the foreign keys and draw a relationship diagram for the simplified Order Entry database. The *CustNo* column references the *Customer* table and the *EmpNo* column references the *Employee* table. For each relationship, identify the parent table and the child table.
5. Extend your CREATE TABLE statement from problem (3) with referential integrity constraints. Updates and deletes on related rows are restricted.
6. From examination of the sample data and your common understanding of order entry businesses, are null values allowed for the foreign keys in the *OrderTbl* table? Why or why not? Extend the CREATE TABLE statement in problem (5) to enforce the null value restrictions if any.
7. Extend your CREATE TABLE statement for the *Employee* table (problem 2) with a unique constraint for *EmpEMail*. Use a named constraint clause for the unique constraint.
8. Show the result of a restrict operation that lists the orders in February 2017.
9. Show the result of a restrict operation that lists the customers residing in Seattle, WA.
10. Show the result of a project operation that lists the *CustNo*, *CustFirstName*, and *CustLastName* columns of the *Customer* table.
11. Show the result of a project operation that lists the *CustCity* and *CustState* columns of the *Customer* table.
12. Show the result of a natural join that combines the *Customer* and *OrderTbl* tables.
13. Show the steps to derive the natural join for problem (10). How many rows and columns are in the extended cross product step?
14. Show the result of a natural join of the *Employee* and *OrderTbl* tables.
15. Show the result of a one-sided outer join between the *Employee* and *OrderTbl* tables. Preserve the rows of the *OrderTbl* table in the result.
16. Show the result of a full outer join between the *Employee* and *OrderTbl* tables.

17. Show the result of the restrict operation on *Customer* where the condition is *CustCity* equals "Denver" or "Seattle" followed by a project operation to retain the *CustNo*, *CustFirstName*, *CustLastName*, and *CustCity* columns.
18. Show the result of a natural join that combines the *Customer* and *OrderTbl* tables followed by a restrict operation to retain only the Colorado customers (CustState = "CO").
19. Show the result of a summarize operation on *Customer*. The grouping column is *CustState* and the aggregate calculation is COUNT. COUNT shows the number of rows with the same value for the grouping column.
20. Show the result of a summarize operation on *Customer*. The grouping column is *CustState* and the aggregate calculations are the minimum and maximum *CustBal* values.
21. What tables are required to show the *CustLastName*, *EmpLastName*, and *OrdNo* columns in the result table?
22. Extend your relationship diagram from problem (4) by adding two tables (*OrdLine* and *Product*). Partial CREATE TABLE statements for the primary keys and referential integrity constraints are shown below:

```
CREATE TABLE Product … PRIMARY KEY (ProdNo)
CREATE TABLE OrdLine … PRIMARY KEY (OrdNo, ProdNo)
  FOREIGN KEY (OrdNo) REFERENCES Order
  FOREIGN KEY (ProdNo) REFERENCES Product
```

23. Extend your relationship diagram from problem (22) by adding a foreign key in the *Employee* table. The foreign key *SupEmpNo* is the employee number of the supervising employee. Thus, the *SupEmpNo* column references the *Employee* table.
24. What relational algebra operator do you use to find products contained in <u>every</u> order? What relational algebra operator do you use to find products contained in <u>any</u> order?
25. Are the *Customer* and *Employee* tables union compatible? Why or why not?
26. Using the database after problem (23), what tables must be combined to list the product names on order number O1116324?
27. Using the database after problem (23), what tables must be combined to list the product names ordered by customer number C0954327?
28. Using the database after problem (23), what tables must be combined to list the product names ordered by the customer named Sheri Gordon?
29. Using the database after problem (23), what tables must be combined to list the number of orders submitted by customers residing in Colorado?
30. Using the database after problem (23), what tables must be combined to list the product names appearing on an order taken by an employee named Landi Santos?
31. If two tables such as *Customer* and *Employee* are not union compatible, what operations would you use before performing a union operation?
32. With the *Employee* table extended with the foreign key *SupEmpNo* as specified in problem 23, what tables must be combined to list the supervisor name of the employee who took a specified order?
33. In problem 22, what type of relationship does the *OrdLine* table represent?
34. In problem 22, can the foreign keys in the *OrdLine* table accept null values? Explain your answer.

REFERENCES FOR FURTHER STUDY

Codd defined the Relational Model in a seminal paper in 1970. His paper inspired research projects at the IBM research laboratories and the University of California at Berkeley that led to commercial relational DBMSs. Date (2003) provides a syntax for the relational algebra. Elmasri and Navathe (2017) provide a more theoretical treatment of the Relational Model, especially the relational algebra.

chapter

4

Query Formulation with SQL

Learning Objectives

This chapter provides the foundation for developing your query formulation skills using the industry standard Structured Query Language (SQL). Query formulation involves conversion of a request for data into a statement of a database language such as SQL. After this chapter, the student should have acquired the following knowledge and skills:

- Write SQL SELECT statements for queries involving the restrict, project, and join operators
- Understand the meaning of the WHERE and GROUP BY clauses using the conceptual evaluation process
- Use the critical questions to transform a problem statement into a database representation
- Write SELECT statements for more difficult queries involving joins of three or more tables, self joins, joins with grouping, and multiple joins between tables
- Write brief descriptions to document SQL SELECT statements
- Write INSERT, UPDATE, and DELETE statements to change the rows of a table

OVERVIEW

Chapter 3 provided a foundation for using relational databases. Most importantly, you learned about connections among tables and fundamental operators to extract useful data. This chapter helps you to apply this knowledge in using the SQL SELECT statement.

Much of your skill with SQL will derive from imitating examples. This chapter provides many examples to facilitate your learning process. Initially you will see relatively simple examples so that you become comfortable with the basics of the SQL SELECT statement. To prepare for more difficult examples, this chapter presents two problem-solving guidelines (conceptual evaluation process and critical questions). The conceptual evaluation process explains the meaning of the SELECT statement through the sequence of operations and intermediate tables that produce a result table. The critical questions help you transform a problem statement into a relational database representation in a language such as SQL. These guidelines should help you to formulate and understand advanced problems presented in Section 4.5. The last part of this chapter presents negative examples with formulation errors and poor coding practices to help you avoid errors and poor coding practices.

4.1 BACKGROUND

Before using SQL, you should understand its history and scope. The history reveals the origin of the name and the efforts to standardize the language. The scope puts the various parts of SQL into perspective. You have already seen the CREATE TABLE statement in Chapter 3. This chapter presents basics of the SELECT, UPDATE, DELETE, and INSERT statements, while Chapter 9 provides more details about complex query formulation problems and associated SELECT statement details. To broaden your understanding, you should be aware of other parts of SQL and different usage contexts.

4.1.1 Brief History of SQL

The Structured Query Language (SQL) has a colorful history. Table 4-1 depicts the highlights of SQL's development. SQL began life as the SQUARE language in IBM's System R project. The System R project was a response to the interest in relational databases sparked by Dr. Ted Codd, an IBM fellow who wrote a famous paper in 1970 about relational databases. The SQUARE language was somewhat mathematical in nature. After conducting human factors experiments, the IBM research team revised the language and renamed it SEQUEL (a follow-up to SQUARE). After another revision, the language was dubbed SEQUEL 2. Its current name, SQL, resulted from legal issues surrounding the name SEQUEL. Because of this naming history, a number of database professionals, particularly those working during the 1970s, pronounce the name as "sequel" rather than SQL.

SQL is now an international standard[1] although it was not always so. With the force of IBM behind SQL, many imitators used some variant of SQL. Such was the old order of the computer industry when IBM was dominant. It may seem surprising, but IBM was not the first company to commercialize SQL. Until a standards effort developed in the 1980s, SQL was in a state of confusion. Many vendors implemented different subsets of SQL with unique extensions. The standards efforts by the American National Standards Institute (ANSI), the International Organization for Standards (ISO), and the International Electrotechnical Commission (IEC) have restored some

TABLE 4-1
SQL Timeline

Year	Event
1972	System R project at IBM Research Labs
1974	SQUARE language developed
1975	Language revision and name change to SEQUEL
1976	Language revision and name change to SEQUEL 2
1977	Name change to SQL
1978	First commercial implementation by Oracle Corporation
1981	IBM product SQL/DS featuring SQL
1986	SQL-86 (SQL1) standard approved
1989	SQL-89 standard approved (revision to SQL-86)
1992	SQL-92 (SQL2) standard approved
1999	SQL:1999 (SQL3) standard approved
2003	SQL:2003 approved
2008	SQL:2008 approved
2011	SQL:2011 approved
2016	SQL:2016 approved

[1] Dr. Michael Stonebraker, an early database pioneer, has even referred to SQL as "intergalactic data speak."

order. Although SQL was not initially the best database language developed, the standards efforts have improved the language as well as standardized its specification.

The size and scope of the SQL standard has increased substantially since adoption of the first standard. The original standard (SQL-86) contained about 150 pages, while the SQL-92 standard contained more than 600 pages with another 500 pages added after the initial SQL-92 standard was published. The standard grew to about 2,000 pages for SQL:1999 and about 4,000 pages for SQL:2016. The SQL:2016 standard contains 14 parts although four parts (5 to 8 and 12) were never officially released and only three parts of the standard (2, 11, and 14) have become widely implemented. Part 2 on the foundation specifies most of SQL with extensions in other parts.

The SQL standard lacks conformance testing, an important weakness. Until 1996, the U.S. Department of Commerce's National Institute of Standards and Technology conducted conformance tests to provide assurance about portability for government software among conforming DBMSs. Since 1996, however, DBMS vendor claims have substituted for independent conformance testing. Even for basic parts of the SQL foundation, the major vendors lack support for some features and provide proprietary support for other features. With the optional parts, conformance has much greater variance. Writing portable SQL code requires careful study for basic parts of the foundation but is not possible for extended parts of SQL.

4.1.2 Scope of SQL

SQL was designed as a language for database definition, manipulation, and control. Table 4-2 shows a quick summary of important SQL statements. Only database administrators use most of the database definition and control statements. You have already seen the CREATE TABLE statement in Chapter 3. This chapter and Chapter 9 cover the database manipulation statements (SELECT, UPDATE, INSERT, and DELETE). Power users and analysts use the database manipulation statements. Chapter 10 covers the CREATE VIEW statement. Either database administrators or analysts can use the CREATE VIEW statement. Chapter 11 covers the CREATE TRIGGER statement used by both database administrators and analysts. Chapter 15 covers extensions to the SELECT statement and the CREATE MATERIALIZED VIEW statements, both important in data warehouse usage. Chapter 16 covers the GRANT, REVOKE, and CREATE ASSERTION statements, used primarily by database administrators. Chapter 17 presents processing details about the transaction control statements (COMMIT and ROLLBACK), important conceptual background for database administrators. Chapter 19 covers extensions of the SELECT statement for object databases.

SQL supports two usage contexts, stand-alone and embedded. In the stand-alone context, a user submits SQL statements with the use of a specialized editor. The editor alerts the user to syntax errors and sends the statements to the DBMS for execution. The presentation in this chapter assumes stand-alone usage. In the embedded context, an executing program submits SQL statements, and the DBMS sends results back

TABLE 4-2
Selected SQL Statements

Statement Type	Statements	Purpose
Database definition	CREATE SCHEMA, TABLE, VIEW	Define a new database, table, and view
	ALTER TABLE	Modify table definition
Database manipulation	SELECT	Retrieve contents of tables
	UPDATE, DELETE, INSERT	Modify, remove, and add rows
Database control	COMMIT, ROLLBACK	Complete, undo transaction
	GRANT, REVOKE	Add and remove access rights
	CREATE ASSERTION	Define integrity constraint
	CREATE TRIGGER	Define database rule

to the program. The program includes SQL statements along with statements of the host programming language such as Java or Visual Basic. Additional statements allow SQL statements (such as SELECT) to be used inside a computer program. Chapter 11 covers embedded SQL with the Oracle database programming language, PL/SQL.

SQL Usage Contexts: The SQL standard supports two usage environments, standalone with statements submitted using a specialized editor, and embedded with statements inside of a computer program.

4.2 GETTING STARTED WITH THE SELECT STATEMENT

The SELECT statement supports data retrieval from one or more tables. This chapter describes fundamental query formulation problems and a basic syntax of the SELECT statement. Chapter 9 presents more complex query formulation problems and extended syntax for the SELECT statement. The SELECT statement described here has the following format:

```
SELECT <list of columns and expressions usually involving columns>
 FROM <list of tables and join operations>
 WHERE  <row conditions connected by AND, OR, NOT>
 GROUP BY <list of grouping columns>
 HAVING <group conditions connected by AND, OR, NOT>
 ORDER BY <list of sorting specifications>
```

In the preceding format, uppercase words are keywords. You replace the angle brackets <> with details to make a meaningful statement. For example, after the keyword `SELECT`, you specify the list of columns in the result, but do not type the angle brackets. The result list contains columns such as *StdFirstName* or expressions involving constants, column names, and functions. Example expressions are `Price * Qty` and `1.1 * FacSalary`. To make meaningful names for computed columns, you can rename a column in the result table using the AS keyword. For example, `SELECT Price * Qty AS Amount` renames the expression `Price * Qty` to `Amount` in the result table.

Expression: a combination of constants, column names, functions, and operators that generates a value when executed. In conditions and result columns, expressions can be used in any place that column names can appear.

To depict this SELECT statement format and show the meaning of statements, this chapter shows numerous examples. Examples are provided for Microsoft Access, a popular desktop DBMS, and Oracle, a prominent enterprise DBMS. Most examples execute on both DBMSs. Unless noted, the examples execute on the 1997 through 2016 versions of Access and the 8i through 12c versions of Oracle. Examples that only execute on one product are marked. In addition to the examples, Appendix 4.B summarizes syntax differences among major DBMSs.

The examples use the university database tables introduced in Chapter 3. Tables 4-3 through 4-7 list the contents of the tables. Appendix 4.A contains CREATE TABLE statements for the tables. For your reference, Figure 4.1 repeats (from Chapter 3) the relationship diagram with primary and foreign keys. Recall that the *Faculty_1* table with relationship to the *Faculty* table represents a self-referencing relationship with *FacSupervisor* as the foreign key.

StdNo	StdFirstName	StdLastName	StdCity	StdState	StdZip	StdMajor	StdClass	StdGPA
123-45-6789	HOMER	WELLS	SEATTLE	WA	98121-1111	IS	FR	3.00
124-56-7890	BOB	NORBERT	BOTHELL	WA	98011-2121	FIN	JR	2.70
234-56-7890	CANDY	KENDALL	TACOMA	WA	99042-3321	ACCT	JR	3.50
345-67-8901	WALLY	KENDALL	SEATTLE	WA	98123-1141	IS	SR	2.80
456-78-9012	JOE	ESTRADA	SEATTLE	WA	98121-2333	FIN	SR	3.20
567-89-0123	MARIAH	DODGE	SEATTLE	WA	98114-0021	IS	JR	3.60
678-90-1234	TESS	DODGE	REDMOND	WA	98116-2344	ACCT	SO	3.30
789-01-2345	ROBERTO	MORALES	SEATTLE	WA	98121-2212	FIN	JR	2.50
876-54-3210	CRISTOPHER	COLAN	SEATTLE	WA	98114-1332	IS	SR	4.00
890-12-3456	LUKE	BRAZZI	SEATTLE	WA	98116-0021	IS	SR	2.20
901-23-4567	WILLIAM	PILGRIM	BOTHELL	WA	98113-1885	IS	SO	3.80

TABLE 4-3
Sample *Student* Table

FacNo	FacFirstName	FacLastName	FacCity	FacState	FacDept	FacRank	FacSalary
098-76-5432	LEONARD	VINCE	SEATTLE	WA	MS	ASST	$35,000
543-21-0987	VICTORIA	EMMANUEL	BOTHELL	WA	MS	PROF	$120,000
654-32-1098	LEONARD	FIBON	SEATTLE	WA	MS	ASSC	$70,000
765-43-2109	NICKI	MACON	BELLEVUE	WA	FIN	PROF	$65,000
876-54-3210	CRISTOPHER	COLAN	SEATTLE	WA	MS	ASST	$40,000
987-65-4321	JULIA	MILLS	SEATTLE	WA	FIN	ASSC	$75,000

TABLE 4-4A
Sample *Faculty* Table
(first part)

FacNo	FacSupervisor	FacHireDate	FacZipCode
098-76-5432	654-32-1098	10-Apr-2004	98111-9921
543-21-0987		15-Apr-2005	98011-2242
654-32-1098	543-21-0987	01-May-2003	98121-0094
765-43-2109		11-Apr-2006	98015-9945
876-54-3210	654-32-1098	01-Mar-2008	98114-1332
987-65-4321	765-43-2109	15-Mar-2009	98114-9954

TABLE 4-4B
Sample *Faculty* Table
(second part)

OfferNo	CourseNo	OffTerm	OffYear	OffLocation	OffTime	FacNo	OffDays
1111	IS320	SUMMER	2017	BLM302	10:30 AM		MW
1234	IS320	FALL	2016	BLM302	10:30 AM	098-76-5432	MW
2222	IS460	SUMMER	2016	BLM412	1:30 PM		TTH
3333	IS320	SPRING	2017	BLM214	8:30 AM	098-76-5432	MW
4321	IS320	FALL	2016	BLM214	3:30 PM	098-76-5432	TTH
4444	IS320	WINTER	2017	BLM302	3:30 PM	543-21-0987	TTH
5555	FIN300	WINTER	2017	BLM207	8:30 AM	765-43-2109	MW
5678	IS480	WINTER	2017	BLM302	10:30 AM	987-65-4321	MW
5679	IS480	SPRING	2017	BLM412	3:30 PM	876-54-3210	TTH
6666	FIN450	WINTER	2017	BLM212	10:30 AM	987-65-4321	TTH
7777	FIN480	SPRING	2017	BLM305	1:30 PM	765-43-2109	MW
8888	IS320	SUMMER	2017	BLM405	1:30 PM	654-32-1098	MW
9876	IS460	SPRING	2017	BLM307	1:30 PM	654-32-1098	TTH

TABLE 4-5
Sample *Offering* Table

TABLE 4-6
Sample *Course* Table

CourseNo	CrsDesc	CrsUnits
FIN300	FUNDAMENTALS OF FINANCE	4
FIN450	PRINCIPLES OF INVESTMENTS	4
FIN480	CORPORATE FINANCE	4
IS320	FUNDAMENTALS OF BUSINESS PROGRAMMING	4
IS460	SYSTEMS ANALYSIS	4
IS470	BUSINESS DATA COMMUNICATIONS	4
IS480	FUNDAMENTALS OF DATABASE MANAGEMENT	4

TABLE 4-7
Sample *Enrollment* Table

OfferNo	StdNo	EnrGrade
1234	123-45-6789	3.3
1234	234-56-7890	3.5
1234	345-67-8901	3.2
1234	456-78-9012	3.1
1234	567-89-0123	3.8
1234	678-90-1234	3.4
4321	123-45-6789	3.5
4321	124-56-7890	3.2
4321	789-01-2345	3.5
4321	876-54-3210	3.1
4321	890-12-3456	3.4
4321	901-23-4567	3.1
5555	123-45-6789	3.2
5555	124-56-7890	2.7
5678	123-45-6789	3.2
5678	234-56-7890	2.8
5678	345-67-8901	3.3
5678	456-78-9012	3.4
5678	567-89-0123	2.6
5679	123-45-6789	2
5679	124-56-7890	3.7
5679	678-90-1234	3.3
5679	789-01-2345	3.8
5679	890-12-3456	2.9
5679	901-23-4567	3.1
6666	234-56-7890	3.1
6666	567-89-0123	3.6
7777	876-54-3210	3.4
7777	890-12-3456	3.7
7777	901-23-4567	3.4
9876	124-56-7890	3.5
9876	234-56-7890	3.2
9876	345-67-8901	3.2
9876	456-78-9012	3.4
9876	567-89-0123	2.6
9876	678-90-1234	3.3
9876	901-23-4567	4

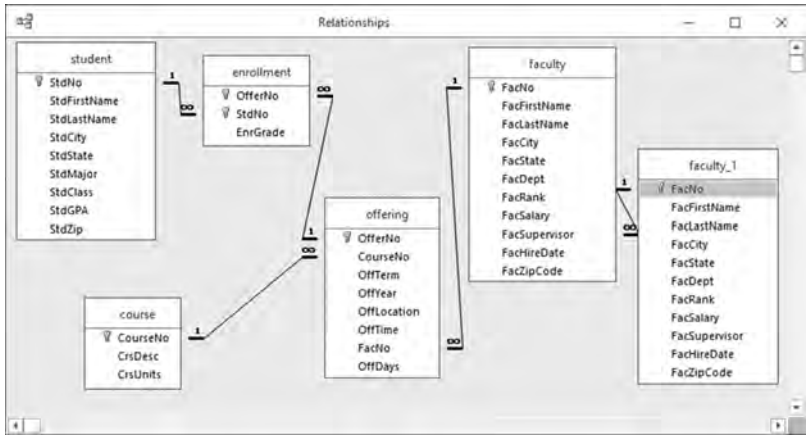


FIGURE 4.1
Relationship Window for the University Database

4.2.1 Single Table Problems

This section shows a variety of examples involving single tables. After some simple initial examples, more complex examples involving mathematical expressions, inexact matching, date comparisons, null values, and logical expressions are shown.

Let us begin with the simple SELECT statement in Example 4.1. In all examples, keywords appear in uppercase while information specific to the query appears in mixed case. In Example 4.1, only the *Student* table is listed in the FROM clause because the conditions in the WHERE clause and columns after the SELECT keyword involve only the *Student* table. In Oracle, a semicolon or / (on a separate line) terminates a statement.

Example 4.1

Testing Rows Using the WHERE Clause

Retrieve the name, city, and grade point average (GPA) of students with a high GPA (greater than or equal to 3.7). The result follows the SELECT statement.

```
SELECT StdFirstName, StdLastName, StdCity, StdGPA
  FROM Student
  WHERE StdGPA >= 3.7
```

StdFirstName	StdLastName	StdCity	StdGPA
CRISTOPHER	COLAN	SEATTLE	4.00
WILLIAM	PILGRIM	BOTHELL	3.80

Table 4-8 depicts the standard comparison operators. These symbols are used by all major DBMSs supporting SQL.

TABLE 4-8
Standard Comparison Operators

Comparison Operator	Meaning
=	equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

Example 4.2 is even simpler than Example 4.1. The result is identical to the original *Faculty* table in Table 4-4. Example 4.2 uses a shortcut to list all columns. The asterisk * in the column list indicates that all columns of the tables in the FROM clause appear in the result. The asterisk serves as a wildcard character matching all column names.

Example 4.2

Show all Columns

List all columns and rows of the *Faculty* table. The resulting table is shown in two parts.

```
SELECT * FROM Faculty
```

FacNo	FacFirstName	FacLastName	FacCity	FacState	FacDept	FacRank	FacSalary
098-76-5432	LEONARD	VINCE	SEATTLE	WA	MS	ASST	$35,000
543-21-0987	VICTORIA	EMMANUEL	BOTHELL	WA	MS	PROF	$120,000
654-32-1098	LEONARD	FIBON	SEATTLE	WA	MS	ASSC	$70,000
765-43-2109	NICKI	MACON	BELLEVUE	WA	FIN	PROF	$65,000
876-54-3210	CRISTOPHER	COLAN	SEATTLE	WA	MS	ASST	$40,000
987-65-4321	JULIA	MILLS	SEATTLE	WA	FIN	ASSC	$75,000

FacNo	FacSupervisor	FacHireDate	FacZipCode
098-76-5432	654-32-1098	10-Apr-2004	98111-9921
543-21-0987		15-Apr-2005	98011-2242
654-32-1098	543-21-0987	01-May-2003	98121-0094
765-43-2109		11-Apr-2006	98015-9945
876-54-3210	654-32-1098	01-Mar-2008	98114-1332
987-65-4321	765-43-2109	15-Mar-2009	98114-9954

Expression Examples Example 4.3 depicts expressions in the SELECT and WHERE clauses. The expression in the SELECT clause increases the salary by 10 percent. The AS keyword is used to rename the computed column. Without renaming, most DBMSs will generate a meaningless name such as Expr001. The expression in the WHERE clause extracts the year from the hiring date. Since Access and Oracle differ on functions for date columns, Example 4.3 provides separate statements. To become proficient with SQL on a particular DBMS, you will need to study the available functions especially with date columns.

Example 4.3 (Access)

Expressions in SELECT and WHERE Clauses

List the name, city, and increased salary of faculty hired after 2005. The **year** function extracts the year part of a column with a date data type.

```
SELECT FacFirstName, FacLastName, FacCity,
       FacSalary*1.1 AS IncreasedSalary, FacHireDate
 FROM  Faculty
 WHERE year(FacHireDate) > 2005
```

FacFirstName	FacLastName	FacCity	IncreasedSalary	FacHireDate
NICKI	MACON	BELLEVUE	71500	11-Apr-2006
CRISTOPHER	COLAN	SEATTLE	44000	01-Mar-2008
JULIA	MILLS	SEATTLE	82500	15-Mar-2009

Example 4.3 (Oracle)

Expressions in SELECT and WHERE Clauses

The **to_char** function extracts the four-digit year from the *FacHireDate* column and the **to_number** function converts the character representation of the year into a number.

```
SELECT FacFirstName, FacLastName, FacCity,
       FacSalary*1.1 AS IncreasedSalary, FacHireDate
 FROM Faculty
 WHERE to_number(to_char(FacHireDate, 'YYYY' ) ) > 2005
```

Example 4.4 uses a more complex expression in the WHERE clause to retrieve faculty hired in the last 10 years. In Access, the **Now()**function retrieves the current date, while in Oracle, the **SysDate** function retrieves the current date. Both formulations are rather imprecise, just using the year component of the date. More precise formulation using all date components involves proprietary functions for date differences.

Example 4.4 (Access)

Using a function to retrieve today's date

List the name, city, and hire date of faculty hired in the last 10 years (2017 was the current year when the statement was executed). The **Now()** function retrieves the current date.

```
SELECT FacFirstName, FacLastName, FacCity, FacHireDate
 FROM  Faculty
 WHERE year(FacHireDate) >= year(Now()) - 10
```

FacFirstName	FacLastName	FacCity	FacHireDate
CRISTOPHER	COLAN	SEATTLE	01-Mar-2008
JULIA	MILLS	SEATTLE	15-Mar-2009

Example 4.4 (Oracle)

Using a function to retrieve today's date

List the name, city, and hire date of faculty hired in the last 10 years. The **SYSDATE** function retrieves the current date.

```
SELECT FacFirstName, FacLastName, FacCity, FacHireDate
 FROM  Faculty
 WHERE to_number(to_char(FacHireDate, 'YYYY' ) ) >=
       to_number(to_char(SYSDATE, 'YYYY' ) ) - 10
```

Example 4.5 uses an expression in the WHERE clause to retrieve students near an A- (3.7) GPA. The formulations in Access and Oracle are identical.

Example 4.5

Using an expression in the WHERE clause

List the name, city, and GPA of students near an A- (3.7) GPA. The first condition eliminates students above the A- threshold. The second condition eliminates students far below an A-.

```
SELECT StdFirstName, StdLastName, StdCity, StdGPA
 FROM  Student
 WHERE StdGPA < 3.7 AND StdGPA * 1.1 >= 3.7
```

StdFirstName	StdLastName	StdCity	StdGPA
CANDY	KENDALL	TACOMA	3.50
MARIAH	DODGE	SEATTLE	3.60

Examples with Exact and Inexact Matching on String Columns Columns using a character string data type (CHAR or VARCHAR) support both exact and inexact matching. You can use the equality = comparison operator for exact matching with a string column. In Example 4.6, the condition, `CourseNo = 'IS480'`, matches a single row in the *Course* table.

Example 4.6

Exact Matching on a String Column with the = Operator

List all columns of the course row with IS480 as the course number.

```
SELECT *
 FROM Course
 WHERE CourseNo = 'IS480'
```

CourseNo	CrsDesc	CrsUnits
IS480	FUNDAMENTALS OF DATABASE MANAGEMENT	4

For conditions on string columns, case sensitivity is an important issue. Some DBMSs such as Microsoft Access are not case sensitive. In Access SQL, the condition in Example 4.6 matches "is480", "Is480", and "iS480" in addition to "IS480". Other DBMSs such as Oracle are case sensitive. In Oracle SQL, the condition in Example 4.6 matches only "IS480", not "is480", "Is480", or "iS480". To alleviate confusion, you can use the Oracle **upper** (see Example 4.7) or **lower** functions to convert strings to upper or lowercase, respectively.

Example 4.7 (Oracle)

Exact Matching using the upper Function

List all columns of the course row with IS480 as the course number.

```
SELECT *
 FROM Course
 WHERE upper(CourseNo) = 'IS480'
```

CourseNo	CrsDesc	CrsUnits
IS480	FUNDAMENTALS OF DATABASE MANAGEMENT	4

Inexact matching supports conditions that match some pattern rather than matching an identical string. One of the most common types of inexact matching is to find values having a common prefix such as "IS4" (400 level information systems courses). Example 4.8 uses the LIKE operator along with the pattern-matching character * to perform prefix matching[2]. The string constant `'IS4*'` means match strings beginning with "IS4" and ending with anything. The wildcard character * matches any string. The Oracle formulation of Example 4.8 uses the percent symbol %, the SQL standard for the wildcard character. Note that string constants must be enclosed in quotation marks[3].

Example 4.8 (Access)

Inexact Matching with the LIKE Operator

List the senior-level IS courses.

```
SELECT *
 FROM Course
 WHERE CourseNo LIKE 'IS4*'
```

CourseNo	CrsDesc	CrsUnits
IS460	SYSTEMS ANALYSIS	4
IS470	BUSINESS DATA COMMUNICATIONS	4
IS470	FUNDAMENTALS OF DATABASE MANAGEMENT	4

Example 4.8 (Oracle)

Inexact Matching with the LIKE Operator

List the senior-level IS courses.

```
SELECT *
 FROM Course
 WHERE CourseNo LIKE 'IS4%'
```

[2] Beginning with Access 2002, the standard SQL pattern-matching characters can be used by specifying ANSI 92 query mode in the Options window. The textbook uses the default * and ? pattern-matching characters for Access SQL statements.

[3] Most DBMSs require single quotes, the SQL:2016 standard. Microsoft Access allows either single or double quotes for string constants.

Another common type of inexact matching is to find columns containing a substring. To perform this kind of matching, a wildcard character should be used before and after the substring. For example, to find courses containing the word DATABASE anywhere in the course description, write the condition: `CrsDesc LIKE '*DATA*'` in Access or `CrsDesc LIKE '%DATA%'` in Oracle as shown in Example 4.9.

Example 4.9 (Access)

Inexact Matching for a Substring

List the courses containing the string "DATA" in the course description.

```
SELECT *
 FROM Course
 WHERE CrsDesc LIKE '*DATA*'
```

CourseNo	CrsDesc	CrsUnits
IS470	BUSINESS DATA COMMUNICATIONS	4
IS480	FUNDAMENTALS OF DATABASE MANAGEMENT	4

Example 4.9 (Oracle)

Inexact Matching for a Substring

List the courses containing the string "DATA" in the course description.

```
SELECT *
 FROM Course
 WHERE CrsDesc LIKE '%DATA%'
```

The wildcard character is not the only pattern-matching character. SQL:2016 specifies the underscore character _ to match any single character. Some DBMSs such as Access use the question mark (?) to match any single character. Figure 4.10 shows examples of single character matching in both Access and Oracle. Most DBMSs also have pattern-matching characters for matching a range of characters (for example, the digits 0 to 9) and any character from a list of characters. The symbols used for these other pattern-matching characters are not standard. To become proficient at writing inexact matching conditions, you should study the pattern-matching characters available with your DBMS.

Example 4.10 (Access)

Inexact Matching for a Single Character

List the name and rank of faculty with a five letter last name ending in "N". Each question mark matches any single character.

```
SELECT FacFirstName, FacLastName, FacRank
 FROM Faculty
 WHERE FacLastName LIKE '????N'
```

FacFirstName	FacLastName	FacRank
LEONARD	FIBON	ASSC
NICKI	MACON	PROF
CRISTOPHER	COLAN	ASST

Example 4.10 (Oracle)

Inexact Matching for a Single Character

List the name and rank of faculty with a five letter last name ending in "N". Each underscore matches any single character.

```
SELECT FacFirstName, FacLastName, FacRank
 FROM Faculty
 WHERE FacLastName LIKE '____N'
```

Date Comparison Examples Example 4.11 depicts range matching on a column with the date data type. In Access SQL, pound symbols enclose date constants, while in Oracle SQL, single quotation marks enclose date constants as shown in Example 4.11. Date columns can be compared just like numbers with the usual comparison operators (=, <, etc.). The BETWEEN-AND operator defines a closed interval (includes end points). In Access Example 4.11, the BETWEEN-AND condition is a shortcut for `FacHireDate >= #1/1/2008# AND FacHireDate <= #12/31/2009#`.

BETWEEN-AND Operator: a shortcut operator to test a numeric or date column against a range of values. The BETWEEN-AND operator returns true if the column is greater than or equal to the first value and less than or equal to the second value.

Example 4.11 (Access)

Comparing a Date Column to Date Constants

List the name and hiring date of faculty hired in 2008 or 2009.

```
SELECT FacFirstName, FacLastName, FacHireDate
 FROM  Faculty
 WHERE FacHireDate BETWEEN #1/1/2008# AND #12/31/2009#
```

FacFirstName	FacLastName	FacHireDate
CRISTOPHER	COLAN	01-Mar-2008
JULIA	MILLS	15-Mar-2009

Example 4.11 (Oracle)

Comparing a Date Column to Date Constants

List the name and hiring date of faculty hired in 2008 or 2009. In Oracle SQL, the standard format for dates is DD-Mon-YYYY where DD is the day number, Mon is the month abbreviation, and YYYY is the four-digit year.

```
SELECT FacFirstName, FacLastName, FacHireDate
 FROM Faculty
 WHERE FacHireDate BETWEEN '1-Jan-2008' AND '31-Dec-2009'
```

You should not use the LIKE operator and pattern matching characters on date columns. Some DBMSs allow the LIKE operator, but it is not portable across DBMSs and the results may vary within a DBMS. You should treat date columns as numeric, not text. You should use standard comparison operators along with proprietary functions to compare dates and components of dates. Example 4.12 shows another example using proprietary functions to extract the month part of a date column.

Example 4.12 (Access)

Using a Proprietary Function to Retrieve Month Number

List the name and hiring date of faculty hired in April of any year. The Access **Month** function retrieves the month number part of a date column.

```
SELECT FacFirstName, FacLastName, FacHireDate
 FROM  Faculty
 WHERE Month(FacHireDate) = 4
```

FacFirstName	FacLastName	FacHireDate
LEONARD	VINCE	10-Apr-2005
VICTORIA	EMMANUEL	15-Apr-2005
NICKI	MACON	11-Apr-2006

Example 4.12 (Oracle)

Using Proprietary Functions to Retrieve Month Number

List the name and hiring date of faculty hired in April of any year. The format string "MM" in the Oracle **to_char** function retrieves the month number part of a date column.

```
SELECT FacFirstName, FacLastName, FacHireDate
 FROM  Faculty
 WHERE to_number(to_char(FacHireDate, 'MM' ) ) = 4
```

Examples with Null Values and Logical Expressions Besides testing columns for specified values, you sometimes need to test for the lack of a value. Null values are used when there is no normal value for a column. A null can mean that the value is unknown or the value is not applicable to the row. For the *Offering* table, a null value for *FacNo* means that the instructor is unknown at the current time. Testing for null values is done with the IS NULL comparison operator as shown in Example 4.13. You can also test for a normal value using IS NOT NULL.

Example 4.13

Testing for Nulls

List the offering number and course number of summer 2017 offerings without an assigned instructor.

```
SELECT OfferNo, CourseNo
 FROM Offering
 WHERE FacNo IS NULL AND OffTerm = 'SUMMER'
   AND OffYear = 2017
```

OfferNo	CourseNo
1111	IS320

Example 4.14 depicts a complex logical expression involving both logical operators AND and OR. When **mixing AND and OR** in a logical expression, you should use parentheses. Otherwise, the reader of the SELECT statement may not understand the grouping of the AND and OR conditions. Without parentheses, you must depend on the default precedence (AND evaluated before OR). The reader of the statement may not know the default precedence.

Mixing AND and OR
always use parentheses to make the grouping of conditions explicit.

Example 4.14

Complex Logical Expression

List the offer number, course number, and faculty number for course offerings scheduled in fall 2016 or winter 2017.

```
SELECT OfferNo, CourseNo, FacNo
 FROM Offering
 WHERE (OffTerm = 'FALL' AND OffYear = 2016)
    OR (OffTerm = 'WINTER' AND OffYear = 2017)
```

OfferNo	CourseNo	FacNo
1234	IS320	098-76-5432
4321	IS320	098-76-5432
4444	IS320	543-21-0987
5555	FIN300	765-43-2109
5678	IS480	987-65-4321
6666	FIN450	987-65-4321

4.2.2 Joining Tables

Example 4.15 demonstrates a join of the *Course* and *Offering* tables. The join condition `Course.CourseNo = Offering.CourseNo` is specified in the WHERE clause.

Example 4.15 (Access)

Join Tables but Show Columns from One Table Only

List the offering number, course number, days, and time of offerings containing the words *database* or *programming* in the course description and taught in spring 2017. The Oracle version of this example uses the % instead of the * as the wildcard character.

```
SELECT OfferNo, Offering.CourseNo, OffDays, OffTime
 FROM Offering, Course
 WHERE OffTerm = 'SPRING' AND OffYear = 2017
   AND (CrsDesc LIKE '*DATABASE*'
    OR  CrsDesc LIKE '*PROGRAMMING*')
   AND Course.CourseNo = Offering.CourseNo
```

OfferNo	CourseNo	OffDays	OffTime
3333	IS320	MW	8:30 AM
5679	IS480	TTH	3:30 PM

You should note two additional points about Example 4.15. First, the *CourseNo* column name must be qualified (prefixed) with a table name (*Course* or *Offering*). Otherwise, the SELECT statement is ambiguous because *CourseNo* can refer to a column in either the *Course* or *Offering* tables. Second, both tables must be listed in the FROM clause even though the result columns come from only the *Offering* table. The *Course* table is needed in the FROM clause because conditions in the WHERE clause reference *CrsDesc*, a column of the *Course* table.

Example 4.16 demonstrates another join, but this time the result columns come from both tables. There are conditions on each table in addition to the join conditions. The Oracle formulation uses the % instead of the * as the wildcard character.

Example 4.16 (Access)

Join Tables and Show Columns from Both Tables

List the offer number, course number, and name of the instructor of IS course offerings scheduled in fall 2016 taught by assistant professors.

```
SELECT OfferNo, CourseNo, FacFirstName, FacLastName
 FROM Offering, Faculty
 WHERE OffTerm = 'FALL' AND OffYear = 2016
   AND FacRank = 'ASST' AND CourseNo LIKE 'IS*'
   AND Faculty.FacNo = Offering.FacNo
```

OfferNo	CourseNo	FacFirstName	FacLastName
1234	IS320	LEONARD	VINCE
4321	IS320	LEONARD	VINCE

Example 4.16 (Oracle)

Join Tables and Show Columns from Both Tables

List the offer number, course number, and name of the instructor of IS course offerings scheduled in fall 2016 taught by assistant professors.

```
SELECT OfferNo, CourseNo, FacFirstName, FacLastName
 FROM Offering, Faculty
 WHERE OffTerm = 'FALL' AND OffYear = 2016
   AND FacRank = 'ASST' AND CourseNo LIKE 'IS%'
   AND Faculty.FacNo = Offering.FacNo
```

In the SQL standard, a join operation can be expressed directly in the FROM clause rather than being expressed in both the FROM and WHERE clauses as shown in Examples 4.15 and 4.16. To make a join operation in the FROM clause, use the keywords INNER JOIN as shown in Example 4.17. The join conditions are indicated by the ON

keyword inside the FROM clause. Notice that the join condition no longer appears in the WHERE clause.

Example 4.17 (Access)

Join Tables Using a Join Operation in the FROM Clause

List the offer number, course number, and name of the instructor of IS course offerings scheduled in fall 2016 that are taught by assistant professors (result is identical to Example 4.16). In Oracle, you should use the % instead of *.

```
SELECT OfferNo, CourseNo, FacFirstName, FacLastName
 FROM Offering INNER JOIN Faculty
   ON Faculty.FacNo = Offering.FacNo
 WHERE OffTerm = 'FALL' AND OffYear = 2016
   AND FacRank = 'ASST' AND CourseNo LIKE 'IS*'
```

4.2.3 Summarizing Tables with GROUP BY and HAVING

So far, the results of all examples in this section relate to individual rows. Even Example 4.9 relates to a combination of columns from individual *Offering* and *Faculty* rows. As indicated in Chapter 3 with the Summarize operator, it is sometimes important to show summaries of rows. The GROUP BY and HAVING clauses are used to show results about groups of rows rather than individual rows.

Example 4.18 depicts the GROUP BY clause to summarize groups of rows. Each result row contains a value of the grouping column (*StdMajor*) along with the aggregate calculation summarizing rows with the same value for the grouping column. The GROUP BY clause must contain every column in the SELECT clause except for aggregate expressions. For example, adding the *StdClass* column in the SELECT clause would make Example 4.18 invalid unless *StdClass* was also added to the GROUP BY clause.

Example 4.18

Grouping on a Single Column

Summarize the averageGPA of students by major.

```
SELECT StdMajor, AVG(StdGPA) AS AvgGPA
 FROM Student
 GROUP BY StdMajor
```

StdMajor	AvgGPA
ACCT	3.39999997615814
FIN	2.80000003178914
IS	3.23333330949148

GROUP BY Reminder: the columns in the SELECT clause must either be in the GROUP BY clause or be part of a summary calculation with an aggregate function.

Table 4-9 shows the standard aggregate functions. If you have a statistical calculation that cannot be performed with these functions, check your DBMS. Most DBMSs feature many functions beyond these standard ones.

TABLE 4-9
Standard Aggregate Functions

Aggregate Function	Meaning and Comments
COUNT(*)	Computes the number of rows.
COUNT(column)	Counts the non-null values in column; DISTINCT can be used to count unique column values.
AVG	Computes the average of a numeric column or expression excluding null values; DISTINCT can be used to compute the average of unique column values.
SUM	Computes the sum of a numeric column or expression excluding null values; DISTINCT can be used to compute the sum of unique column values.
MIN	Computes the smallest value. For string columns, the collating sequence is used to compare strings.
MAX	Computes the largest value. For string columns, the collating sequence is used to compare strings.

The COUNT, AVG, and SUM functions support the DISTINCT keyword to restrict the computation to unique column values. Example 4.19 demonstrates the DISTINCT keyword for the COUNT function. This example retrieves the number of offerings in a year as well as the number of distinct courses taught. Some DBMSs such as Microsoft Access do not support the DISTINCT keyword inside of aggregate functions. Chapter 9 presents an alternative formulation in Access SQL to compensate for the inability to use the DISTINCT keyword inside the COUNT function.

COUNT Function Usage: COUNT(*) and COUNT(column) produce identical results except when "column" contains null values. See Chapter 9 for more details about the effect of null values on aggregate functions.

Example 4.19 (Oracle)

Counting Rows and Unique Column Values

Summarize the number of offerings and unique courses by year.

```
SELECT OffYear, COUNT(*) AS NumOfferings,
       COUNT(DISTINCT CourseNo) AS NumCourses
 FROM Offering
 GROUP BY OffYear
```

OffYear	NumOfferings	NumCourses
2016	3	2
2017	10	6

Examples 4.20 and 4.21 contrast the WHERE and HAVING clauses. In Example 4.20, the WHERE clause selects upper-division students (juniors or seniors) before grouping on major. Because the WHERE clause eliminates students before grouping occurs, only upper-division students are grouped. In Example 4.21, a HAVING condition retains groups with an average GPA greater than 3.1. The HAVING clause applies to groups of rows, whereas the WHERE clause applies to individual rows. To use a HAVING clause, there must be a GROUP BY clause.

WHERE vs. HAVING: use the WHERE clause for conditions that can be tested on individual rows. Use the HAVING clause for conditions that can be tested only on groups. Conditions in the HAVING clause should involve aggregate functions, whereas conditions in the WHERE clause cannot involve aggregate functions.

Example 4.20

Grouping with Row Conditions

Summarize the average GPA of upper division (junior or senior) students by major.

```
SELECT StdMajor, AVG(StdGPA) AS AvgGpa
 FROM Student
 WHERE StdClass = 'JR' OR StdClass = 'SR'
 GROUP BY StdMajor
```

StdMajor	AvgGPA
ACCT	3.5
FIN	2.80000003178914
IS	3.14999997615814

Example 4.21

Grouping with Row and Group Conditions

Summarize the average GPA of upper-division (junior or senior) students by major. Only list the majors with average GPA greater than 3.1.

```
SELECT StdMajor, AVG(StdGPA) AS AvgGpa
 FROM Student
 WHERE StdClass IN ('JR', 'SR')
 GROUP BY StdMajor
 HAVING AVG(StdGPA) > 3.1
```

StdMajor	AvgGPA
ACCT	3.5
IS	3.14999997615814

HAVING Reminder: the HAVING clause must be preceded by the GROUP BY clause.

One other point about Examples 4.20 and 4.21 is the use of the OR operator as compared to the IN operator (set element of operator). The WHERE condition in Examples 4.20 and 4.21 retains the same rows. The IN condition is true if *StdClass* matches any value in the parenthesized list. Chapter 9 provides additional explanation about the IN operator for nested queries.

To summarize all rows, aggregate functions can be used in SELECT clause without a GROUP BY clause as demonstrated in Example 4.22. The result contains a single row with just the aggregate calculations for all rows in the result before computing the aggregate functions.

Example 4.22

Grouping all Rows

List the number of upper-division students and their average GPA.

```
SELECT COUNT(*) AS StdCnt, AVG(StdGPA) AS AvgGPA
 FROM Student
 WHERE StdClass IN ('JR','SR')
```

StdCnt	AvgGPA
8	3.0625

Sometimes it is useful to group on more than one column as demonstrated by Example 4.23. The result shows one row for each combination of *StdMajor* and *StdClass*. Some rows have the same value for both aggregate calculations because there is only one associated row in the *Student* table. For example, there is only one row for the combination ('ACCT', 'JR').

Example 4.23

Grouping on Two Columns

Summarize the minimum and maximum GPA of students by major and class.

```
SELECT StdMajor, StdClass, MIN(StdGPA) AS MinGPA,
       MAX(StdGPA) AS MaxGPA
 FROM Student
 GROUP BY StdMajor, StdClass
```

StdMajor	StdClass	MinGPA	MaxGPA
ACCT	JR	3.5	3.5
ACCT	SO	3.3	3.3
FIN	JR	2.5	2.7
FIN	SR	3.2	3.2
IS	FR	3	3.0
IS	JR	3.6	3.6
IS	SO	3.8	3.8
IS	SR	2.2	4.0

A powerful combination is to use grouping with joins. There is no reason to restrict grouping to just one table. Often, more useful information is obtained by summarizing rows that result from a join. Example 4.24 demonstrates grouping applied to a join between *Course* and *Offering*. You should note that the join is performed before the

grouping occurs. For example, after the join, the result contains six rows with a course description of FUNDAMENTALS OF BUSINESS PROGRAMMING. Because queries combining joins and grouping can be difficult to understand, Section 4.3 provides a more detailed explanation.

Example 4.24 (Access)

Combining Grouping and Joins

Summarize the number of IS course offerings by course description.

```
SELECT CrsDesc, COUNT(*) AS OfferCount
 FROM Course, Offering
 WHERE Course.CourseNo = Offering.CourseNo
   AND Course.CourseNo LIKE 'IS*'
 GROUP BY CrsDesc
```

CrsDesc	OfferCount
FUNDAMENTALS OF BUSINESS PROGRAMMING	6
FUNDAMENTALS OF DATABASE MANAGEMENT	2
SYSTEMS ANALYSIS	2

Example 4.24 (Oracle)

Combining Grouping and Joins

Summarize the number of IS course offerings by course description.

```
SELECT CrsDesc, COUNT(*) AS OfferCount
 FROM Course, Offering
 WHERE Course.CourseNo = Offering.CourseNo
   AND Course.CourseNo LIKE 'IS%'
 GROUP BY CrsDesc
```

4.2.4 Improving the Appearance of Results

We finish this section with two parts of the SELECT statement that can improve the appearance of results. Examples 4.25 and 4.26 demonstrate sorting using the ORDER BY clause. The sort sequence depends on the date type of the sorted column (numeric for numeric data types, ASCII collating sequence for string columns, and calendar sequence for date columns). By default, sorting occurs in ascending order. The keyword DESC can be used after a column name to sort in descending order as demonstrated in Example 4.26.

Example 4.25

Sorting on a Single Column

List the GPA, name, city, and state of juniors. Order the result by GPA in ascending order.

```
SELECT StdGPA, StdFirstName, StdLastName, StdCity,
       StdState
 FROM Student
 WHERE StdClass = 'JR'
 ORDER BY StdGPA
```

StdGPA	StdFirstName	StdLastName	StdCity	StdState
2.50	ROBERTO	MORALES	SEATTLE	WA
2.70	BOB	NORBERT	BOTHELL	WA
3.50	CANDY	KENDALL	TACOMA	WA
3.60	MARIAH	DODGE	SEATTLE	WA

Example 4.26

Sorting on Two Columns with Descending Order

List the rank, salary, name, and department of faculty. Order the result by ascending (alphabetic) rank and descending salary.

```
SELECT FacRank, FacSalary, FacFirstName, FacLastName,
       FacDept
 FROM Faculty
 ORDER BY FacRank, FacSalary DESC
```

FacRank	FacSalary	FacFirstName	FacLastName	FacDept
ASSC	75000.00	JULIA	MILLS	FIN
ASSC	70000.00	LEONARD	FIBON	MS
ASST	40000.00	CRISTOPHER	COLAN	MS
ASST	35000.00	LEONARD	VINCE	MS
PROF	120000.00	VICTORIA	EMMANUEL	MS
PROF	65000.00	NICKI	MACON	FIN

Some students confuse ORDER BY and GROUP BY. In most DBMSs, GROUP BY has the side effect of sorting by the grouping columns. You should not depend on this side effect. If you just want to sort, use ORDER BY rather than GROUP BY. If you want to sort and group, use both ORDER BY and GROUP BY.

Another way to improve the appearance of the result is to remove duplicate rows. By default, SQL does not remove duplicate rows. Duplicate rows are not possible when the primary keys of the result tables are included. There are a number of situations in which the primary key does not appear in the result. Example 4.28 demonstrates the DISTINCT keyword to remove duplicates that appear in the result of Example 4.27.

ORDER BY vs. DISTINCT: use the ORDER BY clause to sort a result table on one or more columns. Use the DISTINCT keyword to remove duplicates in the result.

Example 4.27

Result with Duplicates

List the city and state of faculty members.

```
SELECT FacCity, FacState
  FROM Faculty
```

FacCity	FacState
SEATTLE	WA
BOTHELL	WA
SEATTLE	WA
BELLEVUE	WA
SEATTLE	WA
SEATTLE	WA

Example 4.28

Eliminating Duplicates with DISTINCT

List the unique city and state combinations in the *Faculty* table.

```
SELECT DISTINCT FacCity, FacState
 FROM Faculty
```

FacCity	FacState
BELLEVUE	WA
BOTHELL	WA
SEATTLE	WA

4.3 CONCEPTUAL EVALUATION PROCESS FOR SELECT STATEMENTS

To clarify the meaning of the SELECT statement, you should understand the conceptual evaluation process or sequence of steps to produce the desired result. The **conceptual evaluation process** describes operations (mostly relational algebra operations) that produce intermediate tables leading to a result table. You may find it useful to refer to the conceptual evaluation process when first learning to write SELECT statements. After you gain initial competence with the SELECT statement, you should not need to refer to the conceptual evaluation process except to gain insight about difficult problems.

Conceptual Evaluation Process
the sequence of operations and intermediate tables used to derive the result of a SELECT statement. The conceptual evaluation process may help you gain an initial understanding of the SELECT statement as well as help you to understand more difficult problems.

To demonstrate the conceptual evaluation process, Example 4.29 uses many parts of the SELECT statement. It involves multiple tables (*Enrollment* and *Offering* in the FROM clause), row conditions in the WHERE clause, aggregate functions (COUNT and AVG) over groups of rows (GROUP BY), a group condition in the HAVING clause, and sorting of the final result (ORDER BY).

In the ORDER BY clause, you should note number 3 as the second column to sort. The number 3 means sort by the third column (*AvgGrade*) in the SELECT clause. Some DBMSs do not allow aggregate expressions or alias names (*AvgGrade*) in the ORDER BY clause.

Tables 4-10 to 4-12 show the input tables and the result. Only small input and result tables have been used so that you can understand more clearly the process to derive the result. Small tables can depict the conceptual evaluation process well.

The conceptual evaluation process involves a sequence of operations as indicated in Figure 4.2. This process is conceptual rather than actual because most SQL compilers can produce the same output using many shortcuts. Because the shortcuts are DBMS specific, rather mathematical, and performance oriented, we will not review them. The conceptual evaluation process provides a foundation for understanding the meaning

of SQL statements, independent of SQL compiler and performance issues. The remainder of this section applies the conceptual evaluation process to Example 4.29.

1) The first step in the conceptual evaluation process combines the tables in the FROM clause with the cross product and join operators. Example 4.29 uses a cross product operation because two tables are listed. A join operation is not used because the INNER JOIN keyword does not appear in the FROM statement. Recall that the cross product operator shows all possible rows by combining two tables. The resulting table contains the product of the number of rows and the sum of the columns. In this case, the cross product contains 35 rows (5×7) and 7 columns (3+4). Table 4-13 shows a partial result. As an exercise, you are encouraged to derive the entire result. As a notational shortcut here, the table name (abbreviated as *E* and *O*) is prefixed before the column name for *OfferNo*.

Example 4.29 (Access)

Depict Many Parts of the SELECT Statement

List the course number, offer number, and average grade of students enrolled in fall 2016, IS course offerings in which more than one student is enrolled. Sort the result by course number in ascending order and average grade in descending order. The Oracle version of Example 4.29 is identical except for the % instead of the * as the wildcard character.

```
SELECT CourseNo, Offering.OfferNo,
       AVG(EnrGrade) AS AvgGrade
 FROM Enrollment, Offering
 WHERE CourseNo LIKE 'IS*' AND OffYear = 2016
   AND OffTerm = 'FALL'
   AND Enrollment.OfferNo = Offering.OfferNo
 GROUP BY CourseNo, Offering.OfferNo
 HAVING  COUNT(*) > 1
 ORDER BY CourseNo, 3 DESC
```

TABLE 4-10
Sample *Offering* Table

OfferNo	CourseNo	OffYear	OffTerm
1111	IS480	2016	FALL
2222	IS480	2016	FALL
3333	IS320	2016	FALL
5555	IS480	2016	WINTER
6666	IS320	2016	SPRING

TABLE 4-11
Sample *Enrollment* Table

StdNo	OfferNo	EnrGrade
111-11-1111	1111	3.1
111-11-1111	2222	3.5
111-11-1111	3333	3.3
111-11-1111	5555	3.8
222-22-2222	1111	3.2
222-22-2222	2222	3.3
333-33-3333	1111	3.6

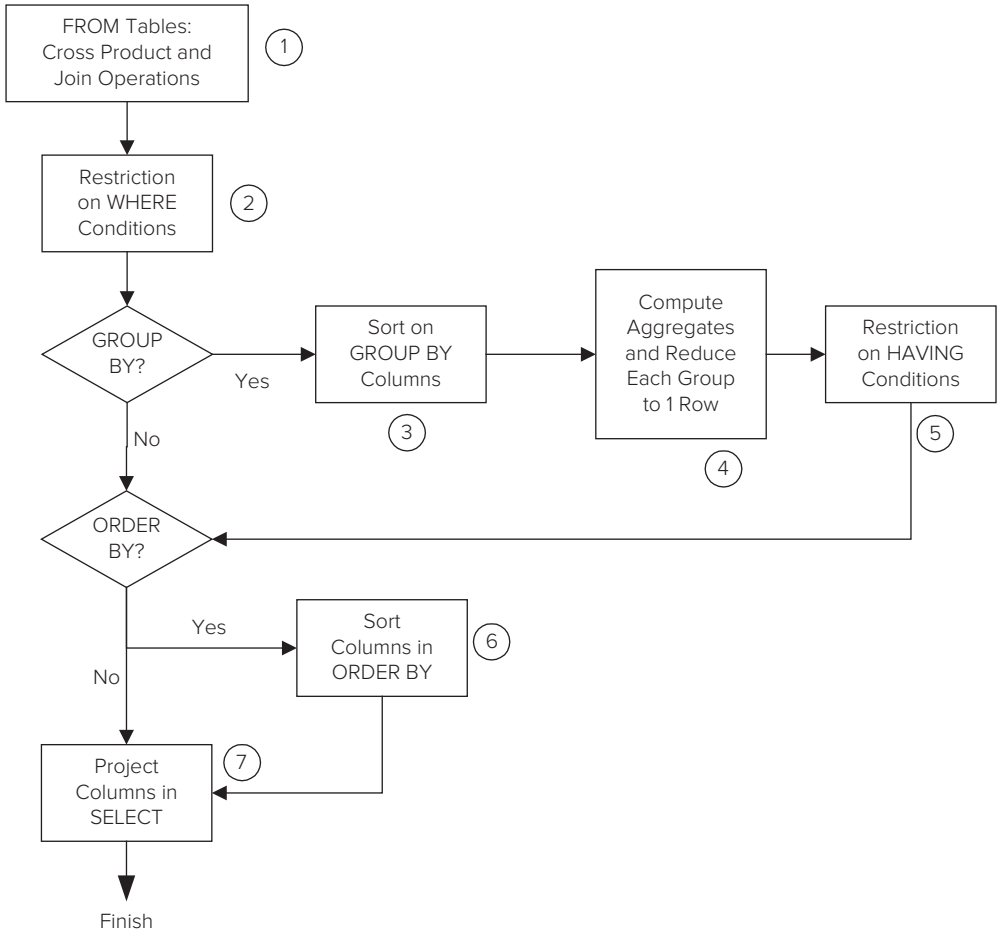


FIGURE 4.2
Flowchart of the Conceptual Evaluation Process

CourseNo	OfferNo	AvgGrade
IS480	2222	3.4
IS480	1111	3.3

TABLE 4-12
Example 4.22 Result

O.OfferNo	CourseNo	OffYear	OffTerm	StdNo	E.OfferNo	EnrGrade
1111	IS480	2016	FALL	111-11-1111	1111	3.1
1111	IS480	2016	FALL	111-11-1111	2222	3.5
1111	IS480	2016	FALL	111-11-1111	3333	3.3
1111	IS480	2016	FALL	111-11-1111	5555	3.8
1111	IS480	2016	FALL	222-22-2222	1111	3.2
1111	IS480	2016	FALL	222-22-2222	2222	3.3
1111	IS480	2016	FALL	333-33-3333	1111	3.6
2222	IS480	2016	FALL	111-11-1111	1111	3.1
2222	IS480	2016	FALL	111-11-1111	2222	3.5
2222	IS480	2016	FALL	111-11-1111	3333	3.3
2222	IS480	2016	FALL	111-11-1111	5555	3.8
2222	IS480	2016	FALL	222-22-2222	1111	3.2
2222	IS480	2016	FALL	222-22-2222	2222	3.3
2222	IS480	2016	FALL	333-33-3333	1111	3.6

TABLE 4-13
Partial Result of Step 1 for First Two *Offering* Rows (1111 and 2222)

2) The second step uses a restriction operation to eliminate rows that do not satisfy the conditions in the WHERE clause. The statement contains four conditions: a join condition on *OfferNo*, a condition on *CourseNo*, a condition on *OffYear*, and a condition on *OffTerm*. Note that the condition on *CourseNo* includes the wildcard character (*). Course numbers beginning with "IS" match this condition. Table 4-14 reduces the result to six rows from the 35 rows in step 1 with the cross product operation.

3) The third step sorts the result of step 2 by the columns specified in the GROUP BY clause. The GROUP BY clause indicates that the output should relate to groups of rows rather than individual rows. If the output relates to individual rows rather than groups of rows, the GROUP BY clause is omitted. When using the GROUP BY clause, you must include every column from the SELECT clause except for expressions that involve an aggregate function[4]. Table 4-15 shows the result of step 2 sorted by *CourseNo* and *O.OfferNo*. Note that the columns have been rearranged to make the result easier to read.

4) The fourth step is only necessary if there is a GROUP BY clause. The fourth step computes aggregate function(s) for each group of rows and reduces each group to a single row. All rows in a group have the same values for the GROUP BY columns. Table 4-16 contains three groups {<IS320,3333>, <IS480,1111>, <IS480,2222>} with computed columns added for aggregate functions in the SELECT and HAVING clauses. Thus, Table 4-16 shows two new columns for the AVG function in the SELECT clause and the COUNT function in the HAVING clause. Note that remaining columns are eliminated at this point because they are not needed in the remaining steps.

TABLE 4-14
Result of Step 2

O.OfferNo	CourseNo	OffYear	OffTerm	StdNo	E.OfferNo	EnrGrade
1111	IS480	2016	FALL	111-11-1111	1111	3.1
2222	IS480	2016	FALL	111-11-1111	2222	3.5
1111	IS480	2016	FALL	222-22-2222	1111	3.2
2222	IS480	2016	FALL	222-22-2222	2222	3.3
1111	IS480	2016	FALL	333-33-3333	1111	3.6
3333	IS320	2016	FALL	111-11-1111	3333	3.3

TABLE 4-15
Result of Step 3

CourseNo	O.OfferNo	OffYear	OffTerm	StdNo	E.OfferNo	EnrGrade
IS320	3333	2016	FALL	111-11-1111	3333	3.3
IS480	1111	2016	FALL	111-11-1111	1111	3.1
IS480	1111	2016	FALL	222-22-2222	1111	3.2
IS480	1111	2016	FALL	333-33-3333	1111	3.6
IS480	2222	2016	FALL	111-11-1111	2222	3.5
IS480	2222	2016	FALL	222-22-2222	2222	3.3

TABLE 4-16
Result of Step 4

CourseNo	O.OfferNo	AvgGrade	Count(*)
IS320	3333	3.3	1
IS480	1111	3.3	3
IS480	2222	3.4	2

[4] In other words, when using the GROUP BY clause, every column in the SELECT clause should either be in the GROUP BY clause or be part of an expression with an aggregate function.

5) The fifth step eliminates rows that do not satisfy the HAVING condition. Table 4-17 shows that the first row in Table 4-16 is removed because it fails the HAVING condition. Note that the HAVING clause specifies a restriction operation for groups of rows. The HAVING clause cannot be present without a preceding GROUP BY clause. The conditions in the HAVING clause always relate to groups of rows, not to individual rows. Conditions in the HAVING clause should involve aggregate functions.
6) The sixth step sorts the results according to the ORDER BY clause. Note that the ORDER BY clause is optional. Table 4-18 shows the result table after sorting.
7) The seventh step performs a final projection. Columns appearing in the result of step 6 are eliminated if they do not appear in the SELECT clause. Table 4-19 (identical to Table 4-12) shows the result after the projection of step 6. The COUNT(*) column is eliminated because it does not appear in the SELECT list. The seventh step (projection) occurs after the sixth step (sorting) because the ORDER BY clause can contain columns that do not appear in the SELECT list.

This section finishes by discussing three major lessons about the conceptual evaluation process. These lessons are more important to remember than the specific details about the conceptual process.

- GROUP BY conceptually occurs after WHERE. If you have an error in a SELECT statement involving WHERE or GROUP BY, the problem is most likely in the WHERE clause. You can check the intermediate results after the WHERE clause by submitting a SELECT statement without the GROUP BY clause.
- Grouping occurs only one time in the evaluation process. If your problem involves more than one independent aggregate calculation, you may need more than one SELECT statement. Query requirements involving multiple independent aggregate calculations are uncommon so this chapter does not cover them.
- Using sample tables can help you analyze difficult problems. You should not need to perform the entire evaluation process. Rather, you can use sample tables to understand only the difficult part. Section 4.5 and Chapter 9 depict the use of sample tables to help analyze difficult problems.

TABLE 4-17
Result of Step 5

CourseNo	O.OfferNo	AvgGrade	Count(*)
IS480	1111	3.3	3
IS480	2222	3.4	2

TABLE 4-18
Result of Step 6

CourseNo	O.OfferNo	AvgGrade	Count(*)
IS480	2222	3.4	3
IS480	1111	3.3	2

TABLE 4-19
Result of Step 7

CourseNo	O.OfferNo	AvgGrade
IS480	2222	3.4
IS480	1111	3.3

4.4 CRITICAL QUESTIONS FOR QUERY FORMULATION

The conceptual evaluation process depicted in Figure 4.2 should help you understand the meaning of most SELECT statements, but it will probably not help you to formulate queries. Query formulation involves a conversion from a problem statement into a statement of a database language such as SQL as shown in Figure 4.3. In between the problem statement and the database language statement, you convert the problem statement into a database representation. Typically, the difficult part is to convert the problem statement into a database representation. This conversion involves a detailed knowledge of a database especially tables, relationships, and data types along with careful attention to possible ambiguities in a problem statement. The critical questions presented in this section provide a structured process to convert a problem statement into a database representation.

Critical Questions for Query Formulation: provide a checklist to convert a problem statement into a database representation consisting of tables, columns, table connection operations, and row grouping requirements.

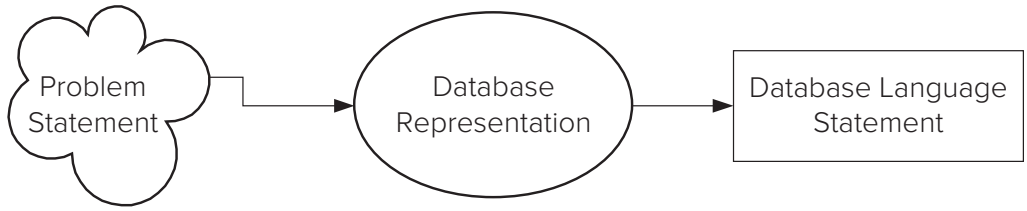
In converting from the problem statement into a database representation, you should answer three critical questions. Table 4-20 summarizes the analysis of the critical questions.

What tables are needed? For the first question, you should match data requirements to columns and tables. You should identify columns for output and conditions as well as intermediate tables to connect other tables. For example, if you want to join the *Student* and *Offering* tables, the *Enrollment* table should be included because it provides a connection to these tables. The *Student* and *Offering* tables cannot be combined directly. All tables needed in the query should be listed in the FROM clause.

How are the tables combined? For the second question, most tables are combined by a join operation. In Chapter 9, you will use the outer join, difference, and division operators to combine tables. For now, you should concentrate on combining tables with joins. You need to identify the matching columns for each join. In most joins, the primary key of a parent table is matched with a foreign key of a related child table. Occasionally, the primary key of the parent table contains multiple columns. In this case, you need to match on all columns. In some situations, the matching columns do not involve a primary key/foreign key combination. You can perform a join as long as the matching columns have compatible data types. For example, when joining customer tables from different databases, a common primary key may not exist. Joining on other columns such as name, address, and so on may be necessary.

Does the output involve individual rows or groups of rows? For the third question, you should look for computations involving aggregate functions in the problem statement. For example, the problem "list the name and average grade of students" contains an aggregate computation. Problems referencing an aggregate function indicate that the output relates to groups of rows. Hence the SELECT statement requires a GROUP BY clause. If the problem contains conditions with aggregate functions, a HAVING clause should accompany the GROUP BY clause. For example, the problem

FIGURE 4.3
Query Formulation Process



Question	Analysis Tips
What tables are needed?	Match columns to output data requirements and conditions to test. If tables are not directly related, identify intermediate tables to provide a join path between tables.
How are the tables combined?	Most tables are combined using a primary key from a parent table to a foreign key of a child table. More difficult problems may involve other join conditions as well as other combining operators (outer join, difference, or division).
Does the output involve individual rows or groups of rows?	Identify aggregate functions used in output data requirements and conditions to test. A SELECT statement requires a GROUP BY clause if aggregate functions are needed. If conditions involve aggregate functions, the statement needs a HAVING clause.

TABLE 4-20
Summary of Critical Questions for Query Formulation

"list the offer number of course offerings with more than 30 students" needs a HAVING clause with a condition involving the COUNT function.

After answering these questions, you are ready to convert the database representation into a database language statement. To help in this process, you should develop a collection of statements for each kind of relational algebra operator using a database that you understand well. For example, you should have statements for problems that involve join operations, joins with grouping, and joins with grouping conditions. As you increase your understanding of SQL, this conversion will become easy for most problems. For difficult problems such as those discussed in Section 4.5 and Chapter 9, relying on similar problems may be necessary because difficult problems are not common.

4.5 REFINING QUERY FORMULATION SKILLS WITH EXAMPLES

Let's apply your query formulation skills and knowledge of the SELECT statement to more difficult problems. All problems in this section involve the parts of the SELECT statement discussed in Sections 4.2 and 4.3. The problems involve more difficult aspects such as joining more than two tables, grouping after joins of several tables, joining a table to itself, and traditional set operators.

4.5.1 Joining Multiple Tables with the Cross Product Style

We begin with a number of join problems that are formulated using cross product operations in the FROM clause. This way to formulate joins is known as the **cross product style** because of the implied cross product operations. Query language compilers recognize the join conditions in the WHERE clause so cross product operations are not actually performed. The next subsection uses join operations in the FROM clause to contrast the ways that joins can be expressed.

Cross Product Style
lists tables in the FROM clause and join conditions in the WHERE clause. The cross product style is easy to read but does not support outer join operations.

In Example 4.30, some student rows appear more than once in the result. For example, Roberto Morales appears twice. Because of the 1-M relationship between the *Student* and *Enrollment* tables, a *Student* row can match multiple *Enrollment* rows.

Example 4.30

Joining Two Tables

List the student name, offering number, and grade of students who have a grade ≥ 3.5 in a course offering.

```
SELECT StdFirstName, StdLastName, OfferNo, EnrGrade
 FROM Student, Enrollment
 WHERE EnrGrade >= 3.5
   AND Student.StdNo = Enrollment.StdNo
```

StdFirstName	StdLastName	OfferNo	EnrGrade
CANDY	KENDALL	1234	3.5
MARIAH	DODGE	1234	3.8
HOMER	WELLS	4321	3.5
ROBERTO	MORALES	4321	3.5
BOB	NORBERT	5679	3.7
ROBERTO	MORALES	5679	3.8
MARIAH	DODGE	6666	3.6
LUKE	BRAZZI	7777	3.7
BOB	NORBERT	9876	3.5
WILLIAM	PILGRIM	9876	4.0

Examples 4.31 and 4.32 depict duplicate elimination after a join. In Example 4.31, some students appear more than once. Because only columns from the *Student* table are used in the output, duplicate rows appear. When you join a parent table to a child table and show only columns from the parent table in the result, duplicate rows appear in the result if a parent row matches with more than one child row. To eliminate duplicate rows, you should use the DISTINCT keyword as shown in Example 4.32.

Example 4.31

Join with Duplicates

List the names of students who have a grade ≥ 3.5 in a course offering.

```
SELECT StdFirstName, StdLastName
 FROM Student, Enrollment
 WHERE EnrGrade >= 3.5
   AND Student.StdNo = Enrollment.StdNo
```

StdFirstName	StdLastName
CANDY	KENDALL
MARIAH	DODGE
HOMER	WELLS
ROBERTO	MORALES
BOB	NORBERT
ROBERTO	MORALES
MARIAH	DODGE
LUKE	BRAZZI
BOB	NORBERT
WILLIAM	PILGRIM

Example 4.32

Join with Duplicates Removed

List the student names (without duplicates) that have a grade ≥ 3.5 in a course offering.

```
SELECT DISTINCT StdFirstName, StdLastName
 FROM Student, Enrollment
 WHERE EnrGrade >= 3.5
   AND Student.StdNo = Enrollment.StdNo
```

StdFirstName	StdLastName
BOB	NORBERT
CANDY	KENDALL
HOMER	WELLS
LUKE	BRAZZI
MARIAH	DODGE
ROBERTO	MORALES
WILLIAM	PILGRIM

Examples 4.33 through 4.36 depict problems involving more than two tables. In these problems, it is important to identify the tables in the FROM clause. You should examine conditions to test as well as columns in the result. In Example 4.35, the *Enrollment* table is needed even though it does not supply columns in the result or conditions to test. The *Enrollment* table is needed to connect the *Student* table with the *Offering* table. Example 4.36 extends Example 4.35 with details from the *Course* table. All five tables are needed to supply outputs, to test conditions, and to connect other tables.

Example 4.33

Joining Three Tables with Columns from Only Two Tables

List the student name and the offering number in which the grade is greater than 3.7 and the offering is given in fall 2016.

```
SELECT StdFirstName, StdLastName, Enrollment.OfferNo
 FROM Student, Enrollment, Offering
 WHERE Student.StdNo = Enrollment.StdNo
   AND Offering.OfferNo = Enrollment.OfferNo
   AND OffYear = 2016 AND OffTerm = 'FALL'
   AND EnrGrade >= 3.7
```

StdFirstName	StdLastName	OfferNo
MARIAH	DODGE	1234

Example 4.34

Joining Three Tables with Columns from Only Two Tables

List Leonard Vince's teaching schedule in fall 2016. For each course, list the offering number, course number, number of units, days, location, and time.

```
SELECT OfferNo, Offering.CourseNo, CrsUnits, OffDays,
       OffLocation, OffTime
 FROM Faculty, Course, Offering
 WHERE Faculty.FacNo = Offering.FacNo
   AND Offering.CourseNo = Course.CourseNo
   AND OffYear = 2016 AND OffTerm = 'FALL'
   AND FacFirstName = 'LEONARD'
   AND FacLastName = 'VINCE'
```

OfferNo	CourseNo	CrsUnits	OffDays	OffLocation	OffTime
1234	IS320	4	MW	BLM302	10:30 AM
4321	IS320	4	TTH	BLM214	3:30 PM

Example 4.35

Joining Four Tables

List Bob Norbert's course schedule in spring 2017. For each course, list the offering number, course number, days, location, time, and faculty name.

```
SELECT Offering.OfferNo, Offering.CourseNo, OffDays,
       OffLocation, OffTime, FacFirstName, FacLastName
 FROM Faculty, Offering, Enrollment, Student
 WHERE Offering.OfferNo = Enrollment.OfferNo
   AND Student.StdNo = Enrollment.StdNo
   AND Faculty.FacNo = Offering.FacNo
   AND OffYear = 2017 AND OffTerm = 'SPRING'
   AND StdFirstName = 'BOB'
   AND StdLastName = 'NORBERT'
```

OfferNo	CourseNo	OffDays	OffLocation	OffTime	FacFirstName	FacLastName
5679	IS480	TTH	BLM412	3:30 PM	CRISTOPHER	COLAN
9876	IS460	TTH	BLM307	1:30 PM	LEONARD	FIBON

Example 4.36

Joining Five Tables

List Bob Norbert's course schedule in spring 2017. For each course, list the offering number, course number, days, location, time, course units, and faculty name.

```
SELECT Offering.OfferNo, Offering.CourseNo, OffDays,
       OffLocation, OffTime, CrsUnits, FacFirstName,
       FacLastName
 FROM Faculty, Offering, Enrollment, Student, Course
 WHERE Faculty.FacNo = Offering.FacNo
   AND Offering.OfferNo = Enrollment.OfferNo
   AND Student.StdNo = Enrollment.StdNo
   AND Offering.CourseNo = Course.CourseNo
   AND OffYear = 2017 AND OffTerm = 'SPRING'
   AND StdFirstName = 'BOB'
   AND StdLastName = 'NORBERT'
```

OfferNo	CourseNo	OffDays	OffLocation	OffTime	CrsUnits	FacFirstName	FacLastName
5679	IS480	TTH	BLM412	3:30 PM	4	CRISTOPHER	COLAN
9876	IS460	TTH	BLM307	1:30 PM	4	LEONARD	FIBON

Example 4.37 demonstrates another way to combine the *Student* and *Faculty* tables. In Example 4.35, you saw it was necessary to combine the *Student*, *Enrollment*, *Offering*, and *Faculty* tables to find faculty teaching a specified student. To find students who are on the faculty (perhaps teaching assistants), the tables can be joined directly. Combining the *Student* and *Faculty* tables in this way is similar to an intersection operation. However, intersection cannot actually be performed here because the *Student* and *Faculty* tables are not union compatible.

Example 4.37

Joining Two Tables without Matching on a Primary and Foreign Key

List students who are on the faculty. Include all student columns in the result.

```
SELECT Student.*
 FROM Student, Faculty
 WHERE StdNo = FacNo
```

StdNo	StdFirstName	StdLastName	StdCity	StdState	StdMajor	StdClass	StdGPA	StdZip
876-54-3210	CRISTOPHER	COLAN	SEATTLE	WA	IS	SR	4.00	98114-1332

A minor point about Example 4.37 is the use of the * after the SELECT keyword. Prefixing the * with a table name and period indicates all columns of the specified table are in the result. Using an * without a table name prefix indicates that all columns from all FROM tables are in the result.

4.5.2 Joining Multiple Tables with the Join Operator Style

As demonstrated in Section 4.2, join operations can be expressed directly in the FROM clause using the INNER JOIN and ON keywords. This **join operator style** can be used to combine any number of tables. To ensure that you are comfortable using this style, this subsection presents examples of multiple table joins beginning with a two-table join in Example 4.38. Note that these examples do not execute in older Oracle versions (before 9i).

Join Operator Style lists join operations in the FROM clause using the INNER JOIN and ON keywords. The join operator style can be somewhat difficult to read for statements with many join operations, but it supports outer join operations as shown in Chapter 9.

Example 4.38 (Access and Oracle)

Join Two Tables Using the Join Operator Style

Retrieve the name, city, and grade of students who have a high grade (greater than or equal to 3.5) in a course offering.

```
SELECT StdFirstName, StdLastName, StdCity, EnrGrade
 FROM Student INNER JOIN Enrollment
   ON Student.StdNo = Enrollment.StdNo
 WHERE EnrGrade >= 3.5
```

StdFirstName	StdLastName	StdCity	EnrGrade
CANDY	KENDALL	TACOMA	3.5
MARIAH	DODGE	SEATTLE	3.8
HOMER	WELLS	SEATTLE	3.5
ROBERTO	MORALES	SEATTLE	3.5
BOB	NORBERT	BOTHELL	3.7
ROBERTO	MORALES	SEATTLE	3.8
MARIAH	DODGE	SEATTLE	3.6
LUKE	BRAZZI	SEATTLE	3.7
BOB	NORBERT	BOTHELL	3.5
WILLIAM	PILGRIM	BOTHELL	4.0

The join operator style can be extended to handle any number of tables. Think of the join operator style as writing a mathematical formula with lots of parentheses. To add another part to the formula, you need to add the variable, operator, and another level of parentheses. For example, with the formula (X + Y) * Z, you can add another operation as ((X + Y) * Z) / W. This same principle can be applied with the join operator style. Examples 4.39 and 4.40a extend Example 4.38 with additional conditions that need other tables. In both examples, another INNER JOIN is added to the end of the previous INNER JOIN operations. The INNER JOIN could also have been added at the beginning or middle if desired. The ordering of INNER JOIN operations is not important.

Example 4.39 (Access and Oracle)

Join Three Tables using the Join Operator Style

Retrieve the name, city, and grade of students who have a high grade (greater than or equal 3.5) in a course offered in fall 2016.

```
SELECT StdFirstName, StdLastName, StdCity, EnrGrade
 FROM ( Student INNER JOIN Enrollment
     ON Student.StdNo = Enrollment.StdNo )
  INNER JOIN Offering
    ON Offering.OfferNo = Enrollment.OfferNo
 WHERE EnrGrade >= 3.5 AND OffTerm = 'FALL'
   AND OffYear = 2016
```

StdFirstName	StdLastName	StdCity	EnrGrade
CANDY	KENDALL	TACOMA	3.5
MARIAH	DODGE	SEATTLE	3.8
HOMER	WELLS	SEATTLE	3.5
ROBERTO	MORALES	SEATTLE	3.5

Example 4.40a (Access and Oracle)

Join Four tables Using the Join Operator Style

Retrieve the name, city, and grade of students who have a high grade (greater than or equal to 3.5) in a course offered in fall 2016 taught by Leonard Vince.

```
SELECT StdFirstName, StdLastName, StdCity, EnrGrade
 FROM ( (Student INNER JOIN Enrollment
       ON Student.StdNo = Enrollment.StdNo )
  INNER JOIN Offering
     ON Offering.OfferNo = Enrollment.OfferNo )
  INNER JOIN Faculty ON Faculty.FacNo = Offering.FacNo
 WHERE EnrGrade >= 3.5 AND OffTerm = 'FALL'
   AND OffYear = 2016 AND FacFirstName = 'LEONARD'
   AND FacLastName = 'VINCE'
```

StdFirstName	StdLastName	StdCity	EnrGrade
CANDY	KENDALL	TACOMA	3.5
MARIAH	DODGE	SEATTLE	3.8
HOMER	WELLS	SEATTLE	3.5
ROBERTO	MORALES	SEATTLE	3.5

Parentheses makes the join operator style cumbersome. Because the order of executing join operations does not matter, parentheses should not be required. Oracle does not requ8ire parentheses, consistent with the definition of the join operator. However, Microsoft Access SQL still requires parentheses, inconsistent with the definition of the join operator. Example 4.40b is identical to Example 4.40a except for the lack of parentheses in the FROM clause.

Example 4.40b (Oracle only)

Join Four Tables using the Join Operator Style without Parentheses. This statement generates a syntax error in Access.

Retrieve the name, city, and grade of students who have a high grade (greater than or equal to 3.5) in a course offered in fall 2016 taught by Leonard Vince.

```
SELECT StdFirstName, StdLastName, StdCity, EnrGrade
 FROM Student INNER JOIN Enrollment
       ON Student.StdNo = Enrollment.StdNo
  INNER JOIN Offering
     ON Offering.OfferNo = Enrollment.OfferNo
  INNER JOIN Faculty ON Faculty.FacNo = Offering.FacNo
 WHERE EnrGrade >= 3.5 AND OffTerm = 'FALL'
   AND OffYear = 2016 AND FacFirstName = 'LEONARD'
   AND FacLastName = 'VINCE'
```

The cross product and join operator styles can be mixed as demonstrated in Example 4.41. In most cases, it is preferable to use only one style. Mixing styles can confuse the reader of the statement. You may also forget the join conditions in the WHERE clause leading to excessive resource consumption and many extra rows in the result.

Example 4.41 (Access and Oracle)

Combine the Cross Product and Join Operator Styles

Retrieve the name, city, and grade of students who have a high grade (greater than or equal to 3.5) in a course offered in fall 2016 taught by Leonard Vince (same result as Example 4.33).

```
SELECT StdFirstName, StdLastName, StdCity, EnrGrade
 FROM ( (Student INNER JOIN Enrollment
    ON Student.StdNo = Enrollment.StdNo )
   INNER JOIN Offering
    ON Offering.OfferNo = Enrollment.OfferNo ),
   Faculty
 WHERE EnrGrade >= 3.5 AND OffTerm = 'FALL'
   AND OffYear = 2016 AND FacFirstName = 'LEONARD'
   AND FacLastName = 'VINCE'
   AND Faculty.FacNo = Offering.FacNo
```

The choice between the cross product and the join operator styles is largely a matter of preference. In the cross product style, it is easy to see the tables in the SELECT statement. The cross product style has been criticized because users sometimes forget a join condition in the WHERE clause leading to a disk intensive cross product operation. For multiple joins, the join operator style can be difficult to read because of nested parentheses. The primary advantage of the join operator style is that you can formulate queries involving outer joins as described in Chapter 9.

You should be comfortable reading both join styles even if you only write SQL statements using one style. You may need to maintain statements written with both styles. In addition, some visual query languages generate code in one of the styles. For example, Query Design, the visual query language of Microsoft Access, generates code in the join operator style.

4.5.3 Self-Joins and Multiple Joins between Two Tables

Self-Join
a join between a table and itself (two copies of the same table). Self-joins are useful for finding relationships among rows of the same table.

Example 4.42 demonstrates a **self-join**, a join involving a table with itself. A self-join is necessary to find relationships among rows of the same table. The foreign key, *FacSupervisor*, shows relationships among *Faculty* rows. To find the supervisor name of a faculty member, match on the *FacSupervisor* column with the *FacNo* column. You should formulate the statement imagining that you are working with two copies of the *Faculty* table. One copy plays the role of the subordinate, while the other copy plays the role of the superior. In SQL, a self-join requires alias names (*Subr* and *Supr*) in the FROM clause to distinguish between the two roles or copies.

Example 4.42

Self-join

List faculty members who have a higher salary than their supervisor. List the faculty number, name, and salary of the faculty and supervisor.

```
SELECT Subr.FacNo, Subr.FacLastName, Subr.FacSalary,
       Supr.FacNo, Supr.FacLastName, Supr.FacSalary
 FROM Faculty Subr, Faculty Supr
 WHERE Subr.FacSupervisor = Supr.FacNo
   AND Subr.FacSalary > Supr.FacSalary
```

Subr.FacNo	Subr.FacLastName	Subr.FacSalary	Supr.FacNo	Supr.FacLastName	Supr.FacSalary
987-65-4321	MILLS	75000.00	765-43-2109	MACON	65000.00

Problems involving self-joins can be difficult to understand. If you are having trouble understanding Example 4.42, use the conceptual evaluation process to help. Start with a small *Faculty* table. Copy this table and use the names *Subr* and *Supr* to distinguish between the two copies. Join the two tables over *Subr.FacSupervisor* and *Supr.FacNo*. If you need, derive the join using a cross product operation. You should be able to see that each result row in the join shows a subordinate and supervisor pair.

Problems involving self-referencing (unary) relationships are part of hierarchical queries. In hierarchical queries, a table can be visualized as a tree structure in which every row has at most one parent row. For example, the *Faculty* table has a structure showing an organization hierarchy. At the top, the college dean resides. At the bottom, faculty members without subordinates reside. Similar structures apply to the chart of accounts in accounting systems, part structures in manufacturing systems, and route networks in transportation systems.

A more difficult problem than a self-join is to find all subordinates (direct or indirect) in an organization hierarchy. This problem can be solved with the SELECT statement shown in this chapter if the number of subordinate levels is known. One join for each subordinate level is needed. Without knowing the number of subordinate levels, this problem cannot be done in SQL-92 although it can be solved in SQL:2016 using recursive common table expressions or proprietary SQL extensions. In SQL-92, tree-structured queries can be solved by using SQL inside a programming language. Chapter 9 provides details about formulation of hierarchical queries using both recursive common table expressions and proprietary Oracle SQL extensions.

Example 4.43 shows another difficult join problem. This problem involves two joins between the same two tables (*Offering* and *Faculty*). Alias table names (*O1* and *O2*) are needed to distinguish between the two copies of the *Offering* table used in the statement.

Example 4.43

More Than One Join between Tables using Alias Table Names

List the names of faculty members and the course number for which the faculty member teaches the same course number as his or her supervisor in 2017.

```
SELECT FacFirstName, FacLastName, O1.CourseNo
 FROM Faculty, Offering O1, Offering O2
 WHERE Faculty.FacNo = O1.FacNo
   AND Faculty.FacSupervisor = O2.FacNo
   AND O1.OffYear = 2017 AND O2.OffYear = 2017
   AND O1.CourseNo = O2.CourseNo
```

FacFirstName	FacLastName	CourseNo
LEONARD	VINCE	IS320
LEONARD	FIBON	IS320

If this problem is too difficult, you should use the conceptual evaluation process (Figure 4.2) with sample tables to gain insight. Perform a join between the sample *Faculty* and *Offering* tables, then join this result to another copy of *Offering* (*O2*) matching

FacSupervisor with *O2.FacNo*. In the resulting table, select the rows that have matching course numbers and year equal to 2017.

4.5.4 Combining Joins and Grouping

Example 4.44 demonstrates the reason it is sometimes necessary to group on multiple columns. After studying Example 4.44, you might be confused about the necessity to group on both *OfferNo* and *CourseNo*. One simple explanation is that any column appearing in a SELECT list must be either a grouping column or an aggregate expression. However, this explanation does not quite tell the entire story. Grouping on *OfferNo* alone produces the same values for the computed column (*NumStudents*) because *OfferNo* is the primary key. Including non-unique columns such as *CourseNo* adds information to each result row but does not change the aggregate calculations. If you do not understand this point, use sample tables to demonstrate it. When evaluating your sample tables, remember that joins occur before grouping as indicated in the conceptual evaluation process.

Example 4.44

Join with Grouping on Multiple Columns

List the course number, the offering number, and the number of students enrolled. Only include courses offered in spring 2017.

```
SELECT CourseNo, Enrollment.OfferNo,
       Count(*) AS NumStudents
 FROM Offering, Enrollment
 WHERE Offering.OfferNo = Enrollment.OfferNo
   AND OffYear = 2017 AND OffTerm = 'SPRING'
 GROUP BY Enrollment.OfferNo, CourseNo
```

CourseNo	OfferNo	NumStudents
FIN480	7777	3
IS460	9876	7
IS480	5679	6

Example 4.45 demonstrates another problem involving joins and grouping. An important part of this problem is the need for the *Student* table and the HAVING condition. They are needed because the problem statement refers to an aggregate function involving the *Student* table.

Example 4.45

Joins, grouping, and a grouping condition

List the course number, offer number, and average student GPA for course offerings taught in fall 2016 in which the average GPA is greater than 3.0.

```
SELECT CourseNo, Enrollment.OfferNo,
       Avg(StdGPA) AS AvgGPA
 FROM Student, Offering, Enrollment
 WHERE Offering.OfferNo = Enrollment.OfferNo
   AND Enrollment.StdNo = Student.StdNo
   AND OffYear = 2016 AND OffTerm = 'FALL'
 GROUP BY CourseNo, Enrollment.OfferNo
 HAVING Avg(StdGPA) > 3.0
```

CourseNo	OfferNo	AvgGPA
IS320	1234	3.23
IS320	4321	3.03

Example 4.46 demonstrates a problem involving grouping on a computed column, the hiring year of the faculty. This problem requires a different formulation in Access and Oracle because the functions to extract the year component (**year** function in Access and **to_char** function in Oracle) are different. In both formulations, the grouping column must be the expression, not the result column name (*FacHireYear*). You will get a syntax error if the result column name is used instead of the expression.

Example 4.46 (Access)

Joins and grouping on a computed column

List the hiring year, offering year, and number of course offerings taught by faculty hired after 2003. The **year** function extracts the year component of a date column.

```
SELECT year(FacHireDate) AS FacHireYear, OffYear,
       COUNT(*) as NumCourses
 FROM Offering, Faculty
 WHERE Offering.FacNo = Faculty.FacNo
   AND year(FacHireDate) > 2003
 GROUP BY year(FacHireDate), OffYear
```

FacHireYear	OffYear	NumCourses
2004	2016	2
2004	2017	1
2005	2017	1
2006	2017	2
2008	2017	1
2009	2017	2

Example 4.46 (Oracle)

Joins and grouping on a computed column

List the hiring year, offering year, and number of course offerings taught by faculty hired after 2003. The **to_char** function extracts the year component of a date column using the "YYYY" format string.

```
SELECT to_number(to_char(FacHireDate, 'YYYY' ) )
        AS FacHireYear, OffYear, COUNT(*) as NumCourses
 FROM Offering, Faculty
 WHERE Offering.FacNo = Faculty.FacNo
   AND to_number(to_char(FacHireDate, 'YYYY' ) ) > 2003
 GROUP BY to_number(to_char(FacHireDate, 'YYYY' ) ),
          OffYear
```

4.5.5 Traditional Set Operators in SQL

In SQL, you can directly use the traditional set operators with the UNION, INTERSECT, and EXCEPT keywords. Some DBMSs including Microsoft Access do not support the INTERSECT and EXCEPT keywords. As with relational algebra, you must

ensure that the tables are union compatible. In SQL, you can use a SELECT statement to make tables compatible by listing only compatible columns. Examples 4.47 through 4.49 demonstrate set operations on column subsets of the *Faculty* and *Student* tables. The columns have been renamed to avoid confusion.

Example 4.47

UNION Query

Show all faculty and students. Only show the common columns in the result.

```
SELECT FacNo AS PerNo, FacFirstName AS FirstName,
       FacLastName AS LastName, FacCity AS City,
       FacState AS State
 FROM Faculty
     UNION
SELECT StdNo AS PerNo, StdFirstName AS FirstName,
       StdLastName AS LastName, StdCity AS City,
       StdState AS State
 FROM Student
```

PerNo	FirstName	LastName	City	State
098765432	LEONARD	VINCE	SEATTLE	WA
123456789	HOMER	WELLS	SEATTLE	WA
124567890	BOB	NORBERT	BOTHELL	WA
234567890	CANDY	KENDALL	TACOMA	WA
345678901	WALLY	KENDALL	SEATTLE	WA
456789012	JOE	ESTRADA	SEATTLE	WA
543210987	VICTORIA	EMMANUEL	BOTHELL	WA
567890123	MARIAH	DODGE	SEATTLE	WA
654321098	LEONARD	FIBON	SEATTLE	WA
678901234	TESS	DODGE	REDMOND	WA
765432109	NICKI	MACON	BELLEVUE	WA
789012345	ROBERTO	MORALES	SEATTLE	WA
876543210	CRISTOPHER	COLAN	SEATTLE	WA
890123456	LUKE	BRAZZI	SEATTLE	WA
901234567	WILLIAM	PILGRIM	BOTHELL	WA
987654321	JULIA	MILLS	SEATTLE	WA

Example 4.48 (Oracle only)

INTERSECT Query

Show teaching assistants, graduate students who also teach courses so they appear in both *Student* and *Faculty* tables. Only show the common columns in the result.

```
SELECT FacNo AS PerNo, FacFirstName AS FirstName,
       FacLastName AS LastName, FacCity AS City,
       FacState AS State
 FROM Faculty
   INTERSECT
SELECT StdNo AS PerNo, StdFirstName AS FirstName,
       StdLastName AS LastName, StdCity AS City,
       StdState AS State
 FROM Student
```

PerNo	FirstName	LastName	City	State
876543210	CRISTOPHER	COLAN	SEATTLE	WA

Example 4.49 (Oracle only)

Difference Query

Show faculty who are not students (faculty who are not graduate students). Only show the common columns in the result. Oracle uses the MINUS keyword instead of the EXCEPT keyword used in SQL:2016.

```
SELECT FacNo AS PerNo, FacFirstName AS FirstName,
       FacLastName AS LastName, FacCity AS City,
       FacState AS State
 FROM Faculty
     MINUS
SELECT StdNo AS PerNo, StdFirstName AS FirstName,
       StdLastName AS LastName, StdCity AS City,
       StdState AS State
 FROM Student
```

PerNo	FirstName	LastName	City	State
098765432	LEONARD	VINCE	SEATTLE	WA
543210987	VICTORIA	EMMANUEL	BOTHELL	WA
654321098	LEONARD	FIBON	SEATTLE	WA
765432109	NICKI	MACON	BELLEVUE	WA
987654321	JULIA	MILLS	SEATTLE	WA

By default, duplicate rows are removed in the results of SQL statements with the UNION, INTERSECT, and EXCEPT (MINUS) keywords. If you want to retain duplicate rows, use the ALL keyword after the operator. For example, the UNION ALL keyword performs a union operation but does not remove duplicate rows.

4.6 SQL MODIFICATION STATEMENTS

The modification statements support adding new rows (INSERT), changing columns in one or more rows (UPDATE), and deleting one or more rows (DELETE). Although well designed and powerful, they are not as widely used as the SELECT statement because data entry forms are easier to use for most users.

The INSERT statement has two formats as demonstrated in Examples 4.50 and 4.51. In the first format, one row at a time can be added. You specify values for each column with the VALUES clause. You must format the constant values appropriately for each column. Refer to the documentation of your DBMS for details about specifying constants, especially string and date constants. Specifying a null value for a column is also not standard across DBMSs. In some DBMSs, you simply omit the column name and the value. In other systems, you use a particular symbol for a null value. Of course, you must be careful that the table definition permits null values for the column of interest. Otherwise, the INSERT statement will be rejected.

Example 4.50

Single Row Insert

Insert a row into the *Student* table supplying values for all columns.

```
INSERT INTO Student
 (StdNo, StdFirstName, StdLastName,
  StdCity, StdState, StdZip, StdClass, StdMajor, StdGPA)
 VALUES ('999999999', 'JOE', 'STUDENT', 'SEATAC',
         'WA', '98042-1121', 'FR', 'IS', 0.0)
```

The second format of the INSERT statement supports addition of a set of records as shown in Example 4.51. Using the SELECT statement inside the INSERT statement, you can specify any derived set of rows. You can use the second format when you want to create temporary tables for specialized processing.

Example 4.51

Multiple Row Insert

Assume a new table *ISStudent* has been previously created. *ISStudent* has the same columns as *Student*. This INSERT statement copies rows from *Student* into *ISStudent*.

```
INSERT INTO ISStudent
 SELECT * FROM Student WHERE StdMajor = 'IS'
```

The UPDATE statement allows one or more rows to be changed, as shown in Examples 4.52 and 4.53. Any number of columns can be changed, although typically only one column at a time is changed. When changing the primary key, update rules on referenced rows may not allow the operation.

Example 4.52

Single Column Update

Give faculty members in the MS department a 10 percent raise. Four rows are updated.

```
UPDATE Faculty
 SET FacSalary = FacSalary * 1.1
 WHERE FacDept = 'MS'
```

Example 4.53

Update Multiple Columns

Change the major and class of Homer Wells. One row is updated.

```
UPDATE Student
 SET StdMajor = 'ACCT', StdClass = 'SO'
 WHERE StdFirstName = 'HOMER'
   AND StdLastName = 'WELLS'
```

The DELETE statement allows one or more rows to be removed, as shown in Examples 4.54 and 4.55. DELETE is subject to the rules on referenced rows. For example, a *Student* row cannot be deleted if related *Enrollment* rows exist and the deletion action is restrict.

Example 4.54

Delete Selected Rows

Delete all IS majors who are seniors. Three rows are deleted.

```
DELETE FROM Student
 WHERE StdMajor = 'IS' AND StdClass = 'SR'
```

Example 4.55

Delete All Rows in a Table.

Delete all rows in the *ISStudent* table. This example assumes that the *ISStudent* table has been previously created.

```
DELETE FROM ISStudent
```

Sometimes it is useful for the condition inside the WHERE clause of an UPDATE or DELETE statement to reference rows from another table. Microsoft Access supports the join operator style to combine tables as shown in Examples 4.56 and 4.57. You cannot use the cross product style inside an UPDATE or DELETE statement. Chapter 9 shows another way to reference other tables in an UPDATE or DELETE statement that most DBMSs (including Access and Oracle) support.

Example 4.56 (Access only)

UPDATE Statement Using the Join Operator Style

Update the location of offerings taught by Leonard Vince in 2016 to BLM412. Two *Offering* rows are updated.

```
UPDATE Offering INNER JOIN Faculty
        ON Offering.FacNo = Faculty.FacNo
 SET OffLocation = 'BLM412'
 WHERE OffYear = 2016 AND FacFirstName = 'LEONARD'
   AND FacLastName = 'VINCE'
```

Example 4.57 (Access only)

DELETE Statement Using the Join Operator Style

Delete offerings taught by Leonard Vince. Three *Offering* rows are deleted. In addition, this statement deletes related rows in the *Enrollment* table because the ON DELETE clause is set to CASCADE.

```
DELETE Offering.*
 FROM Offering INNER JOIN Faculty
   ON Offering.FacNo = Faculty.FacNo
 WHERE FacFirstName = 'LEONARD'
   AND FacLastName = 'VINCE'
```

4.7 QUERY FORMULATION ERRORS AND CODING PRACTICES

To develop query formulation skills, this chapter presented many example statements and guidelines. You should apply these guidelines and use these example statements to learn SQL coding patterns. Example statements are useful to help you learn correct syntax as well as coding patterns for complex query formulation problems especially problems in section 4.5 and Chapter 9.

In many years of teaching query formulation, the author has found that correct examples and guidelines are not sufficient. Students also need awareness of incorrect examples with various kinds of errors. Awareness of query formulation errors can help avoid errors as well as diagnose incorrect statements, reducing frustration and increasing confidence. Since even skilled professionals make errors, you should remain vigilant about query formulation errors as your skills progress.

Table 4-21 summarizes major types of errors. Syntax errors are the most frustrating because your statement will not execute. Sometimes error messages from SQL compilers are difficult to understand especially if a statement contains multiple syntax errors. Redundancy and semantic errors are subtle because an SQL compiler does not indicate an error. Errors of redundancy have least severity as the result contains correct rows but extra resource consumption occurs. Semantic errors are more severe as the result contains incorrect rows, either too many rows or missing rows. Missing a join condition is the worst error because of excessive resource consumption. A missing join condition involves a cross product operation instead of a join. For large tables such as student and enrollment tables for a major university, a missing join condition can cause hours of excessive query execution time.

The remainder of this section presents examples of each type of error with identification of each error. You should try to find errors before seeing error identifications in Table 4-22. The examples begin with syntax errors and progress to redundancy and semantic errors.

TBLE 4-21
Summary of Major Error Types in Query Formulation

Error Type	Typical Errors	Severity
Syntax	Missing table, unqualified column name, misspelled keyword, row condition in HAVING clause, missing column in GROUP BY clause, aggregate function in a WHERE condition	No execution with sometimes confusing error message
Redundancy	Extra table, unneeded GROUP BY clause	Execution with correct rows but extra resource consumption
Semantic	Missing row condition, missing parentheses, incorrect condition, missing join condition	Execution but incorrect rows in result; Sometimes excessive resource consumption

Example 4.58

Misspelled Keywords

List the offer number, course number, and faculty number for course offerings scheduled in fall 2016. The Oracle SQL compiler indicates "FROM keyword not found where expected". The Access SQL compiler indicates "Syntax error (missing operator) ...".

```
SELECT OfferNo, CourseNo, FacNo
 FROMM Offering
 WHERRE OffTerm = 'FALL' AND OffYear = 2016
```

Example 4.59

Unqualified Column Name

List the student name and offering number in which the grade is greater than 3.7 and the offering is given in fall 2016. The Oracle SQL compiler indicates "column ambiguously defined". The Access SQL compiler indicates "The specified field 'OfferNo' could refer to more than one table ...".

```
SELECT StdFirstName, StdLastName, OfferNo
 FROM Student, Enrollment, Offering
 WHERE Student.StdNo = Enrollment.StdNo
   AND Offering.OfferNo = Enrollment.OfferNo
   AND OffYear = 2016 AND OffTerm = 'FALL'
   AND EnrGrade >= 3.7
```

Example 4.60

Missing Table

List the student name and the offering number in which the grade is greater than 3.7 and the offering occurred in fall 2016. The Oracle SQL compiler indicates "EnrGrade; invalid identifier. The Access SQL compiler generates a window asking for a parameter value.

```
SELECT StdFirstName, StdLastName, Enrollment.OfferNo
 FROM Student, Offering
 WHERE Student.StdNo = Enrollment.StdNo
   AND Offering.OfferNo = Enrollment.OfferNo
   AND OffYear = 2016 AND OffTerm = 'FALL'
   AND EnrGrade >= 3.7
```

Example 4.61

Row Condition in HAVING Clause

List the course number, offer number, and average student GPA for course offerings taught in fall 2016 in which the average GPA is greater than 3.0. The Oracle SQL compiler generates a syntax error indicating, "not a GROUP BY expression". The Access SQL compiler generates a syntax error indicating, "Your query does not include the specified expression as part of an aggregate function".

```
SELECT CourseNo, Enrollment.OfferNo,
       Avg(StdGPA) AS AvgGPA
 FROM Student, Offering, Enrollment
 WHERE Offering.OfferNo = Enrollment.OfferNo
   AND Enrollment.StdNo = Student.StdNo
   AND OffTerm = 'FALL'
 GROUP BY CourseNo, Enrollment.OfferNo
 HAVING Avg(StdGPA) > 3.0 AND OffYear = 2016
```

Example 4.62

Missing Column in GROUP BY Clause

List the course number, offer number, and average student GPA for course offerings taught in fall 2016 in which the average GPA is greater than 3.0. The Oracle SQL compiler generates a syntax error indicating, "not a GROUP BY expression". The Access SQL compiler generates a syntax error indicating, "Your query does not include the specified expression as part of an aggregate function".

```
SELECT CourseNo, Enrollment.OfferNo,
       Avg(StdGPA) AS AvgGPA
 FROM Student, Offering, Enrollment
 WHERE Offering.OfferNo = Enrollment.OfferNo
   AND Enrollment.StdNo = Student.StdNo
   AND OffTerm = 'FALL' AND OffYear = 2016
 GROUP BY CourseNo
 HAVING Avg(StdGPA) > 3.0
```

Example 4.63

Extra Table

List the student name and the offering number in which the grade is greater than 3.7 and the offering is given in fall 2016. The statement executes with the correct rows in the result.

```
SELECT StdFirstName, StdLastName, Enrollment.OfferNo
 FROM Student, Enrollment, Offering, Course
 WHERE Student.StdNo = Enrollment.StdNo
   AND Offering.OfferNo = Enrollment.OfferNo
   AND Course.CourseNo = Offering.CourseNo
   AND OffYear = 2016 AND OffTerm = 'FALL'
   AND EnrGrade >= 3.7
```

Example 4.64

Unnecessary GROUP BY Clause

List the student name and the offering number in which the grade is greater than 3.7 and the offering is given in fall 2016. The GROUP BY clause causes extra resource consumption.

```
SELECT StdFirstName, StdLastName, Enrollment.OfferNo
 FROM Student, Enrollment, Offering
 WHERE Student.StdNo = Enrollment.StdNo
   AND Offering.OfferNo = Enrollment.OfferNo
   AND OffYear = 2016 AND OffTerm = 'FALL'
   AND EnrGrade >= 3.7
GROUP BY StdFirstName, StdLastName, Enrollment.OfferNo
```

Example 4.65

Missing Parentheses

List the offer number, course number, and faculty number for course offerings scheduled in spring or summer of 2016. Note that the AND operator takes precedence over the OR operator.

```
SELECT OfferNo, CourseNo, FacNo, OffYear, OffTerm
 FROM Offering
 WHERE OffTerm = 'SPRING' OR OffTerm = 'SUMMER'
   AND OffYear = 2016
```

Example 4.66

Missing Join Condition

List the student name and the offering number in which the grade is greater than 3.7 and the offering is given in fall 2016. The result contains extra *Enrollment* rows that do not match *Offering* rows because of the missing join condition. You should remember that joining three tables typically requires two join conditions.

```
SELECT StdFirstName, StdLastName, Enrollment.OfferNo
 FROM Student, Enrollment, Offering
 WHERE Student.StdNo = Enrollment.StdNo
   AND OffYear = 2016 AND OffTerm = 'FALL'
   AND EnrGrade >= 3.7
```

Example 4.67

Missing Condition

List the student name and the offering number in which the grade is greater than 3.7 and the offering is given in fall 2016. The result only contains the correct set of rows because 2017 offerings do not occur in the fall term. If additional rows in fall term of another year are added, the result rows will not be correct.

```
SELECT StdFirstName, StdLastName, Enrollment.OfferNo
 FROM Student, Enrollment, Offering
 WHERE Student.StdNo = Enrollment.StdNo
   AND Offering.OfferNo = Enrollment.OfferNo
   AND OffTerm = 'FALL'
   AND EnrGrade >= 3.7
```

TABLE 4-22
Identification of Errors in Examples

Example	Errors	Error Identification
4.58	Syntax with misspelled keywords	FROMM and WHERRE misspelled
4.59	Syntax with unqualified column name	OfferNo column needs table prefix.
4.60	Syntax with missing table name	Enrollment table missing in FROM clause
4.61	Syntax with row condition in the HAVING clause	OffYear = 2016 should be moved to the WHERE clause.
4.62	Syntax with missing column in the GROUP BY clause	OfferNo should be in the GROUP BY clause.
4.63	Redundancy with extra table	Course table is not needed as it does not provide columns or conditions. Since CourseNo is a required foreign key, join does not ensure that Course exists.
4.64	Redundancy with unneeded GROUP BY clause	GROUP BY clause not needed as statement does not contain aggregate functions in result or conditions.
4.65	Semantic with missing parentheses	Result contains extra rows. Parentheses should be placed around (OffTerm = 'SPRING' OR OffTerm = 'SUMMER'). Always use parentheses when mixing AND/OR operators.
4.66	Semantic with missing join condition	Result contains incorrect rows. Excessive resource consumption with cross product operation combining *Offering* and *Enrollment* tables. Add join condition Offering.OfferNo = Enrollment.OfferNo
4.67	Semantic with missing condition	Add OffYear = 2016. Correct rows because *Offering* table has no fall rows in any other year.

Beyond awareness of errors, you should strive to write statements with good coding practices. The examples in sections 4.2 to 4.6 demonstrate good coding practices. Table 4-23 summarizes poor SQL coding practices, while Example 4.68 demonstrates poor coding practices in a complete SELECT statement. Some of the practices are subjective (such as clause alignment) with multiple ways to achieve good practice. Most practices should be avoided such as incompatible constants.

TABLE 4-23
Summary of Poor SQL Coding Practices

Practice	Definition and impact	Example
Mixed join styles	Using a combination of the cross product and join operator style; Difficult to read and prone to missing join conditions	Example 4.41
Incompatible constant	Constant in a condition with a data type different than column; May lead to incorrect result or slow performance	StdGPA > '3.2'
LIKE operator in date comparison	LIKE operator applies to columns with character data types. May lead to incorrect results; May not be portable across SQL compilers	HireDate LIKE '12/*/2017'
Poor clause alignment	Clauses mixed together and aligned poorly; Difficult to read	Example 4.68
LIKE operator without pattern matching characters	LIKE operator without pattern matching characters is the same as equality (=) comparison. May lead to confusion among statement readers	StdtState LIKE 'WA'

Example 4.68

Poor SQL Coding Practices

List Bob Norbert's course schedule in spring 2017. For each course, list the offering number, course number, days, location, time, course units, and faculty name. The statement contains poor clause alignment, an incompatible constant, and a LIKE operator without a pattern matching character. Example 4.36 shows the original statement with good coding practices.

```
SELECT Offering.OfferNo, Offering.CourseNo, OffDays,
       OffLocation, OffTime, CrsUnits, FacFirstName,
    FacLastName FROM Faculty, Offering, Enrollment, Student, Course WHERE Faculty.FacNo =
Offering.FacNo AND
Offering.OfferNo
  = Enrollment.OfferNo
  AND Student.StdNo = Enrollment.StdNo
AND Offering.CourseNo = Course.CourseNo
   AND OffYear = '2017' AND OffTerm = 'SPRING'
 AND StdFirstName LIKE 'BOB' AND StdLastName = 'NORBERT'
```

CLOSING THOUGHTS

Chapter 4 introduced the fundamental statements of the industry standard Structured Query Language (SQL). SQL has a wide scope covering database definition, manipulation, and control. As a result of careful analysis and compromise, standards groups have produced a well-designed language. SQL has become the common glue that binds the database industry even though strict conformance to the standard is lacking. You will no doubt continually encounter SQL throughout your career.

This chapter has focused on the most widely used parts of the SELECT statement from the foundation of the SQL:2016 standard. Numerous examples were shown to

demonstrate conditions on different data types, complex logical expressions, multiple table joins, summarization of tables with GROUP BY and HAVING, sorting of tables, self joins, and the traditional set operators. To facilitate hands-on usage of SQL, examples were shown for both Oracle and Access with special attention to deviations from the SQL:2016 standard. This chapter also briefly described the modification statements INSERT, UPDATE, and DELETE. These statements are not as complex and widely used as SELECT.

This chapter emphasized two problem-solving guidelines to help you formulate queries. The conceptual evaluation process was presented to demonstrate derivation of result rows for SELECT statements involving joins and grouping. You may find this evaluation process helps in your initial learning of the SELECT statement as well as provides insight on more challenging problems. To help formulate queries, three questions were provided to guide you. You should explicitly or implicitly answer these questions before writing a SELECT statement to solve a problem. An understanding of both the critical questions and the conceptual evaluation process will provide you a solid foundation for using relational databases. Even with these formulation aids, you need to work many problems to learn query formulation and the SELECT statement.

This chapter covered an important subset of the SELECT statement. Other parts of the SELECT statement not covered in this chapter are outer joins, nested queries, division problems, null value effects, and hierarchical queries. Chapter 9 covers advanced query formulation and additional parts of the SELECT statement so that you can gain a competitive advantage in your database skills.

REVIEW CONCEPTS

- SQL consists of statements for database definition (CREATE TABLE, ALTER TABLE, etc.), database manipulation (SELECT, INSERT, UPDATE, and DELETE), and database control (GRANT, REVOKE, etc.).
- The most recent SQL standard is known as SQL:2016. Major DBMS vendors support most features in the core part of this standard although the lack of independent conformance testing hinders strict conformance with the standard.
- SELECT is a complex statement. Chapter 4 covered SELECT statements with the format:

  `SELECT` <list of column and column expressions>
  `FROM` <list of tables and join operations>
  `WHERE` <row conditions connected by AND, OR, and NOT>
  `GROUP BY` <list of columns>
  `HAVING` <group conditions connected by AND, OR, and NOT>
  `ORDER BY` <list of sorting specifications>
- Use the standard comparison operators to select rows:

```
SELECT StdFirstName, StdLastName, StdCity, StdGPA
 FROM Student
 WHERE StdGPA >= 3.7
```

- Inexact matching is done with the LIKE operator and pattern-matching characters:

  Oracle and SQL:2016

```
SELECT CourseNo, CrsDesc
 FROM Course
 WHERE CourseNo LIKE 'IS4%'
```

  Access

```
SELECT CourseNo, CrsDesc
 FROM Course
 WHERE CourseNo LIKE 'IS4*'
```

- Use BETWEEN … AND to compare dates:

  Oracle
  ```
  SELECT FacFirstName, FacLastName, FacHireDate
   FROM Faculty
   WHERE FacHireDate BETWEEN '1-Jan-2008' AND '31-Dec-2009'
  ```
  Access:
  ```
  SELECT FacFirstName, FacLastName, FacHireDate
   FROM Faculty
   WHERE FacHireDate BETWEEN #1/1/2008# AND #12/31/2009#
  ```
- Use expressions in the SELECT column list and WHERE clause:

  Oracle
  ```
  SELECT FacFirstName, FacLastName, FacCity,
         FacSalary*1.1 AS InflatedSalary, FacHireDate
   FROM Faculty
   WHERE to_number(to_char(FacHireDate, 'YYYY' ) ) > 2008
  ```
  Access
  ```
  SELECT FacFirstName, FacLastName, FacCity,
         FacSalary*1.1 AS InflatedSalary, FacHireDate
   FROM Faculty
   WHERE year(FacHireDate) > 2008
  ```
- Test for null values:
  ```
  SELECT OfferNo, CourseNo
   FROM Offering
   WHERE FacNo IS NULL AND OffTerm = 'SUMMER'
     AND OffYear = 2017
  ```
- Create complex logical expressions with AND and OR:
  ```
  SELECT OfferNo, CourseNo, FacNo
   FROM Offering
   WHERE (OffTerm = 'FALL' AND OffYear = 2016)
      OR (OffTerm = 'WINTER' AND OffYear = 2017)
  ```
- Sort results with the ORDER BY clause:
  ```
  SELECT StdGPA, StdFirstName, StdLastName, StdCity,
         StdState
   FROM Student
   WHERE StdClass = 'JR'
   ORDER BY StdGPA
  ```
- Eliminate duplicates with the DISTINCT keyword:
  ```
  SELECT DISTINCT FacCity, FacState
   FROM Faculty
  ```
- Qualify column names in join queries:
  ```
  SELECT Course.CourseNo, CrsDesc
   FROM Offering, Course
   WHERE OffTerm = 'SPRING' AND OffYear = 2017
     AND Course.CourseNo = Offering.CourseNo
  ```
- Use the GROUP BY clause to summarize rows:
  ```
  SELECT StdMajor, AVG(StdGPA) AS AvgGpa
   FROM Student
   GROUP BY StdMajor
  ```
- GROUP BY must precede HAVING:
  ```
  SELECT StdMajor, AVG(StdGPA) AS AvgGpa
   FROM Student
   GROUP BY StdMajor
   HAVING AVG(StdGPA) > 3.1
  ```

- Use the WHERE clause to test row conditions and the HAVING clause to test group conditions:

```
SELECT StdMajor, AVG(StdGPA) AS AvgGpa
 FROM Student
 WHERE StdClass IN ('JR', 'SR')
 GROUP BY StdMajor
 HAVING AVG(StdGPA) > 3.1
```

- Difference betwen COUNT(*) and COUNT(DISTINCT column) –– not supported by Access:

```
SELECT OffYear, COUNT(*) AS NumOfferings,
        COUNT(DISTINCT CourseNo) AS NumCourses
 FROM Offering
 GROUP BY OffYear
```

- Conceptual evaluation process lessons: use small sample tables, GROUP BY occurs after WHERE, and only one grouping per SELECT statement.
- Query formulation questions: what tables?, how combined?, and row or group output?
- Joining more than two tables with the cross product and join operator styles:

```
SELECT OfferNo, Offering.CourseNo, CrsUnits, OffDays,
        OffLocation, OffTime
 FROM Faculty, Course, Offering
 WHERE Faculty.FacNo = Offering.FacNo
   AND Offering.CourseNo = Course.CourseNo
   AND OffYear = 2016 AND OffTerm = 'FALL'
   AND FacFirstName = 'LEONARD'
   AND FacLastName = 'VINCE'

SELECT OfferNo, Offering.CourseNo, CrsUnits, OffDays,
       OffLocation, OffTime
 FROM ( Faculty INNER JOIN Offering
     ON Faculty.FacNo = Offering.FacNo )
   INNER JOIN Course
     ON Offering.CourseNo = Course.CourseNo
 WHERE OffYear = 2016 AND OffTerm = 'FALL'
   AND FacFirstName = 'LEONARD'
   AND FacLastName = 'VINCE'
```

- Self-joins:

```
SELECT Subr.FacNo, Subr.FacLastName, Subr.FacSalary,
        Supr.FacNo, Supr.FacLastName, Supr.FacSalary
 FROM Faculty Subr, Faculty Supr
 WHERE Subr.FacSupervisor = Supr.FacNo
   AND Subr.FacSalary > Supr.FacSalary
```

- Combine joins and grouping:

```
SELECT CourseNo, Enrollment.OfferNo,
        COUNT(*) AS NumStudents
 FROM Offering, Enrollment
 WHERE Offering.OfferNo = Enrollment.OfferNo
   AND OffYear = 2017 AND OffTerm = 'SPRING'
 GROUP BY Enrollment.OfferNo, CourseNo
```

- Traditional set operators and union compatibility:

```
SELECT FacNo AS PerNo, FacLastName AS LastName
        FacCity AS City, FacState AS State
 FROM Faculty
   UNION
```

```
SELECT StdNo AS PerNo, StdLastName AS LastName,
       StdCity AS City, StdState AS State
 FROM Student
```

- Use the INSERT statement to add one or more rows:

```
INSERT INTO Student
 (StdNo, StdFirstName, StdLastName, StdCity, StdState,
  StdClass, StdMajor, StdGPA)
 VALUES ('999999999', 'Joe', 'Student', 'Seatac', 'WA',
         'FR', 'IS', 0.0)
```

- Use the UPDATE statement to change columns in one or more rows:

```
UPDATE Faculty
 SET FacSalary = FacSalary * 1.1
 WHERE FacDept = 'MS'
```

- Use the DELETE statement to remove one or more rows:

```
DELETE FROM Student
 WHERE StdMajor = 'IS' AND StdClass = 'SR'
```

- Use a join operation inside an UPDATE statement (Access only):

```
UPDATE Offering INNER JOIN Faculty
         ON Offering.FacNo = Faculty.FacNo
 SET OffLocation = 'BLM412'
 WHERE OffYear = 2016 AND FacFirstName = 'LEONARD'
   AND FacLastName = 'VINCE'
```

- Use a join operation inside a DELETE statement (Access only):

```
DELETE Offering.*
 FROM Offering INNER JOIN Faculty
   ON Offering.FacNo = Faculty.FacNo
 WHERE FacFirstName = 'LEONARD' AND FacLastName = 'VINCE'
```

- Syntax errors with sometimes confusing error messages and no execution: missing table, unqualified column name, misspelled keyword, row condition in HAVING clause, missing column in GROUP BY clause, and aggregate function in a WHERE condition
- Redundancy errors with correct result rows but extra resource consumption: extra table and unneeded GROUP BY clause
- Semantic errors with incorrect rows in the result and sometimes excessive resource consumption: missing row condition, missing parentheses, incorrect condition, and missing join condition
- Poor coding practices involving misalignment of clauses, incompatible constants in conditions, LIKE operator without pattern matching characters, and LIKE operator in conditions on date columns

QUESTIONS

1. Why do some information systems professionals pronounce SQL as "sequel"?
2. Why are the manipulation statements of SQL more widely used than the definition and control statements?
3. Briefly describe the organization and acceptance of SQL:2016.
4. Why is conformance testing important for the SQL standard?
5. In general, what is the state of conformance among major DBMS vendors for the SQL:2016 standard?
6. What is stand-alone SQL?
7. What is embedded SQL?

8. What is an expression in the context of database languages?
9. From the examples and the discussion in Chapter 4, what parts of the SELECT statement are not supported by all DBMSs?
10. Recite the rule about the GROUP BY and HAVING clauses.
11. Recite the rule about columns in SELECT when a GROUP BY clause is used.
12. How does a row condition differ from a group condition?
13. Why should row conditions be placed in the WHERE clause rather than the HAVING clause?
14. Why are most DBMSs not case sensitive when matching on string conditions?
15. Explain how working with sample tables can provide insight about difficult problems.
16. When working with date columns, why is it necessary to refer to documentation of your DBMS?
17. How do exact and inexact matching differ in SQL?
18. How do you know when the output of a query relates to groups of rows as opposed to individual rows?
19. What tables belong in the FROM statement?
20. Explain the cross product style for join operations.
21. Explain the join operator style for join operations.
22. Discuss the pros and cons of the cross product versus the join operator styles. Do you need to know both the cross product and the join operator styles?
23. What is a self-join? When is a self-join useful?
24. Provide a SELECT statement example in which a table is needed even though the table does not provide conditions to test or columns to show in the result.
25. What is the requirement when using the traditional set operators in a SELECT statement?
26. When combining joins and grouping, what conceptually occurs first, join operations or grouping?
27. How many times does grouping occur in a SELECT statement?
28. Why is the SELECT statement more widely used than the modification statements INSERT, UPDATE, and DELETE?
29. Provide an example of an INSERT statement that can insert multiple rows.
30. What is the relationship between the DELETE statement and the rules about deleting referenced rows?
31. What is the relationship between the UPDATE statement and the rules about updating the primary key of referenced rows?
32. How does COUNT(*) differ from COUNT(ColumnName)?
33. How does COUNT(DISTINCT ColumnName) differ from COUNT(ColumnName)?
34. When mixing AND and OR in a logical expression, why is it a good idea to use parentheses?
35. What are the most important lessons about the conceptual evaluation process?
36. What are the mental steps involved in query formulation?
37. What kind of join queries often have duplicates in the result?
38. What mental steps in the query formulation process are addressed by the conceptual evaluation process and critical questions?
39. In the SQL SELECT statement, how do you apply the set operators to two tables with only some compatible columns?

40. Why should you avoid mixing the join styles in a SELECT statement?
41. What is the SQL:2016 symbol for matching any single character?
42. What symbols are used by Microsoft Access and Oracle for matching any single character?
43. Provide a brief example to depict the single character pattern matching symbol.
44. Should you use the LIKE operator for conditions involving date columns?
45. What is the default format for date constants in Oracle SQL?
46. In the join operator style, does Oracle require parentheses when multiple join operations are used?
47. In the join operator style, does Access require parentheses when multiple join operations are used?
48. What is the impact of a syntax error in a SELECT statement?
49. What is the impact of a redundancy error in a SELECT statement? Provide an answer using a specific redundancy error.
50. What is the impact of a semantic error in a SELECT statement? Provide an answer using a specific semantic error.
51. Can a result contain the correct rows if a SELECT statement contains a semantic error? Please explain with an example.

PROBLEMS

The problems use the tables of the Order Entry database, an extension of the order entry tables used in the problems of Chapter 3. Table 4-P1 lists the meaning of each table and Figure 4.P1 shows the Access Relationship window. After the relationship diagram, row listings and Oracle CREATE TABLE statements are shown for each table. In addition to the other documentation, here are some notes about the Order Entry Database:

- The primary key of the *OrdLine* table is a combination of *OrdNo* and *ProdNo*.
- The *Employee* table has a self-referencing (unary) relationship to itself through the foreign key, *SupEmpNo*, the employee number of the supervising employee. In the relationship diagram, the table *Employee_1* is a representation of the self-referencing relationship, not a real table.
- The relationship from *OrderTbl* to *OrdLine* cascades deletions and primary key updates of referenced rows. All other relationships restrict deletions and primary key ues of referend rows if related rows exist.

TABLE 4-P1
Tables of the Order Entry Database

Table Name	Description
Customer	List of customers who have placed orders
OrderTbl	Contains the heading part of an order; Internet orders do not have an employee
Employee	List of employees who can take orders
OrdLine	Contains the product detail parts of orders
Product	List of products that may be ordered

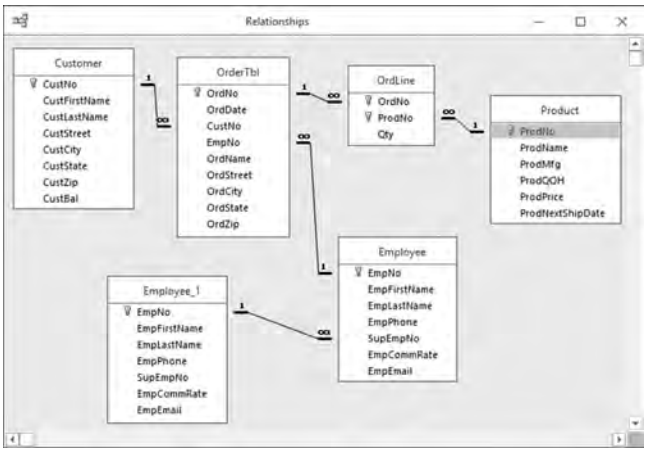


FIGURE 4.P1
Relationship Window for the Order Entry Database

Customer

CustNo	CustFirstName	CustLastName	CustStreet	CustCity	CustState	CustZip	CustBal
C0954327	Sheri	Gordon	336 Hill St.	Littleton	CO	80129-5543	$230.00
C1010398	Jim	Glussman	1432 E. Ravenna	Denver	CO	80111-0033	$200.00
C2388597	Beth	Taylor	2396 Rafter Rd	Seattle	WA	98103-1121	$500.00
C3340959	Betty	Wise	4334 153rd NW	Seattle	WA	98178-3311	$200.00
C3499503	Bob	Mann	1190 Lorraine Cir.	Monroe	WA	98013-1095	$0.00
C8543321	Ron	Thompson	789 122nd St.	Renton	WA	98666-1289	$85.00
C8574932	Wally	Jones	411 Webber Ave.	Seattle	WA	98105-1093	$1,500.00
C8654390	Candy	Kendall	456 Pine St.	Seattle	WA	98105-3345	$50.00
C9128574	Jerry	Wyatt	16212 123rd Ct.	Denver	CO	80222-0022	$100.00
C9403348	Mike	Boren	642 Crest Ave.	Englewood	CO	80113-5431	$0.00
C9432910	Larry	Styles	9825 S. Crest Lane	Bellevue	WA	98104-2211	$250.00
C9543029	Sharon	Johnson	1223 Meyer Way	Fife	WA	98222-1123	$856.00
C9549302	Todd	Hayes	1400 NW 88th	Lynnwood	WA	98036-2244	$0.00
C9857432	Homer	Wells	123 Main St.	Seattle	WA	98105-4322	$500.00
C9865874	Mary	Hill	206 McCaffrey	Littleton	CO	80129-5543	$150.00
C9943201	Harry	Sanders	1280 S. Hill Rd.	Fife	WA	98222-2258	$1,000.00

OrderTbl

OrdNo	OrdDate	CustNo	EmpNo	OrdName	OrdStreet	OrdCity	OrdState	OrdZip
O1116324	01/23/2017	C0954327	E8544399	Sheri Gordon	336 Hill St.	Littleton	CO	80129-5543
O1231231	01/23/2017	C9432910	E9954302	Larry Styles	9825 S. Crest Lane	Bellevue	WA	98104-2211
O1241518	02/10/2017	C9549302		Todd Hayes	1400 NW 88th	Lynnwood	WA	98036-2244
O1455122	01/09/2017	C8574932	E9345771	Wally Jones	411 Webber Ave.	Seattle	WA	98105-1093
O1579999	01/05/2017	C9543029	E8544399	Tom Johnson	1632 Ocean Dr.	Des Moines	WA	98222-1123
O1615141	01/23/2017	C8654390	E8544399	Candy Kendall	456 Pine St.	Seattle	WA	98105-3345
O1656777	02/11/2017	C8543321		Ron Thompson	789 122nd St.	Renton	WA	98666-1289
O2233457	01/12/2017	C2388597	E9884325	Beth Taylor	2396 Rafter Rd	Seattle	WA	98103-1121
O2334661	01/14/2017	C0954327	E1329594	Mrs. Ruth Gordon	233 S. 166th	Seattle	WA	98011

OrdNo	OrdDate	CustNo	EmpNo	OrdName	OrdStreet	OrdCity	OrdState	OrdZip
O3252629	01/23/2017	C9403348	E9954302	Mike Boren	642 Crest Ave.	Englewood	CO	80113-5431
O3331222	01/13/2017	C1010398		Jim Glussman	1432 E. Ravenna	Denver	CO	80111-0033
O3377543	01/15/2017	C9128574	E8843211	Jerry Wyatt	16212 123rd Ct.	Denver	CO	80222-0022
O4714645	01/11/2017	C2388597	E1329594	Beth Taylor	2396 Rafter Rd	Seattle	WA	98103-1121
O5511365	01/22/2017	C3340959	E9884325	Betty White	4334 153rd NW	Seattle	WA	98178-3311
O6565656	01/20/2017	C9865874	E8843211	Mr. Jack Sibley	166 E. 344th	Renton	WA	98006-5543
O7847172	01/23/2017	C9943201		Harry Sanders	1280 S. Hill Rd.	Fife	WA	98222-2258
O7959898	02/19/2017	C8543321	E8544399	Ron Thompson	789 122nd St.	Renton	WA	98666-1289
O7989497	01/16/2017	C3499503	E9345771	Bob Mann	1190 Lorraine Cir.	Monroe	WA	98013-1095
O8979495	01/23/2017	C9865874		HelenSibley	206 McCaffrey	Renton	WA	98006-5543
O9919699	02/11/2017	C9857432	E9954302	Homer Wells	123 Main St.	Seattle	WA	98105-4322

Employee

EmpNo	EmpFirstName	EmpLastName	EmpPhone	EmpEMail	SupEmpNo	EmpCommRate
E1329594	Landi	Santos	(303) 789-1234	LSantos@bigco.com	E8843211	0.02
E8544399	Joe	Jenkins	(303) 221-9875	JJenkins@bigco.com	E8843211	0.02
E8843211	Amy	Tang	(303) 556-4321	ATang@bigco.com	E9884325	0.04
E9345771	Colin	White	(303) 221-4453	CWhite@bigco.com	E9884325	0.04
E9884325	Thomas	Johnson	(303) 556-9987	TJohnson@bigco.com		0.05
E9954302	Mary	Hill	(303) 556-9871	MHill@bigco.com	E8843211	0.02
E9973110	Theresa	Beck	(720) 320-2234	TBeck@bigco.com	E9884325	

Product

ProdNo	ProdName	ProdMfg	ProdQOH	ProdPrice	ProdNextShipDate
P0036566	17 inch Color Monitor	ColorMeg, Inc.	12	$169.00	2/20/2017
P0036577	19 inch Color Monitor	ColorMeg, Inc.	10	$319.00	2/20/2017
P1114590	R3000 Color Laser Printer	Connex	5	$699.00	1/22/2017
P1412138	10 Foot Printer Cable	Ethlite	100	$12.00	
P1445671	8-Outlet Surge Protector	Intersafe	33	$14.99	
P1556678	CVP Ink Jet Color Printer	Connex	8	$99.00	1/22/2017
P3455443	Color Ink Jet Cartridge	Connex	24	$38.00	1/22/2017
P4200344	36-Bit Color Scanner	UV Components	16	$199.99	1/29/2017
P6677900	Black Ink Jet Cartridge	Connex	44	$25.69	
P9995676	Battery Back-up System	Cybercx	12	$89.00	2/1/2017

OrdLine

OrdNo	ProdNo	Qty
O1116324	P1445671	1
O1231231	P0036566	1
O1231231	P1445671	1

OrdNo	ProdNo	Qty
O1241518	P0036577	1
O1455122	P4200344	1
O1579999	P1556678	1
O1579999	P6677900	1
O1579999	P9995676	1
O1615141	P0036566	1
O1615141	P1445671	1
O1615141	P4200344	1
O1656777	P1445671	1
O1656777	P1556678	1
O2233457	P0036577	1
O2233457	P1445671	1
O2334661	P0036566	1
O2334661	P1412138	1
O2334661	P1556678	1
O3252629	P4200344	1
O3252629	P9995676	1
O3331222	P1412138	1
O3331222	P1556678	1
O3331222	P3455443	1
O3377543	P1445671	1
O3377543	P9995676	1
O4714645	P0036566	1
O4714645	P9995676	1
O5511365	P1412138	1
O5511365	P1445671	1
O5511365	P1556678	1
O5511365	P3455443	1
O5511365	P6677900	1
O6565656	P0036566	10
O7847172	P1556678	1
O7847172	P6677900	1
O7959898	P1412138	5
O7959898	P1556678	5
O7959898	P3455443	5
O7959898	P6677900	5
O7989497	P1114590	2
O7989497	P1412138	2
O7989497	P1445671	3
O8979495	P1114590	1
O8979495	P1412138	1
O8979495	P1445671	1
O9919699	P0036577	1
O9919699	P1114590	1
O9919699	P4200344	1

```
CREATE TABLE Customer
( CustNo           CHAR(8),
  CustFirstName    VARCHAR2(20) CONSTRAINT CustFirstNameRequired NOT NULL,
  CustLastName     VARCHAR2(30) CONSTRAINT CustLastNameRequired NOT NULL,
  CustStreet       VARCHAR2(50),
  CustCity         VARCHAR2(30),
  CustState        CHAR(2),
  CustZip          CHAR(10),
  CustBal          DECIMAL(12,2) DEFAULT 0,
 CONSTRAINT PKCustomer PRIMARY KEY (CustNo)  )
```

```
CREATE TABLE OrderTbl
(     OrdNo            CHAR(8),
      OrdDate          DATE    CONSTRAINT OrdDateRequired NOT NULL,
      CustNo           CHAR(8) CONSTRAINT CustNoRequired NOT NULL,
      EmpNo            CHAR(8),
      OrdName          VARCHAR2(50),
      OrdStreet        VARCHAR2(50),
      OrdCity          VARCHAR2(30),
      OrdState         CHAR(2),
      OrdZip           CHAR(10),
CONSTRAINT PKOrderTbl PRIMARY KEY (OrdNo) ,
CONSTRAINT FKCustNo FOREIGN KEY (CustNo) REFERENCES Customer,
CONSTRAINT FKEmpNo FOREIGN KEY (EmpNo) REFERENCES Employee  )
```

```
CREATE TABLE OrdLine
(     OrdNo            CHAR(8),
      ProdNo           CHAR(8),
      Qty              INTEGER DEFAULT 1,
CONSTRAINT PKOrdLine PRIMARY KEY (OrdNo, ProdNo),
CONSTRAINT FKOrdNo FOREIGN KEY (OrdNo) REFERENCES OrderTbl
    ON DELETE CASCADE,
CONSTRAINT FKProdNo FOREIGN KEY (ProdNo) REFERENCES Product )
```

```
CREATE TABLE Employee
(     EmpNo          CHAR(8),
      EmpFirstName   VARCHAR2(20) CONSTRAINT EmpFirstNameRequired NOT NULL,
      EmpLastName    VARCHAR2(30) CONSTRAINT EmpLastNameRequired NOT NULL,
      EmpPhone       CHAR(15),
      EmpEMail       VARCHAR(50) CONSTRAINT EmpEMailRequired NOT NULL,
      SupEmpNo       CHAR(8),
      EmpCommRate    DECIMAL(3,3),
CONSTRAINT PKEmployee PRIMARY KEY (EmpNo),
CONSTRAINT UNIQUEEMail UNIQUE(EmpEMail),
CONSTRAINT FKSupEmpNo FOREIGN KEY (SupEmpNo) REFERENCES Employee )
```

```
CREATE TABLE Product
(     ProdNo            CHAR(8),
      ProdName          VARCHAR2(50) CONSTRAINT ProdNameRequired NOT NULL,
      ProdMfg           VARCHAR2(20) CONSTRAINT ProdMfgRequired NOT NULL,
      ProdQOH           INTEGER DEFAULT 0,
      ProdPrice         DECIMAL(12,2) DEFAULT 0,
      ProdNextShipDate  DATE,
 CONSTRAINT PKProduct PRIMARY KEY (ProdNo)   )
```

Part 1: SELECT

1. List the customer number, the name (first and last), and the balance of customers.
2. List the customer number, the name (first and last), and the balance of customers who reside in Colorado (CustState is CO).
3. List all columns of the *Product* table for products costing more than $50. Order the result by product manufacturer (*ProdMfg*) and product name.
4. List the order number, order date, and shipping name (*OrdName*) of orders sent to addresses in Denver or Englewood.
5. List the customer number, the name (first and last), the city, and the balance of customers who reside in Denver with a balance greater than $150 or who reside in Seattle with a balance greater than $300.
6. List the cities and states where orders have been placed. Remove duplicates from the result.
7. List all columns of the *OrderTbl* table for Internet orders placed in January 2017. An Internet order does not have an associated employee.
8. List all columns of the *OrderTbl* table for phone orders placed in February 2017. A phone order has an associated employee.

9. List all columns of the *Product* table that contain the words Ink Jet in the product name.
10. List the order number, order date, and customer number of orders placed after January 23, 2017, shipped to Washington recipients.
11. List the order number, order date, customer number, and customer name (first and last) of orders placed in January 2017 sent to Colorado recipients.
12. List the order number, order date, customer number, and customer name (first and last) of orders placed in January 2017 placed by Colorado customers (*CustState*) but sent to Washington recipients (*OrdState*).
13. List the customer number, name (first and last), and balance of Washington customers who have placed one or more orders in February 2017. Remove duplicate rows from the result.
14. List the order number, order date, customer number, customer name (first and last), employee number, and employee name (first and last) of January 2017 orders placed by Colorado customers.
15. List the employee number, name (first and last), and phone of employees who have taken orders in January 2017 from customers with balances greater than $300. Remove duplicate rows in the result.
16. List the product number, name, and price of products ordered by customer number C0954327 in January 2017. Remove duplicate products in the result.
17. List the customer number, name (first and last), order number, order date, employee number, employee name (first and last), product number, product name, and order cost `(OrdLine.Qty * ProdPrice)` for products ordered on January 23, 2017, in which the order cost exceeds $150.
18. List the average balance of customers by city. Include only customers residing in Washington state (WA).
19. List the average balance of customers by city and short zip code (the first five digits of the zip code). Include only customers residing in Washington State (WA). In Microsoft Access, the expression `left(CustZip, 5)` returns the first five digits of the zip code. In Oracle, the expression `substr(CustZip, 1, 5)` returns the first five digits.
20. List the average balance and number of customers by city. Only include customers residing in Washington State (WA). Eliminate cities in the result with less than two customers.
21. List the number of unique short zip codes and average customer balance by city. Only include customers residing in Washington State (WA). Eliminate cities in the result in which the average balance is less than $100. In Microsoft Access, the expression `left(CustZip, 5)` returns the first five digits of the zip code. In Oracle, the expression `substr(CustZip, 1, 5)` returns the first five digits. (Note: this problem requires two SELECT statements in Access SQL or a nested query in the FROM clause, see Chapter 9).
22. List the order number and total amount for orders placed on January 23, 2017. The total amount of an order is the sum of the quantity times the product price of each product on the order.
23. List the order number, order date, customer name (first and last), and total amount for orders placed on January 23, 2017. The total amount of an order is the sum of the quantity times the product price of each product on the order.
24. List the customer number, customer name (first and last), the sum of the quantity of products ordered, and the total order amount (sum of the product price times the quantity) for orders placed in January 2008. Only include products in which the product name contains the string Ink Jet or Laser. Only

include customers who have ordered more than two Ink Jet or Laser products in January 2017.

25. List the product number, product name, sum of the quantity of products ordered, and total order amount (sum of the product price times the quantity) for orders placed in January 2017. Only include products that have more than five products ordered in January 2017. Sort the result in descending order of the total amount.
26. List the order number, the order date, the customer number, the customer name (first and last), the customer state, and the shipping state (*OrdState*) in which the customer state differs from the shipping state.
27. List the employee number, the employee name (first and last), the commission rate, the supervising employee name (first and last), and the commission rate of the supervisor.
28. List the employee number, the employee name (first and last), and total amount of commissions on orders taken in January 2017. The amount of a commission is the sum of the dollar amount of products ordered times the commission rate of the employee.
29. List the union of customers and order recipients. Include the name, street, city, state, and zip in the result. You need to use the concatenation function to combine the first and last names so that they can be compared to the order recipient name. In Access SQL, the **&** symbol is the concatenation function. In Oracle SQL, the **||** symbol is the concatenation function.
30. List the first and last name of customers who have the same name (first and last) as an employee.
31. List the employee number and the name (first and last) of second-level subordinates (subordinates of subordinates) of the employee named Thomas Johnson.
32. List the employee number and the name (first and last) of the first- and second-level subordinates of the employee named Thomas Johnson. To distinguish the level of subordinates, include a computed column with the subordinate level (1 or 2).
33. Using a mix of the join operator and the cross product styles, list the names (first and last) of customers who have placed orders taken by Amy Tang. Remove duplicate rows in the result. Note that the join operator style is supported only in Oracle versions 9i and beyond.
34. Using the join operator style, list the product name and the price of all products ordered by Beth Taylor in January 2017. Remove duplicate rows from the result.
35. For Colorado customers, compute the number of orders placed in January 2017. The result should include the customer number, last name, and number of orders placed in January 2017.
36. For Colorado customers, compute the number of orders placed in January 2017 in which the orders contain products made by Connex. The result should include the customer number, last name, and number of orders placed in January 2017.
37. For each employee with a commission rate of less than 0.04, compute the number of orders taken in January 2017. The result should include the employee number, employee last name, and number of orders taken.
38. For each employee with a commission rate greater than 0.03, compute the total commission earned from orders taken in January 2017. The total commission earned is the total order amount times the commission rate. The result should include the employee number, employee last name, and total commission earned.

39. List the total amount of all orders by month in 2017. The result should include the month and the total amount of all orders in each month. The total amount of an individual order is the sum of the quantity times the product price of each product in the order. In Access, the month number can be extracted by the **Month** function with a date as the argument. You can display the month name using the **MonthName** function applied to a month number. In Oracle, the function `to_char(OrdDate, 'MON')` extracts the three-digit month abbreviation from *OrdDate*.
40. List the total commission earned by each employee in each month of 2017. The result should include the month, employee number, employee last name, and the total commission amount earned in that month. The amount of a commission for an individual employee is the sum of the dollar amount of products ordered times the commission rate of the employee. Sort the result by the month in ascending month number and the total commission amount in descending order. In Access, the month number can be extracted by the **Month** function with a date as the argument. You can display the month name using the **MonthName** function applied to a month number. In Oracle, the function `to_char(OrdDate, 'MON')` extracts the three-digit month abbreviation from *OrdDate*.
41. List the product number, name, price, and total quantity ordered of products ordered by Colorado customers in January 2017.
42. List the product number, name, price, and total order value (sum of price times quantity) of products ordered by Colorado customers in January 2017 in phone orders. A phone order has an employee associated with the order.
43. List the product number, name, price, and total order value (sum of price times quantity) of products ordered by Colorado customers in January 2017 in web orders. A web order does not have an employee associated with the order.
44. Combine the results of problems 42 and 43 into one query result. Each row in the result should indicate if the row involves a phone or web order. (Hint: you need to use the UNION operator).
45. List the order number, order date, customer name (first and last), product number, product name, and next shipment date of orders with an order date within 14 days (absolute value of day difference) of the next shipment date of the product on the order. You should use the **datediff** function in Access and the subtraction operator in Oracle to find the difference in days between two dates. You should use the **abs** function in Access and Oracle to calculate the absolute value of a number.
46. List the employee number and the name (first and last) of the first-level superior (direct boss) of the employee named Joe Jenkins.
47. List the employee number and the name (first and last) of first-level superior (direct boss) and second-level superior (boss of direct boss) of the employee named Joe Jenkins. To distinguish the level of subordinates, include a computed column with the superior level (1 or 2).
48. Can you extend problem 47 to list all superiors (direct and indirect) of the employee named Joe Jenkins? Explain your reasoning.
49. Identify errors in the following SQL statement. For each error, you should indicate the error type as specified in Table 4-21. Correct the statement and document its purpose.

```
SELECT ProdName, ProdPrice, SUM(Qty) AS ProdCount
  FROM OrderTbl, OrdLine, Product, Customer
  WHERE ProdPrice > 50
        AND OrderTbl.OrdNo = OrdLine.OrdNo
        AND OrdLine.ProdNo = Product.ProdNo
```

```
        AND Customer.CustNo = OrderTbl.CustNo
        ANE CustState = 'CO'
  GROUP BY ProdName
```

50. Identify errors in the following SQL statement. For each error, you should indicate the error type as specified in Table 4-21. Correct the statement and document its purpose.

```
SELECT CustState, AVG(CustBal) AS AvgBal, COUNT(*) AS
NumCustomers
  FROM Customer
  WHERE CustBal > 100
```

51. The following statement should find customers who have a balance greater than $80 and live in either Denver or Seattle. Correct the statement and explain the error(s) in the statement.

```
SELECT CustNo, CustFirstName, CustLastName, CustCity,
CustBal
  FROM Customer
  WHERE CustCity = 'Seattle' OR CustCity = 'Denver'
          AND CustBal > 80
```

52. Identify errors in the following SQL statement. For each error, you should indicate the error type as specified in Table 4-21 and the impact of the error. Correct the statement and document its purpose.

```
SELECT Customer.CustNo, CustFirstName, CustLastName,
OrderTbl.OrdNo, OrdDate,
                Product.ProdNo, ProdName, ProdPrice
  FROM OrderTbl, OrdLine, Product, Customer
  WHERE CustState =  'WA' AND ProdPrice > 100
        AND OrderTbl.OrdNo = OrdLine.OrdNo
        AND OrderTbl.CustNo = Customer.CustNo
```

53. The following statement should retrieve details about employees who took orders from customers residing in Washington (WA) state with a balance greater than $300. Identify errors in the following SQL statement. For each error, you should indicate the error type as specified in Table 4-21 and the impact of the error. Correct the statement and indicate the impact of the DISTINCT clause.

```
SELECT DISTINCT Employee.EmpNo, EmpFirstName, EmpLastName,
EmpPhone
  FROM OrderTbl, Customer, Employee, OrdLine
  WHERE CustBal > 300 AND CustState =  'WA'
        AND OrderTbl.CustNo = Customer.CustNo
        AND OrderTbl.EmpNo = Employee.EmpNo
        AND OrdLine.OrdNo = OrderTbl.OrdNo
```

54. Identify poor coding practices in the following statement and rewrite the statement with good coding practices. The condition on order date should test for orders in January 2017.

```
SELECT DISTINCT Employee.EmpNo, EmpFirstName,
EmpLastName, EmpPhone FROM OrderTbl INNER JOIN Customer ON
OrderTbl.CustNo = Customer.CustNo, Employee
  WHERE CustBal > '300' AND OrdDate LIKE '1/*/2017'
AND OrderTbl.EmpNo
= Employee.EmpNo
```

Part 2: INSERT, UPDATE, and DELETE statements

1. Insert yourself as a new row in the *Customer* table.
2. Insert your roommate, best friend, or significant other as a new row in the *Employee* table.
3. Insert a new *OrderTbl* row with you as the customer, the person from problem 2 (Part 2) as the employee, and your choice of values for the other columns of the *OrderTbl* table.
4. Insert two rows in the *OrdLine* table corresponding to the *OrderTbl* row inserted in problem 3 (Part 2).
5. Increase the price by 10 percent of products containing the words Ink Jet.
6. Change the address (street, city, and zip) of the new row inserted in problem 1 (Part 2).
7. Identify an order that respects the rules about deleting referenced rows to delete the rows inserted in problems 1 to 4 (part 2).
8. Delete the new row(s) of the table listed first in the order for problem 7 (Part 2).
9. Delete the new row(s) of the table listed second in the order for problem 7 (Part 2).
10. Delete the new row(s) of the remaining tables listed in the order for problem 7 (Part 2).

REFERENCES FOR FURTHER STUDY

SQL tutorials can be found at SQLCourse.com, SQLCourse2.com, W3C's SQL school (www.w3schools.com/sql/), and the SQL Zoo (sqlzoo.net). For product-specific SQL advice, the sqlblog.com site feature forums about a number of DBMSs including Microsoft SQL Server and open source products. The Database Journal (www.databasejournal.com) provides articles, tutorials, and resources about many DBMS products. Oracle documentation can be found at the Oracle Technet site (www.oracle.com/technetwork). The Mimer Developer website has validators (developer.mimer.se/validator) for the SQL standard as aids to writing portable SQL statements.

part three

Data Modeling

The chapters in Part 3 cover data modeling using the entity relationship model to provide skills for conceptual database design. Chapter 5 presents the Crow's Foot notation of the entity relationship model and explains diagram rules to prevent common diagram errors. Chapter 6 emphasizes the practice of data modeling on narrative problems and presents rules to convert entity relationship diagrams (ERDs) into relational tables. Chapter 6 explains design transformations and common design errors to sharpen data modeling skills.

chapter

5

Understanding Entity Relationship Diagrams

Learning Objectives

This chapter explains the notation of entity relationship diagrams as a prerequisite to using entity relationship diagrams in the database development process. After this chapter, the student should have acquired the following knowledge and skills:

- Know the symbols and vocabulary of the Crow's Foot notation for entity relationship diagrams
- Use cardinality symbols to represent 1-1, 1-M, and M-N relationships
- Compare the Crow's Foot notation to the representation of relational tables
- Understand important relationship patterns
- Use generalization hierarchies to represent similar entity types
- Correct notational errors in an entity relationship diagram
- Understand the representation of business rules in an entity relationship diagram
- Appreciate the diversity of notation for entity relationship diagrams

OVERVIEW

Chapter 2 provided a broad presentation about the database development process. You learned about the relationship between database development and information systems development, the phases of database development, and the kinds of skills you need to master. This chapter presents the Crow's Foot notation for entity relationship diagrams to provide a foundation for development of your data modeling skills. To extend your database design skills, Chapter 6 describes the process of using entity relationship diagrams to develop data models for business databases.

To become a good data modeler, you need to understand the notation in entity relationship diagrams and apply the notation on problems of increasing complexity. To help you master the notation, this chapter presents the symbols used in entity relationship diagrams and compares entity relationship diagrams to relational database diagrams that you have seen in previous chapters. This chapter then probes deeper into relationships, the most distinguishing part of entity relationship diagrams. You will learn about identification dependency, relationship patterns, and equivalence between two kinds of relationships. Finally, you will learn to represent similarities among entity types using generalization hierarchies.

The next part of the chapter presents business rule representation and diagram rules to deepen your understanding of the Crow's Foot notation. To provide an organizational focus, this chapter presents formal and informal representation of business rules in an entity relationship diagram. To help you avoid common

notation errors, this chapter presents consistency and completeness rules and depicts their usage in examples.

Because of the plethora of entity relationship notations, you may not have the opportunity to use the Crow's Foot notation exactly as shown in Chapters 5 and 6. To prepare you for understanding other notations, the chapter concludes with a presentation of database diagram variations in commercial CASE tools as well as the Class Diagram notation of the Unified Modeling Notation, one of the popular alternatives to the Entity Relationship Model.

This chapter provides the basic skills of data modeling to enable you to understand the notation of entity relationship diagrams. To apply data modeling as part of the database development process, you should study Chapter 6 on developing data models for business databases. Chapter 6 emphasizes the problem-solving skills of generating alternative designs, mapping a problem statement to an entity relationship diagram, and justifying design decisions. With the background provided in both chapters, you will be prepared to perform data modeling on case studies and databases for moderate-size organizations.

5.1 INTRODUCTION TO ENTITY RELATIONSHIP DIAGRAMS

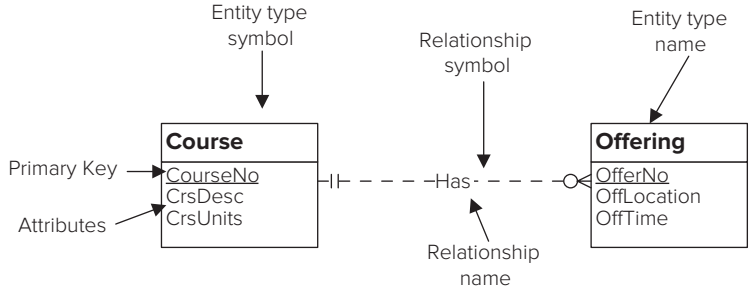
Gaining an initial understanding of entity relationship diagrams (ERDs) requires careful study. This section introduces the Crow's Foot notation[1] for ERDs, a popular notation supported by many CASE tools. To get started, this section begins with the basic symbols of entity types, relationships, and attributes. This section then explains cardinalities and their appearance in the Crow's Foot notation. This section concludes by comparing the Crow's Foot notation to relational database diagrams. If you are covering data modeling before relational databases, you may want to skip the last part of this section.

5.1.1 Basic Symbols

Entity Type
a collection of entities (persons, places, events, or things) of interest represented by a rectangle in an entity relationship diagram.

ERDs have three basic elements: entity types, relationships, and attributes. **Entity types** are collections of things of interest (entities) in an application. Entity types represent collections of physical things such as books, people, and places, as well as events such as payments. An entity is a member or instance of an entity type. Entities are uniquely identified to allow tracking across business processes. For example, customers have a unique identification to support order processing, shipment, and product warranty processes. In the Crow's Foot notation as well as most other notations, rectangles denote entity types. In Figure 5.1, the *Course* entity type represents the set of courses in the database.

FIGURE 5.1
Entity Relationship Diagram Illustrating Basic Symbols



[1] Gordon Everest proposed the Crow's Foot notation in a 1976 paper. Several development methodologies enhanced the notation in the 1980s. Most CASE tools support some variation of the Crow's Foot notation.

Attributes are properties of entity types or relationships. An entity type should have a primary key as well as other descriptive attributes. Attributes are shown inside an entity type rectangle. If an entity type contains many attributes, the attributes can be optionally suppressed. Some ERD drawing tools provide alternative views to show or suppress attributes. Underlining indicates that the attribute(s) serves as the primary key of the entity type.

Attribute
a property of an entity type or relationship. Each attribute has a data type that defines the kind of values and permissible operations on the attribute.

Relationships are named associations among entity types. In the Crow's Foot notation, relationship names appear on the line connecting the entity types involved in the relationship. In Figure 5.1, the *Has* relationship shows that the *Course* and *Offering* entity types are directly related. Relationships store associations in both directions. For example, the *Has* relationship shows the offerings for a given course and the associated course for a given offering. The *Has* relationship is binary because it involves two entity types. Section 5.2 presents examples of more complex relationships involving only one distinct entity type (unary relationships) and more than two entity types (M-way relationships).

Relationship
a named association among entity types. A relationship represents a two-way or bidirectional association among entities. Most relationships involve two distinct entity types.

Informally, ERDs have a natural language correspondence. Entity types can correspond to common nouns and relationships to transitive verbs.[2] In this sense, one can read an entity relationship diagram as a collection of sentences. For example, the ERD in Figure 5.1 can be read as "course has offerings." Note that there is an implied direction in each relationship. In the other direction, one could write, "offering is given for a course." If practical, it is a good idea to use active rather than passive verbs for relationships. Therefore, *Has* is preferred as the relationship name. You should use the natural language correspondence as a guide rather than as a strict rule. For large ERDs, you will not always find a good natural language correspondence for all parts of a diagram.

5.1.2 Relationship Cardinality

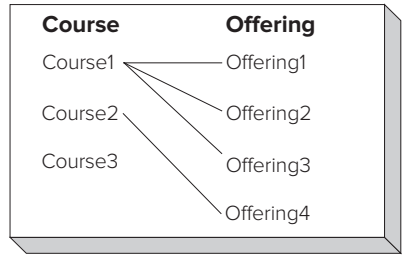
Cardinalities constrain the number of objects that participate in a relationship. To depict the meaning of cardinalities, an instance diagram is useful. Figure 5.2 shows a set of courses ({Course1, Course2, Course3}), a set of offerings ({Offering1, Offering2, Offering3, Offering4}), and connections between the two sets. In Figure 5.2, Course1 is related to Offering1, Offering2, and Offering3, Course2 is related to Offering4, and Course3 is not related to any *Offering* entities. Likewise, Offering1 is related to Course1, Offering2 is related to Course1, Offering3 is related to Course1, and Offering4 is related to Course2. From this instance diagram, we might conclude that each offering is related to exactly one course. In the other direction, each course is related to 0 or more offerings.

Cardinality
a constraint on the number of entities that participate in a relationship. In an ERD, the minimum and maximum cardinalities are specified for both directions of a relationship.

Crow's Foot Representation of Cardinalities The Crow's Foot notation uses three symbols to represent cardinalities. The Crow's Foot symbol (two angled lines and one straight line) denotes many (zero or more) related entities. In Figure 5.3, the Crow's Foot symbol near the *Offering* entity type means that a course can be related to many offerings. The circle means a cardinality of zero, while a line perpendicular to the relationship line denotes a cardinality of one.

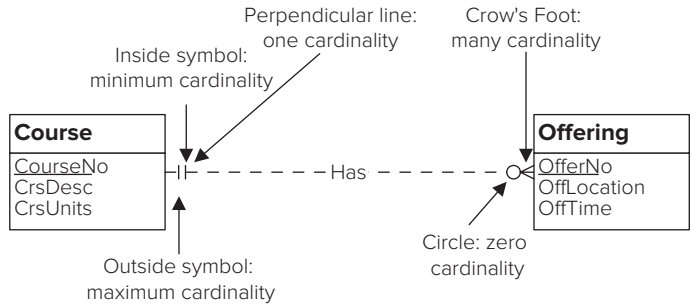
To depict minimum and maximum cardinalities, the cardinality symbols are placed adjacent to each entity type in a relationship. The minimum cardinality symbol appears toward the relationship name while the maximum cardinality symbol appears toward the entity type. In Figure 5.3, a course is related to a minimum of zero offerings (circle in the inside position) and a maximum of many offerings (Crow's Foot in the outside position). Similarly, an offering is related to exactly one (one and only one) course as shown by the single vertical lines in both inside and outside positions.

FIGURE 5.2
Instance Diagram for the *Has* Relationship



[2] A transitive verb can take a direct object indicating the receiver of the action. Most action verbs can take direct objects.

FIGURE 5.3
Entity Relationship Diagram with Cardinalities Noted



Classification of Cardinalities Cardinalities are classified by common values for minimum and maximum cardinality. Table 5-1 shows two classifications for minimum cardinalities. A minimum cardinality of one or more indicates a mandatory relationship. For example, participation in the *Has* relationship is mandatory for each *Offering* entity due to the minimum cardinality of one. A mandatory relationship makes the entity type **existence dependent** on the relationship. The *Offering* entity type depends on the *Has* relationship because an *Offering* entity cannot be stored without a related *Course* entity. In contrast, a minimum cardinality of zero indicates an optional relationship. For example, the *Has* relationship is optional to the *Course* entity type because a *Course* entity can be stored without being related to an *Offering* entity. Figure 5.4 shows that the *Teaches* relationship is optional for both entity types.

Existence Dependency
an entity that cannot exist unless another related entity exists. A mandatory relationship creates an existence dependency.

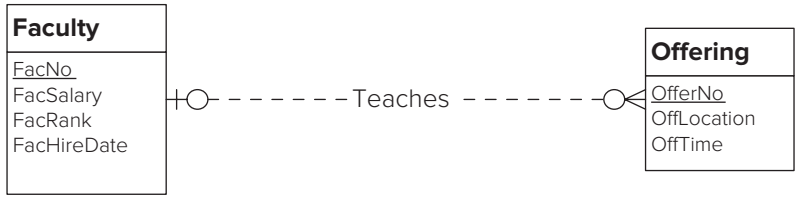
Table 5-1 also shows several classifications for maximum cardinalities. A maximum cardinality of one means the relationship is single-valued or functional. For example, the *Has* and *Teaches* relationships are functional for *Offering* because an *Offering* entity can be related to a maximum of one *Course* and one *Faculty* entity. The word function comes from mathematics where a function gives one value. A relationship that has a maximum cardinality of one in one direction and more than one (many) in the other direction is called a 1-M (read one-to-many) relationship. Both the *Has* and *Teaches* relationships are 1-M.

Similarly, a relationship that has a maximum cardinality of more than one in both directions is known as an M-N (many-to-many) relationship. In Figure 5.5, the *TeamTeaches* relationship allows multiple professors to jointly teach the same offering, as shown in the instance diagram of Figure 5.6. M-N relationships are common

TABLE 5-1
Summary of Cardinality Classifications

Classification	Cardinality Restrictions
Mandatory	Minimum cardinality ≥ 1
Optional	Minimum cardinality = 0
Functional or single-valued	Maximum cardinality = 1
1-M	Maximum cardinality = 1 in one direction and maximum cardinality > 1 in the other direction.
M-N	Maximum cardinality is > 1 in both directions.
1-1	Maximum cardinality = 1 in both directions.

FIGURE 5.4
Optional Relationship for Both Entity Types



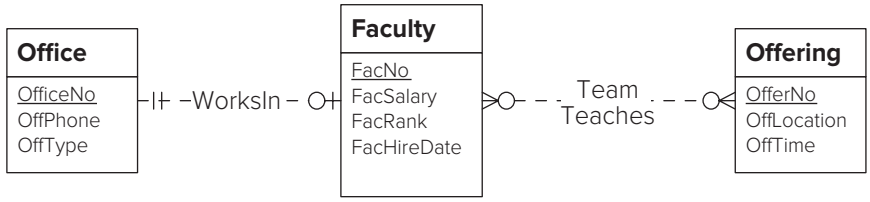
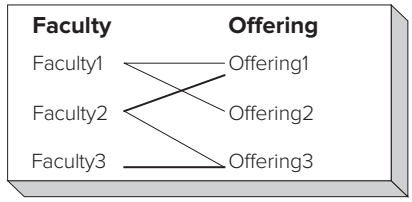


FIGURE 5.5
M-N and 1-1 Relationship Examples

in business databases to represent the connection between parts and suppliers, authors and books, and skills and employees. For example, a part can be supplied by many suppliers and a supplier can supply many parts.

Less common are 1-1 relationships in which the maximum cardinality equals one in both directions. For example, the *WorksIn* relationship in Figure 5.5 allows a faculty to be assigned to one office and an office to be occupied by at most one faculty.

FIGURE 5.6
Instance Diagram for the M-N *TeamTeaches* Relationship



5.1.3 Comparison to Relational Database Diagrams

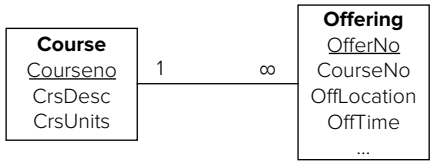
To finish this section, let us compare the notation in Figure 5.3 with the relational database diagrams (from Microsoft Access) which you seen in previous chapters. It is easy to become confused between the two notations. Some of the major differences are listed below.[3] To help you visualize these differences, Figure 5.7 shows a relational database diagram for the *Course-Offering* example.

1. Relational database diagrams do not use names for relationships. Instead foreign keys represent relationships. The ERD notation does not use foreign keys. For example, *Offering.CourseNo* is a column in Figure 5.7 but not an attribute in Figure 5.3.
2. Relational database diagrams show only maximum cardinalities.
3. Some ERD notations (including the Crow's Foot notation) allow both entity types and relationships to have attributes. Relational database diagrams only allow tables to have columns.
4. Relational database diagrams allow a relationship between two tables. Some ERD notations (although not the Crow's Foot notation) allow M-way relationships involving more than two entity types. The next section shows how to represent M-way relationships in the Crow's Foot notation.
5. In some ERD notations (although not the Crow's Foot notation), the position of the cardinalities is reversed.

5.2 UNDERSTANDING RELATIONSHIPS

This section explores the entity relationship notation in more depth by examining important aspects of relationships. The first subsection describes identification dependency, a specialized kind of existence dependency. The second subsection describes three important relationship patterns: (1) relationships with attributes, (2) self-referencing relationships, and (3) associative entity types representing multiway (M-way) relationships. The final subsection describes an important equivalence between M-N and 1-M relationships.

FIGURE 5.7
Relational Database Diagram for the Course-Offering Example



5.2.1 Identification Dependency (Weak Entity Types and Identifying Relationships)

In an ERD, some entity types may not have their own primary key. Entity types without their own primary key must borrow part (or all)

[3] Chapter 6 presents conversion rules that describe the differences more precisely.

Weak Entity Type
an entity type that borrows all or part of its primary key from another entity type. Identifying relationships indicate the entity types that supply components of a borrowed primary key.

of their primary key from other entity types. Entity types that borrow part or their entire primary key are known as **weak entity types**. A relationship that provides a component of the primary key is known as an identifying relationship. Thus, an identification dependency involves a weak entity type and one or more identifying relationships.

Identification dependency occurs because some entities are closely associated with other entities. For example, a room does not have a separate identity from its building because a room is physically contained in a building. You can reference a room only by providing its associated building identifier. In the ERD for buildings and rooms (Figure 5.8), the *Room* entity type is identification dependent on the *Building* entity type in the *Contains* relationship. A solid relationship line indicates an identifying relationship. For weak entity types, the underlined attribute (if present) is part of the primary key, but not the entire primary key. Thus, the primary key of *Room* is a combination of *BldgID* and *RoomNo*. As another example, Figure 5.9 depicts an identification dependency involving the weak entity type *State* and the identifying relationship *Holds*.

Identification dependency is a specialized kind of existence dependency. Recall that an existent-dependent entity type has a mandatory relationship (minimum cardinality of one). Weak entity types are existent dependent on the identifying relationships. In addition to the existence dependency, a weak entity type borrows at least part of its entire primary key. Because of the existence dependency and the primary key borrowing, the minimum and maximum cardinalities of a weak entity type are always 1.

The next section shows several additional examples of identification dependency in the discussion of associative entity types and M-way relationships. The use of identification dependency is necessary for associative entity types.

5.2.2 Relationship Patterns

This section discusses three patterns for relationships that you may encounter in database development efforts: (1) M-N relationships with attributes, (2) self-referencing (unary) relationships, and (3) associative entity types representing M-way relationships. Although these relationship patterns are not common, they are important when

FIGURE 5.8
Identification Dependency Example

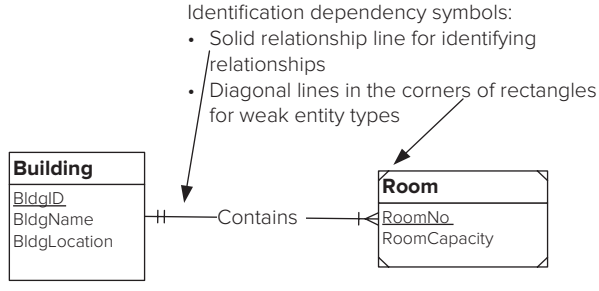
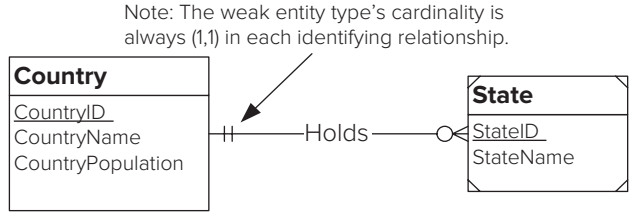


FIGURE 5.9
Another Identification Dependency Example



they occur. You need to study these patterns carefully to apply them correctly in database development efforts.

M-N Relationships with Attributes As briefly mentioned in Section 5.1, relationships can have attributes. This situation typically occurs with M-N relationships. In an M-N relationship, attributes are associated with the combination of entity types, not just one of the entity types. If an attribute is associated with only one entity type, then it should be part of that entity type, not the relationship. Figures 5.10 and 5.11 depict M-N relationships with attributes. In Figure 5.10, the attribute *EnrGrade* is associated with the combination of a student and offering, not either one alone. For example, the *EnrollsIn* relationship records the fact that the student with *StdNo* 123-77-9993 has a grade of 3.5 in the offering with offer number 1256. In Figure 5.11(a), the attribute *Qty* represents the quantity of a part supplied by a given supplier. In Figure 5.11(b), the attribute *AuthOrder* represents the order in which the author's name appears in the title of a book. To reduce clutter on a large diagram, relationship attributes may not be shown.

1-M relationships also can have attributes, but 1-M relationships with attributes are much less common than M-N relationships with attributes. In Figure 5.12, the *Commission* attribute is associated with the *Lists* relationship, not with either the *Agent* or the *Home* entity type. A home will only have a commission if an agent lists it. Typically, 1-M relationships with attributes are optional for the child entity type. The *Lists* relationship is optional for the *Home* entity type.

Self-Referencing Relationship
a relationship involving the same entity type. Self-referencing relationships represent associations among members of the same set.

Self-Referencing (Unary) Relationships A **self-referencing relationship** involves connections among members of the same set. Self-referencing relationships are sometimes called reflexive relationships because they turn back on themselves,

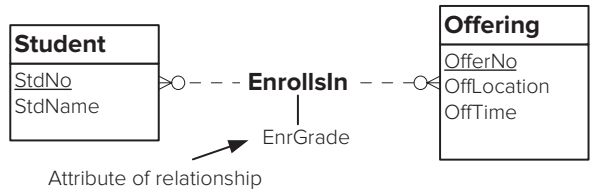


FIGURE 5.10
M-N Relationship with an Attribute

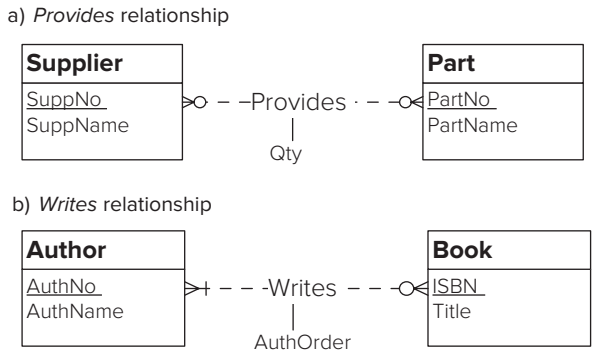


FIGURE 5.11
Additional M-N Relationships with Attributes

FIGURE 5.12
1-M Relationship with an Attribute

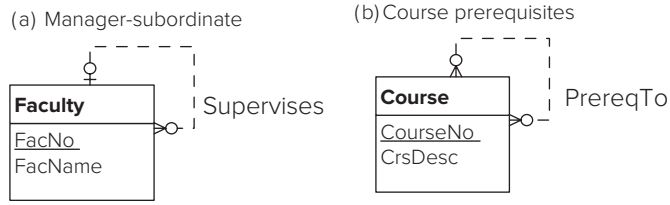


FIGURE 5.13
Examples of Self-Referencing (Unary) Relationships

similar to the concept of a reflexive verb in English. Figure 5.13 displays two self-referencing relationships involving the *Faculty* and *Course* entity types. Both relationships involve two entity types that are the same (*Faculty* for *Supervises* and *Course* for *PreReqTo*). These relationships depict important concepts in a university database. The *Supervises* relationship depicts an organizational chart, while the *PreReqTo* relationship depicts course dependencies that can affect a student's course planning.

For self-referencing relationships, you should distinguish between 1-M and M-N relationships. An instance diagram can help you understand the difference. Figure 5.14(a) shows an instance diagram for the *Supervises* relationship. Notice that each faculty can have at most one superior. For example, Faculty2 and Faculty3 have Faculty1 as a superior. Therefore, *Supervises* is a 1-M relationship because each faculty can have at most one supervisor. In contrast, there is no such restriction in the instance diagram for the *PreReqTo* relationship (Figure 5.14(b)). For example, IS461 has two prerequisites (IS480 and IS460), while IS320 is a prerequisite to both IS480 and IS460. Therefore, *PreReqTo* is an M-N relationship because a course can be a prerequisite to many courses, and a course can have many prerequisites.

Self-referencing relationships occur in a variety of business situations. Any data that can be visualized like Figure 5.14 can be represented as a self-referencing relationship. Typical examples include hierarchical charts of accounts, genealogical charts, part designs, and transportation routes. In these examples, self-referencing relationships are an important part of the database.

There is one other noteworthy aspect of self-referencing relationships. Sometimes a self-referencing relationship is not needed. For example, if you only want to know whether an employee is a supervisor, a self-referencing relationship is not needed. Rather, an attribute can be used to indicate whether an employee is a supervisor.

Associative Entity Types Representing Multi-Way (M-Way) Relationships Some ERD notations support relationships involving more than two entity types known as M-way (multiway) relationships where the *M* means more than two. For example, the Chen[4] ERD notation (with diamonds for relationships) allows relationships to

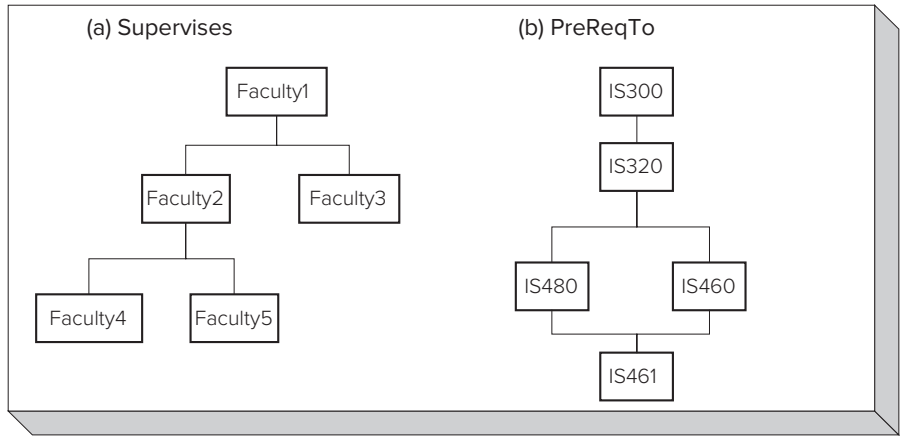


FIGURE 5.14
Instance Diagrams for Self-Referencing Relationships

[4] The Chen notation is named after Dr. Peter Chen, who published the paper defining the Entity Relationship Model in 1976.

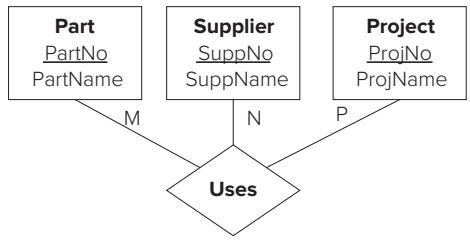


FIGURE 5.15
M-Way (Ternary) Relationship Using the Chen Notation

connect more than two entity types, as depicted in Figure 5.15. The *Uses* relationship lists suppliers and parts used on projects. For example, a relationship instance involving Supplier1, Part1, and Project1 indicates that Supplier1 Supplies Part1 on Project1. An M-way relationship involving three entity types is called a ternary relationship. The letters in the Chen ERD indicate maximum cardinalities.

Although you cannot directly represent M-way relationships in the Crow's Foot notation, you should understand how to indirectly represent them. You use an **associative entity type** and a collection of identifying 1-M relationships to represent an M-way relationship. In Figure 5.16, three 1-M relationships link the associative entity type, *Uses*, to the *Part*, the *Supplier*, and the *Project* entity types. The *Uses* entity type is associative because its role is to connect other entity types. Because associative entity types provide a connecting role, they are sometimes given names using active verbs. In addition, associative entity types are always weak as they must borrow the entire primary key. For example, the *Uses* entity type obtains its primary key through the three identifying relationships.

Associative Entity Type
a weak entity that depends on two or more entity types for its primary key. An associative entity type with more than two identifying relationships is known as an M-way associative entity type.

As another example, Figure 5.17 shows the associative entity type *Provides* that connects the *Employee*, *Skill*, and *Project* entity types. An example instance of the *Provides* entity type contains Employee1 providing Skill1 on Project1.

The issue of when to use an M-way associative entity type (i.e., an associative entity type representing an M-way relationship) can be difficult to understand. If

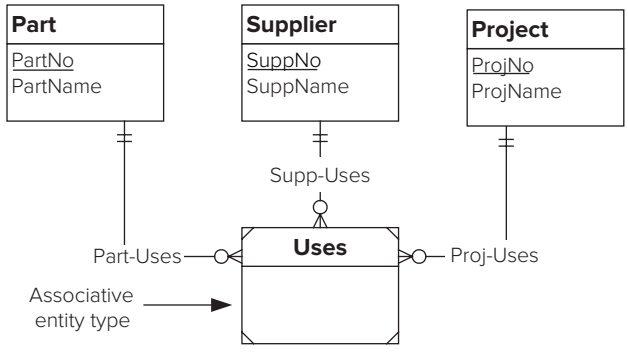


FIGURE 5.16
Associative Entity Type to Represent a Ternary Relationship

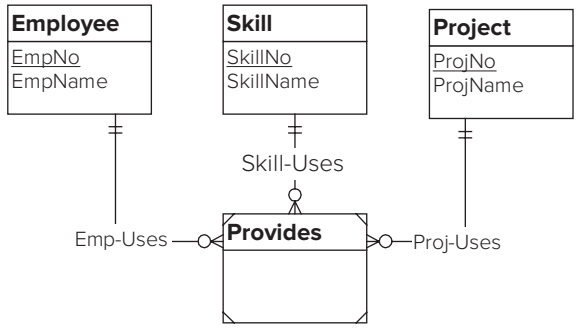


FIGURE 5.17
Associative Entity Type Connecting Employee, Skill, and Project

a database only needs to record pairs of facts, an M-way associative entity type is not needed. For example, if a database only needs to record who supplies a part and what projects use a part, then an M-way associative entity type should not be used. In this case, there should be binary relationships between *Supplier* and *Part* and between *Project* and *Part*. You should use an M-way associative entity type when the database should record combinations of three (or more) entities rather than just combinations of two entities. For example, if a database needs to record which supplier provides parts on specific projects, an M-way associative entity type is needed. Because of the complexity of M-way relationships, Chapter 6 provides a guideline about using them, while Chapter 7 provides a way to reason about them using constraints.

5.2.3 Equivalence between 1-M and M-N Relationships

Relationship Equivalence
an M-N relationship can be replaced by an associative entity type and two identifying 1-M relationships. In most cases, the choice between a M-N relationship and the associative entity type is personal preference.

To improve your understanding of M-N relationships, you should know an important **equivalence for M-N relationships**. An M-N relationship can be replaced by an associative entity type and two identifying 1-M relationships. Figure 5.18 shows the *EnrollsIn* (Figure 5.10) relationship converted to this 1-M style. In Figure 5.18, two identifying relationships and an associative entity type replace the *EnrollsIn* relationship. The relationship name (*EnrollsIn*) has been changed to a noun (*Enrollment*) to follow the convention of nouns for entity type names. The 1-M style is similar to the representation in a relational database diagram. If you feel more comfortable with the 1-M style, then use it. In terms of the ERD, the M-N and 1-M styles have the same meaning.

The transformation of an M-N relationship into 1-M relationships is similar to representing an M-way relationship using 1-M relationships. Whenever an M-N relationship is represented as an associative entity type and two 1-M relationships, the new entity type is identification dependent on both 1-M relationships, as shown in Figure 5.18. Similarly, when representing M-way relationships, the associative entity type is identification dependent on all 1-M relationships as shown in Figures 5.16 and 5.17.

There is one situation when the 1-M style is preferred to the M-N style. When an M-N relationship must be related to other entity types in relationships, you should use the 1-M style. For example, assume that in addition to enrollment in a course offering, attendance in each class session should be recorded. In this situation, the 1-M style is necessary to link an enrollment with attendance records. Figure 5.19 shows the *Attendance* entity type added to the ERD of Figure 5.18. Note that an M-N relationship between the *Student* and *Offering* entity types would not have allowed another relationship with *Attendance*.

Figure 5.19 provides other examples of identification dependencies. *Attendance* is identification dependent on *Enrollment* in the *RecordedFor* relationship. The primary key of *Attendance* consists of *AttDate* along with the primary key of *Enrollment*. Similarly, *Enrollment* is identification dependent on both *Student* and *Offering*. The primary key of *Enrollment* is a combination of *StdNo* and *OfferNo*.

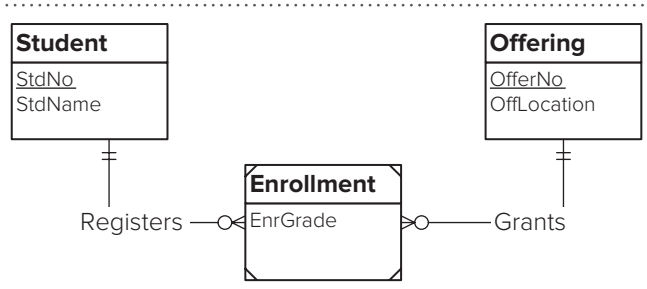


FIGURE 5.18
EnrollsIn M-N Relationship (Figure 5.10) Transformed into 1-M Relationships

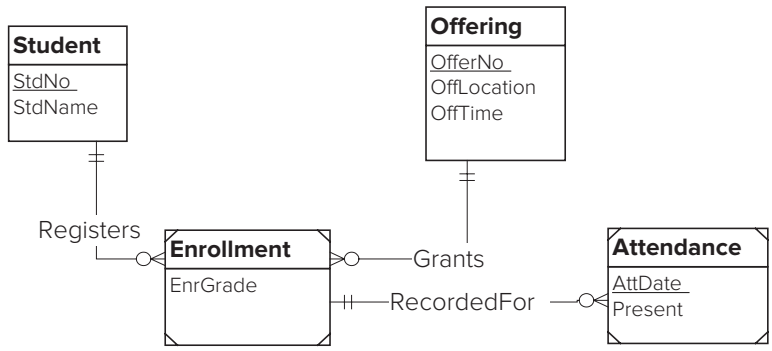


FIGURE 5.19
Attendance Entity Type Added to the ERD of Figure 18

5.3 CLASSIFICATION IN THE ENTITY RELATIONSHIP MODEL

People classify entities to better understand their environment. For example, animals are classified into mammals, reptiles, and other categories to understand the similarities and differences among different species. In business, classification is also pervasive. Classification can be applied to investments, employees, customers, loans, parts, and so on. For example, when applying for a home mortgage, an important distinction is between fixed- and adjustable-rate mortgages. Within each kind of mortgage, there are many variations distinguished by features such as the repayment period, prepayment penalties, and loan amount.

This section describes ERD notation to support classification. You will learn to use generalization hierarchies, specify cardinality constraints for generalization hierarchies, and use multiple-level generalization hierarchies for complex classifications.

5.3.1 Generalization Hierarchies

Generalization hierarchies allow entity types to be related by the level of specialization. Figure 5.20 depicts a generalization hierarchy to classify employees as salaried versus hourly. Both salaried and hourly employees are specialized kinds of employees. The *Employee* entity type is known as the supertype (or parent). The entity types *SalaryEmp* and *HourlyEmp* are known as the subtypes (or children). Because each subtype entity *is a* supertype entity, the relationship between a subtype and supertype is known as ISA. For example, a salaried employee is an employee. Because the relationship name (ISA) is always the same, it is not shown on the diagram.

Inheritance supports sharing between a supertype and its subtypes. Because every subtype entity is also a supertype entity, the attributes of the supertype also apply to all subtypes. For example, every entity of *SalaryEmp* has an employee number, name, and hiring date because it is also an entity of *Employee*. Inheritance means that

Generalization Hierarchy
a collection of entity types arranged in a hierarchical structure to show similarity in attributes. Each subtype or child entity type contains a subset of entities of its supertype or parent entity type.

Inheritance
a data modeling feature that supports sharing of attributes between a supertype and a subtype. Subtypes inherit attributes from their supertypes.

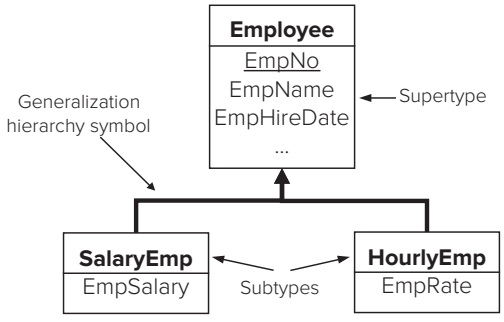


FIGURE 5.20
Generalization Hierarchy for Employees

the attributes of a supertype are automatically part of its subtypes. That is, each subtype inherits the attributes of its supertype. For example, the attributes of the *SalaryEmp* entity type are its direct attribute (*EmpSalary*) and its inherited attributes from *Employee* (*EmpNo*, *EmpName*, *EmpHireDate*, etc.). Inherited attributes are not shown in an ERD. Whenever you have a subtype, assume that it inherits the attributes from its supertype.

5.3.2 Disjointness and Completeness Constraints

Generalization hierarchies do not show cardinalities because they are always the same. Rather disjointness and completeness constraints can be shown. Disjointness means that subtypes in a generalization hierarchy do not have any entities in common. In Figure 5.21, the generalization hierarchy is disjoint because a security cannot be both a stock and a bond. In contrast, the generalization hierarchy in Figure 5.22 is not disjoint because teaching assistants can be considered both students and faculty. Thus, the set of students overlaps with the set of faculty. Completeness means that every entity of a supertype must be an entity in one of the subtypes in the generalization hierarchy. The completeness constraint in Figure 5.21 means that every security must be either a stock or a bond.

Some generalization hierarchies lack both disjointness and completeness constraints. In Figure 5.20, the lack of a disjointness constraint means that some employees can be both salaried and hourly. The lack of a completeness constraint indicates that some employees are not paid by salary or the hour (perhaps by commission).

5.3.3 Multiple Levels of Generalization

Generalization hierarchies can be extended to more than one level. This practice can be useful in disciplines such as investments where knowledge is highly structured. In Figure 5.23, there are two levels of subtypes beneath securities. Inheritance extends to all subtypes, direct and indirect. Thus, both the *Common* and *Preferred* entity types inherit the attributes of *Stock* (the immediate parent) and *Security* (the indirect parent). Note that disjointness and completeness constraints can be made for each group of subtypes.

FIGURE 5.21
Generalization Hierarchy for Securities

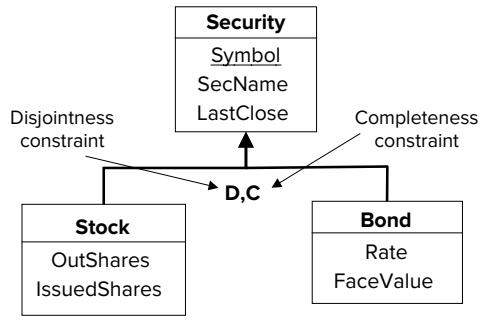
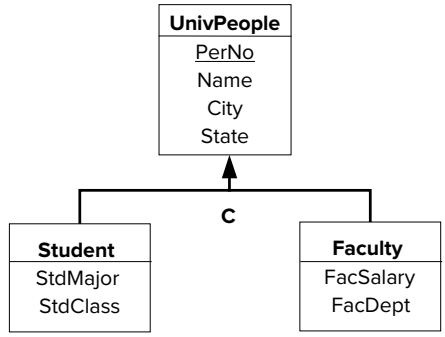


FIGURE 5.22
Generalization Hierarchy for University People



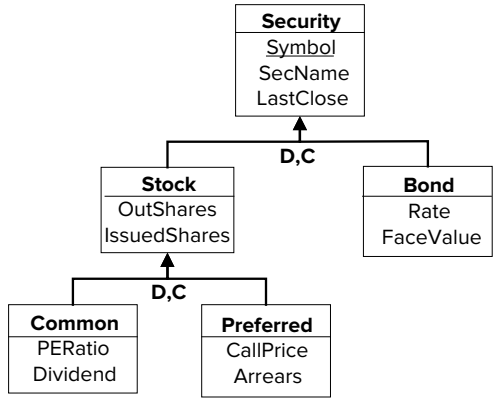


FIGURE 5.23
Multiple Levels of Generalization Hierarchies

5.4 NOTATION SUMMARY AND DIAGRAM RULES

You have seen a lot of ERD notation in the previous sections of this chapter. So that you do not become overwhelmed, this section provides a convenient summary as well as rules to help you avoid common diagramming errors.

5.4.1 Notation Summary

To help you recall the notation introduced in previous sections, Table 5-2 presents a summary while Figure 5.24 demonstrates the notation for the university database of Chapter 4. Figure 5.24 differs in some ways from the university database in Chapter 4 to depict most of the Crow's Foot notation. Figure 5.24 contains a generalization hierarchy to depict the similarities among students and faculty. You should note that

TABLE 5-2
Summary of Crow's Foot Notation

Symbol	Meaning
Student StdNo StdName ...	Entity type with attributes (primary key underlined)
Enrolls_In EnrGrade	M-N relationship with attributes: attributes appear if room permits; otherwise attributes are listed separately.
Contains	Identification dependency: identifying relationship(s) (solid relationship lines) and weak entity (diagonal lines in the corners of the rectangle). Associative entity types also are weak because they are (by definition) identification dependent.
D,C	Generalization hierarchy with disjointness and completeness constraints.
Contains	Existence dependent cardinality (minimum cardinality of 1): inner symbol is a line perpendicular to the relationship line.
Teaches	Optional cardinality (minimum cardinality of 0): inner symbol is a circle.
Has	Single-valued cardinality (maximum cardinality of 1): outer symbol is a perpendicular line.

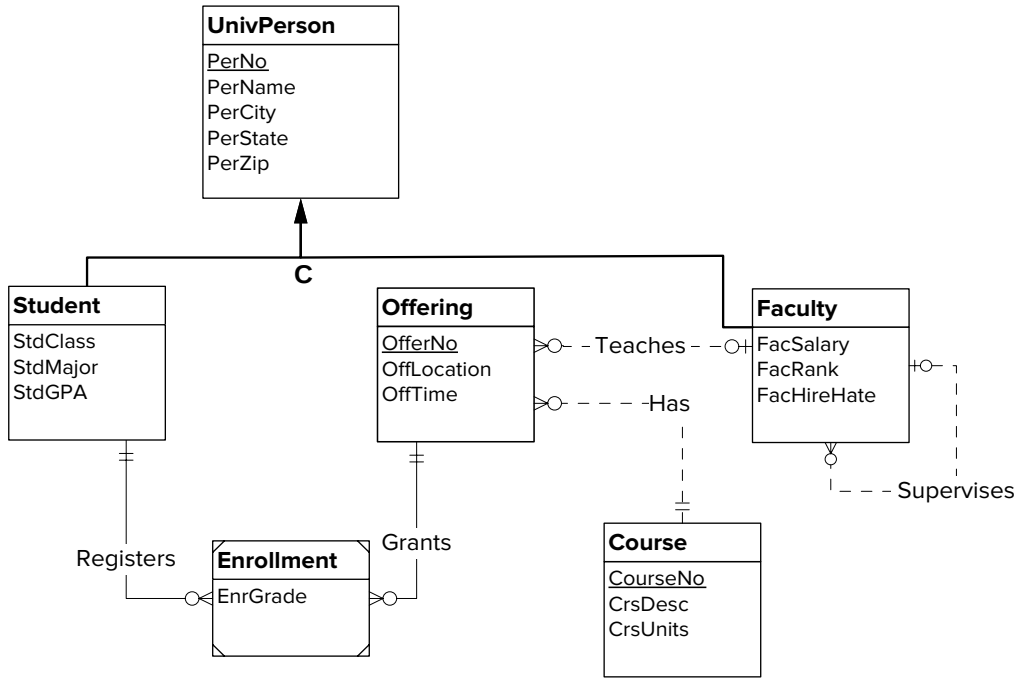


FIGURE 5.24
ERD for the University Database

the primary key of the *Student* and the *Faculty* entity types is *PerNo*, an attribute inherited from the *UnivPerson* entity type. The *Enrollment* entity type (associative) and the identifying relationships (*Registers* and *Grants*) could appear as an M-N relationship as previously shown in Figure 5.10. In addition to these issues, Figure 5.24 omits some attributes for breviy.

Representation of Business Rules in an ERD As you develop an ERD, you should remember that an ERD contains business rules that enforce organizational policies and promote efficient communication among business stakeholders. An ERD contains important business rules represented as primary keys, relationships, cardinalities, and generalization hierarchies. Primary keys support entity identification, an important requirement in business communication. Identification dependency involves an entity that depends on other entities for identification, a requirement in some business communication. Relationships indicate direct connections among units of business communication. Cardinalities restrict the number of related entities in relationships supporting organizational policies and consistent business communication. Generalization hierarchies with disjointness and completeness constraints support classification of business entities and organizational policies. Thus, the elements of an ERD are crucial for enforcement of organizational policies and efficient business communication.

For additional kinds of business constraints, you can enhance an ERD with informal documentation or a formal rules language. Since the SQL standard supports integrity constraints in the CREATE TABLE statement (see Chapter 3) for simple rules and a formal rules language (see Chapters 11 and 16) for complex constraints, this chapter does not present a language or notation for ERDs. In the absence of a formal rules language, business rules can be stored as informal documentation associated with entity types, attributes, and relationships. Typical kinds of business rules to specify as informal documentation are candidate key constraints, attribute comparison constraints, null value constraints, and default values. Candidate keys provide alternative ways to identify business entities. Attribute comparison constraints restrict the values of attributes either to a fixed collection of values or to values of other attributes. Null value constraints and default values support policies about completeness of data collection activities. Table 5-3 summarizes the common kinds of business rules that you can specify either formally or informally in an ERD.

TABLE 5-3
Summary of Business Rules in an ERD

Business Rule	ERD Representation
Entity identification	Primary keys for entity types, identification dependency (weak entities and identifying relationships), informal documentation about other unique attributes
Connections among business entities	Relationships
Number of related entities	Minimum and maximum cardinalities
Inclusion among entity sets	Generalization hierarchies
Reasonable values	Informal documentation about attribute constraints (comparison to constant values or other attributes)
Data collection completeness	Informal documentation about null values and default values

5.4.2 Diagram Rules

To provide guidance about correct usage of the notation, Table 5-4 presents completeness and consistency rules. You should apply these rules when completing an ERD to ensure that no notation errors exist in your ERD. Thus, the diagram rules serve a purpose similar to syntax rules for a computer language. The absence of syntax errors does not mean that a computer program performs its tasks correctly. Likewise, the absence of notation errors does not mean that an ERD provides an adequate data representation. The diagram rules do not ensure that you have considered multiple alternatives, correctly represented user requirements, and properly documented your design. Chapter 6 discusses these issues to enhance your data modeling skills.

TABLE 5-4
Completeness and Consistency Rules

Type of Rule	Description
Completeness	1. Primary key rule: All entity types have a primary key (direct, borrowed, or inherited). 2. Naming rule: All entity types, relationships, and attributes are named. 3. Cardinality rule: Cardinality is given for both entity types in a relationship. 4. Entity participation rule: All entity types except those in a generalization hierarchy participate in at least one relationship. 5. Generalization hierarchy participation rule: Each generalization hierarchy participates in at least one relationship with an entity type not in the generalization hierarchy.
Consistency	1. Entity name rule: Entity type names are unique. 2. Attribute name rule: Attribute names are unique within entity types and relationships. 3. Inherited attribute name rule: Attribute names in a subtype do not match inherited (direct or indirect) attribute names. 4. Relationship/entity type connection rule: All relationships connect two entity types (not necessarily distinct). 5. Relationship/relationship connection rule: Relationships are not connected to other relationships. 6. Weak entity type rule: Weak entity types have at least one identifying relationship. 7. Identifying relationship rule: For each identifying relationship, at least one participating entity type must be weak. 8. Identification dependency cardinality rule: For each identifying relationship, the minimum and maximum cardinality must be 1 in the direction from the child (weak entity) to the parent entity type. 9. Redundant foreign key rule: Redundant foreign keys are not used.

Most of the rules in Table 5-4 do not require much elaboration. The first three completeness rules and the first five consistency rules are simple to understand. Even though the rules are simple, you should still check your ERDs for compliance as it is easy to overlook a violation in a moderate-size ERD.

The consistency rules do not require unique relationship names because participating entity types provide a context for relationship names. However, it is good practice to use unique relationship names as much as possible to make relationships easy to distinguish. In addition, two or more relationships involving the same entity types must be unique because the entity types no longer provide a context to distinguish the relationships. Since it is uncommon to have more than one relationship between the same entity types, the consistency rules do not include this provision.

Completeness rules 4 (entity participation rule) and 5 (generalization hierarchy participation rule) require elaboration. Violating these rules is a warning, not necessarily an error. In most ERDs, all entity types not in a generalization hierarchy and all generalization hierarchies are connected to at least one other entity type. In uncommon situations, an ERD contains an unconnected entity type just to store a list of entities. Rule 5 applies to an entire generalization hierarchy, not to each entity type in a generalization hierarchy. In other words, at least one entity type in a generalization hierarchy should be connected to at least one entity type not in the generalization hierarchy. In many generalization hierarchies, multiple entity types participate in relationships. Generalization hierarchies permit subtypes to participate in relationships thus constraining relationship participation. For example in Figure 5.24, *Student* and *Faculty* participate in relationships.

Consistency rules 6 through 9 involve common errors in ERDs of novice data modelers. Novice data modelers violate consistency rules 6 to 8 because of the complexity of identification dependency. Identification dependency, involving a weak entity type and identifying relationships, provides more sources of errors than other parts of the Crow's Foot notation. In addition, each identifying relationship also requires a minimum and maximum cardinality of 1 in the direction from the child (weak entity type) to the parent entity type. Novice data modelers violate consistency rule 9 (redundant foreign key rule) because of confusion between an ERD and the relational data model. The conversion process transforms 1-M relationships into foreign keys.

Example of Rule Violations and Resolutions Because the identification dependency rules and the redundant foreign key rule are a frequent source of errors to novice designers, this section provides an example to depict rule violations and resolutions. Figure 5.25 demonstrates violations of the identification dependency rules (consistency rules 6 to 9) and the redundant foreign key rule (consistency rule 9) for the university database ERD. The following list explains the violations.

- Consistency rule 6 (weak entity type rule) violation: *Faculty* cannot be a weak entity type without at least one identifying relationship.
- Consistency rule 7 (identifying relationship rule) violation: The *Has* relationship is identifying but *Offering* is not a weak entity type.
- Consistency rule 8 (identification dependency cardinality rule) violation: The cardinality of the *Registers* relationship from *Enrollment* to *Student* should be (1, 1) not (0, Many).
- Consistency rule 9 (redundant foreign key rule) violation: The *CourseNo* attribute in the *Offering* entity type is redundant with the *Has* relationship. Because *CourseNo* is the primary key of *Course*, it should not be an attribute of *Offering* to link an *Offering* to a *Course*. The *Has* relationship provides the linkage to *Course*.

For most rules, you can easily resolve violations. The major task is recognition of the violation. For the identification dependency rules, resolution depends on problem details. The following list suggests possible corrective actions for diagram errors:

- Consistency rule 6 (weak entity type rule) resolution: You can resolve this problem by adding one or more identifying relationships or by changing the

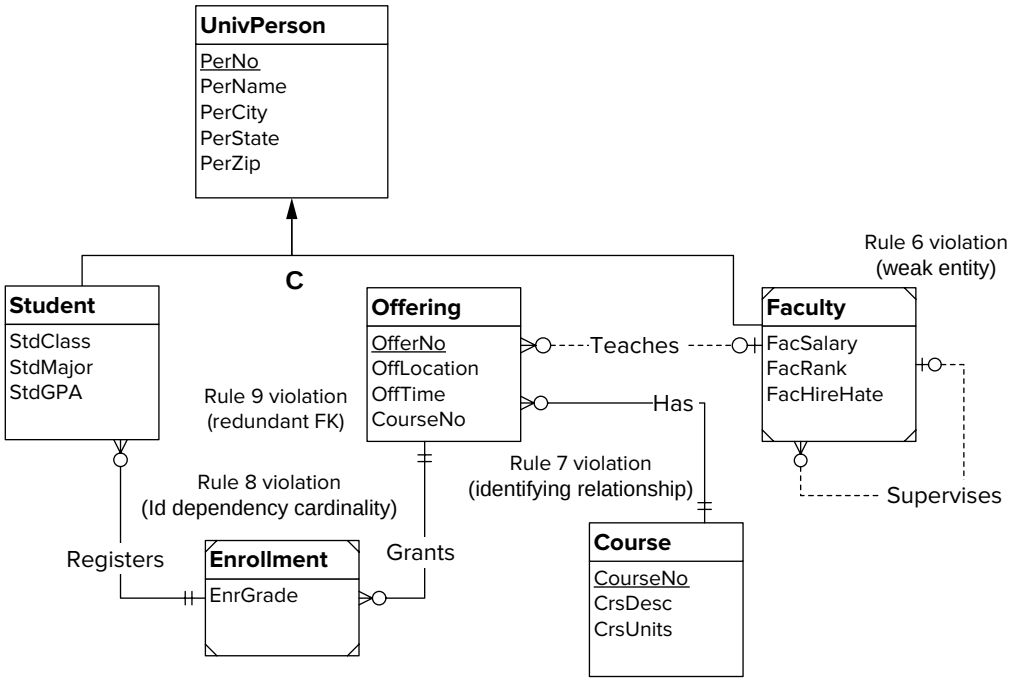


FIGURE 5.25
ERD with Violations of Consistency Rules 6 to 9

weak entity type into a regular entity type. In Figure 5.25, the resolution involves *Faculty* from a weak to regular entity type. The more common resolution involves adding one or more identifying relationships.

- Consistency rule 7 (identifying relationship rule) resolution: You can resolve this problem by adding a weak entity type or making the relationship non-identifying. In Figure 5.25, the resolution involves changing the *Has* relationship from identifying into regular (non-identifying). If more than one identifying relationship exists among the same entity type, the typical resolution involves designating the common entity type as a weak entity type.
- Consistency rule 8 (identification dependency cardinality rule) resolution: You can resolve this problem by changing the weak entity type's cardinality to (1,1). Typically, the cardinality of the identifying relationship is reversed. In Figure 5.25, the resolution involves reversing the cardinality of the *Registers* relationship ((1,1) near *Student* and (0, Many) near *Enrollment*).
- Consistency rule 9 (redundant foreign key rule) resolution: Normally you can resolve this problem by removing the redundant foreign key. In Figure 5.25, the resolution involves removing *CourseNo* as an attribute of *Offering*. In some cases, an attribute may not represent a foreign key. If the attribute does not represent a foreign key, the resolution involves renaming instead of removing the attribute.

Alternative Organization of Rules The organization of rules in Table 5-4 may be difficult to remember. Table 5-5 provides an alternative grouping by rule purpose. If you find this organization more intuitive, you should use it. However you choose to remember the rules, the important point is to apply them after you have completed an ERD. To help you apply diagram rules, most CASE tools perform checks specific to the notations supported by the tools.

These rules can be supported in CASE tools with data modeling features although support for structural rules is uneven in commercial products. Consistency rules 4 and 5 can be supported through diagram construction. Relationships must be connected to two entity types (not necessarily distinct) prohibiting violations of consistency rules 4 and 5. For the other completeness and consistency rules, an analysis tool can generate a report of rule violations. However, the analysis tool would not require fixing rule

TABLE 5-5
Alternative Rule Organization

Category	Rules
Names	All entity types, relationships, and attributes are named. (Completeness rule 2) Entity type names are unique. (Consistency rule 1) Attribute names are unique within entity types and relationships. (Consistency rule 2) Attribute names in a subtype do not match inherited (direct or indirect) attribute names. (Consistency rule 3)
Content	All entity types have a primary key (direct, borrowed, or inherited). (Completeness rule 1) Cardinality is given for both entity types in a relationship. (Completeness rule 3)
Connection	All entity types except those in a generalization hierarchy participate in at least one relationship. (Completeness rule 4) Each generalization hierarchy participates in at least one relationship with an entity type not in the generalization hierarchy. (Completeness rule 5) All relationships connect two entity types. (Consistency rule 4) Relationships are not connected to other relationships. (Consistency rule 5) Redundant foreign keys are not used. (Consistency rule 9)
Identification Dependency	Weak entity types have at least one identifying relationship. (Consistency rule 6) For each identifying relationship, at least one participating entity type must be weak. (Consistency rule 7) For each weak entity type, the minimum and maximum cardinality must equal 1 for each identifying relationship. (Consistency rule 8)

violations in an ERD because some rules are soft and the rules may be applied before an ERD is complete.

For the redundant foreign key rule (consistency rule 9), an analysis tool may use a simple implementation to determine if an ERD contains a redundant foreign key. The analysis tool can check the child entity type (entity type on the many side of the relationship) for an attribute with the same name and data type as the primary key in the parent entity type (entity type on the one side of the relationship). If the tool finds an attribute with the same name and data type, a violation is listed in a rule violation report. In practice, most CASE tools provide an option to show redundant foreign keys to reinforce ERD representation in a table design.

5.5 COMPARISON TO OTHER NOTATIONS

The ERD notation presented in this chapter is similar to but not identical to what you may encounter in practice. With no standard notation for ERDs, the marketplace has created a number of reasonably popular ERD notations, each having its own small variations that appear in practice. The notation in this chapter comes from the Crow's Foot stencil in Visio Professional 2010 with the addition of the generalization notation. The notations that you encounter in practice depend on factors such as data modeling tool used in your organization and industry. One thing is certain: you should be prepared to adapt to the notation in use. This section describes ERD variations that you may encounter in several CASE tools, along with the Class Diagram notation of the Unified Modeling Language (UML), an emerging standard for data modeling for software development.

5.5.1 Range of ERD Variations in Data Modeling Tools

Because no ERD standard exists, a variety of representations has emerged. These ERD variations differ on symbols on the diagram and diagram rules. The original ERD notation, known as the Chen notation[5], uses diamonds for relationships, ovals for

[5] Dr. Peter Chen invented the Entity Relationship Model and ERD notation in a 1976 paper although variants of the notation previously existed.

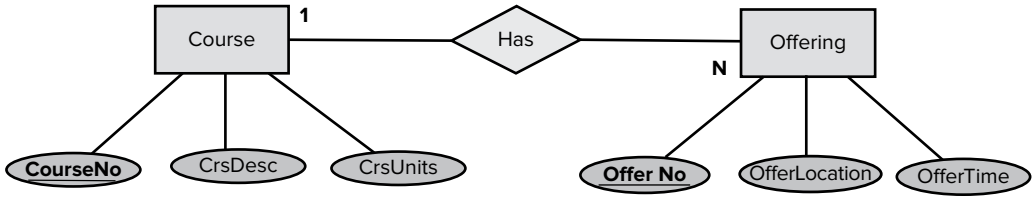


FIGURE 5.26
Chen Notation for the Course-Offering ERD

attributes, and maximum cardinality symbols (1 and M) as shown in Figure 5.26. In addition, the Chen notation deploys different notation for weak entity types (double rectangle), identifying relationships (double diamonds), and M-N relationships with attributes (rectangle with a diamond inside denoting dual qualities of an entity type and relationship). Some ERD drawing tools have modified the original Chen notation so that crow's foot symbols appear for cardinality instead of maximum cardinality symbols.

The Crow's Foot notation has much wider support in ERD drawing tools than the original Chen notation. The Crow's Foot notation replaces the cumbersome symbols for attributes (ovals) and relationships (diamonds) with relationship lines and attribute names inside entity type rectangles. In addition to symbol variations, notation restrictions have made an ERD more similar to a diagram for a table design. Here are common notation restrictions in ERD drawing tools.

- Most notations do not support M-way relationships.
- Some notations do not support M-N relationships.
- Most notations do not support relationships with attributes.
- Some notations do not support self-referencing (unary) relationships.
- Most notations do not permit connections among relationships.
- Most notations show foreign keys as attributes at least as a diagram option.

Restrictions in an ERD notation do not necessarily make the notation less expressive than other notations without the restrictions. Additional symbols in a diagram may be necessary, but the same concepts can still be represented. For example, the Crow's Foot notation does not support M-way relationships. However, M-way relationships can be represented using M-way associative entity types. M-way associative entity types require additional symbols than M-way relationships, but the same concepts are represented.

Data modeling tools support a variety of ERD notations. Many tools support conversion to commercial DBMSs so the representations typically align to relational databases with foreign keys shown and no support for M-N relationships. The next four subsections depict tools (Aqua Data Studio, Oracle SQL Developer, Microsoft Visio Professional, and Visual Paradigm) with these types of restrictions.

5.5.2 ERD Notation in Aqua Data Studio

The data modeling tool in the Aqua Data Studio (www.aquafold.com) uses an ERD notation somewhat aligned to relational database representation. Figure 5.27 depicts the university database developed in the data modeling tool of Aqua Data Studio. The notation is similar to the Crow's Foot notation used in this chapter with solid lines for identifying relationships, dashed lines for regular (non-identifying) relationships, and crow's foot symbols for cardinalities. However, the data modeling tool does not use the weak entity type symbol. In addition, the data modeling tool shows foreign keys with asterisks and does not support M-N relationships.

The data modeling tool in Aqua Data Studio supports generalization hierarchy relationships although in a different representation than presented in this chapter.

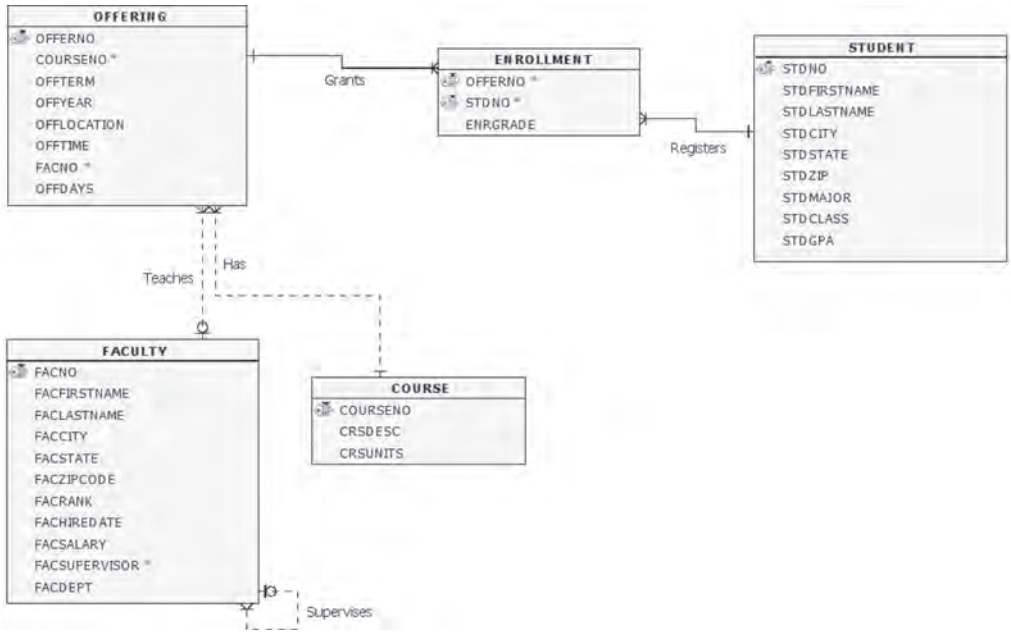


FIGURE 5.27
University Database in the Data Modeling Tool of Aqua Data Studio

Figure 5.28 shows the university database extended with generalization hierarchy relationships for university people. The data modeling tool does not support full generalization hierarchies as shown in Figure 5.24. Instead, generalization relationships are shown separately for each pair (supertype, subtype) of entity types. The data modeling tool supports inheritance as the attributes with asterisks in the *Student* and *Faculty* entity types are inherited from *UnivPerson*. For constraints, the data modeling tool supports only one constraint (inclusive or exclusive) for a generalization relationship instead of the two constraints (disjoint and completeness) presented in this chapter.

The data modeling tool in Aqua Data Studio supports other diagramming features besides diagram construction. The Aqua Data Studio supports several levels of detail

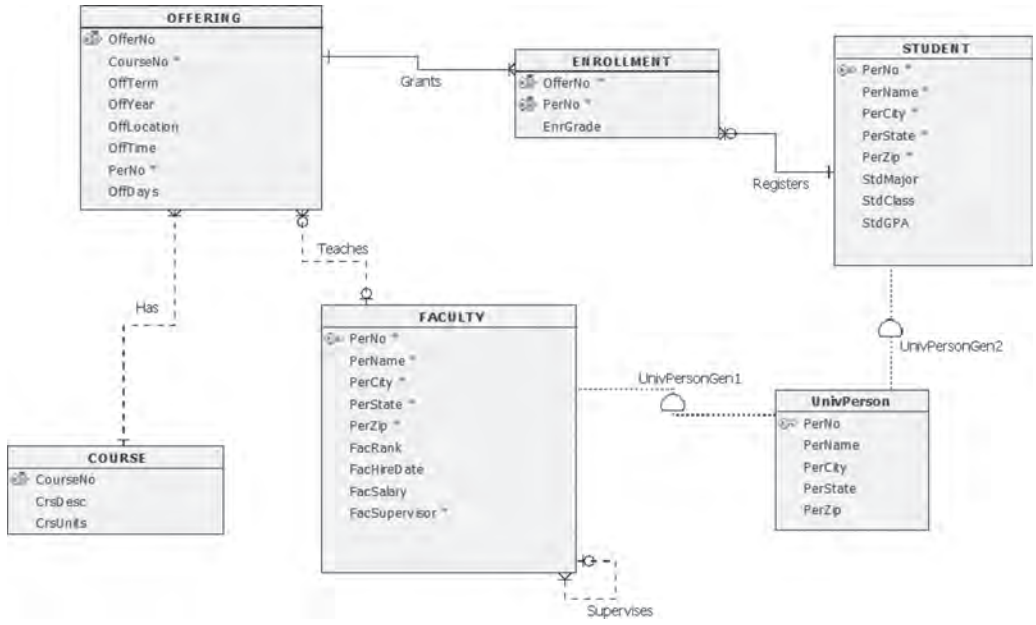


FIGURE 5.28
Extended University Database in the Data Modeling Tool of Aqua Data Studio

in diagrams (attribute level with optional display of data types and null value constraints), entity level, primary key level, comment level, and relationship name level. ERDs can be printed, enlarged, distributed on multiple pages, and laminated for use as posters and quick reference. Regions permit grouping of diagram elements into colored areas that can be manipulated as a unit.

5.5.3 ERD Notation in Oracle SQL Developer

In contrast to Aqua Data Studio, the data modeling tool in Oracle SQL Developer uses a less standard ERD notation as shown in Figure 5.29. The Oracle data modeling tool does not provide relationship names and uses arrows for child to parent relationships. Dashed lines represent optional relationships, while solid lines represent mandatory relationships. Other restrictions in the data modeling tool are no identifying relationships, no M-N relationships, and redundant display of foreign keys. The Oracle data modeling tool also has fewer display and printing options than Aqua Data Studio.

The Oracle data modeling tool supports design rules although not the complete set presented in this chapter. For entity types, the Oracle data modeling tool can detect entity types without relationships, attributes, and a primary key as well as inconsistencies with naming standards. For attributes, the Oracle data modeling tool can detect attributes without a data type and inconsistent naming standards.

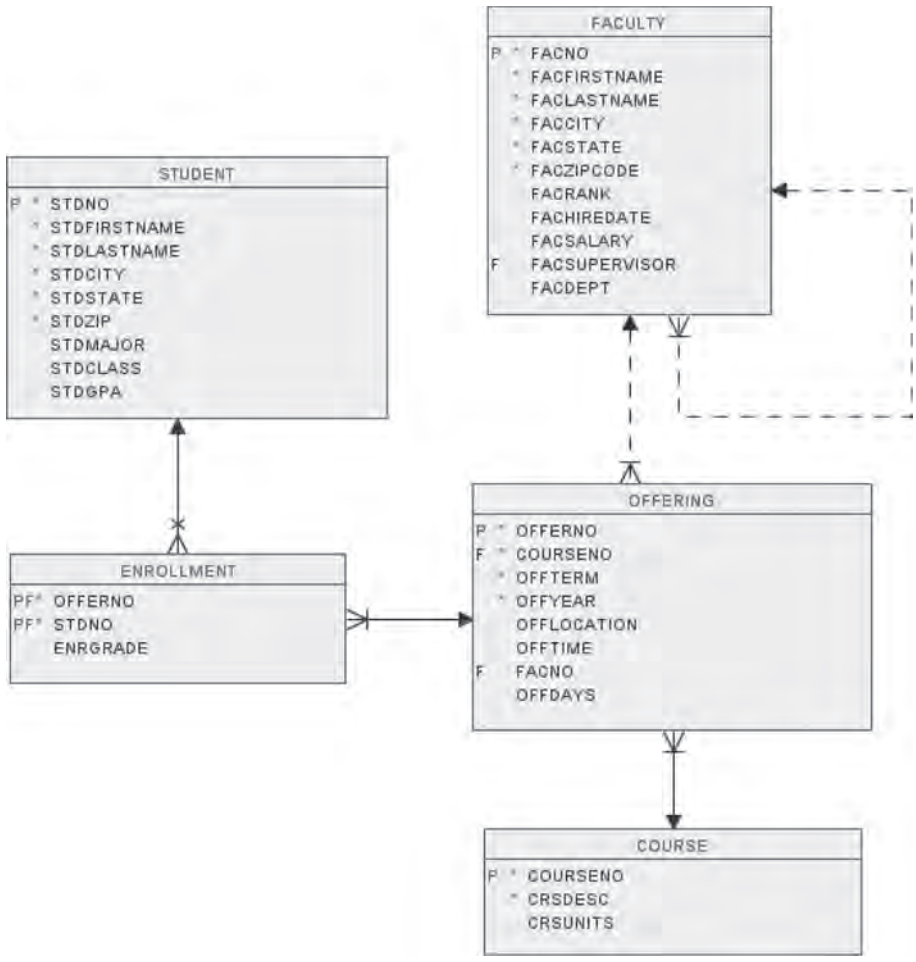


FIGURE 5.29
University Database in the Data Modeling Tool of Oracle SQL Developer

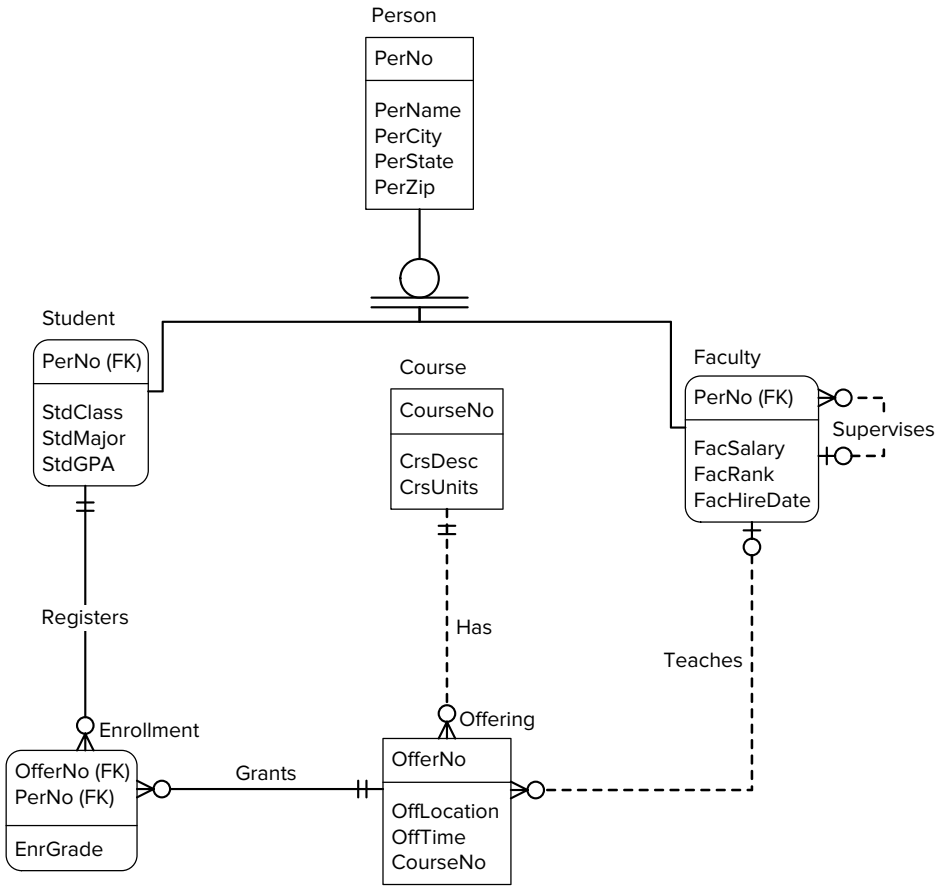
5.5.4 Entity Relationship Stencil in Visio Professional

This section provides details about the Entity Relationship stencil (collection of shapes) available in the Visio 2010 Professional Edition. Using display options for the IDEF1X[6] symbol set and crow's foot cardinality symbols, the Entity Relationship stencil supports most of the chapter notation for Crow's Foot ERDs. The Entity Relationship stencil does not support M-N relationships, uses different symbols for weak entity types, and uses categories instead of generalization hierarchies.

Figure 5.30 depicts the ERD for the extended university database (Figure 5.24) using the Entity Relationship stencil along with options for the IDEF1X symbol set and crow's foot relationship symbols. Some foreign keys are not shown in Figure 5.30 although all foreign keys can be shown by setting a display option. Foreign keys that are part of primary keys are shown when the primary keys are shown. Thus, the *Student*, *Faculty*, and *Enrollment* entity types show foreign keys because they are part of the primary keys. The rounded corner shape denotes a weak entity type and subtypes. The *Student* and *Faculty* entity types are weak because they inherit the primary key from the *Person* entity type. The associative entity type (*Enrollment*) is necessary because Visio Professional does not support M-N relationships.

The Entity Relationship stencil of Visio Professional supports generalization hierarchies using symbols for categories, parent to category connections, and child to category connections. A category contains a collection of subtype entities. Categories in

FIGURE 5.30
Extended University Database in Visio 2010 Professional Edition



[6]The IDEF1X notation was developed in the 1980s to support data modeling needs in U.S. Air Force software projects. IDEF1X uses different symbols than the Crow's Foot notation.

Visio Professional and generalization hierarchies in this chapter differ in several ways as shown in the following list.

- The Visio notation uses a circle with one or two lines below to represent categories.
- The Visio notation allows a parent entity type to have multiple categories. In contrast, the notation in this chapter allows only a single generalization hierarchy for a parent entity type.
- The Visio notation allows a parent entity type to contain a discriminating attribute. A discriminating attribute contains one value for each subtype in the category. The discriminating attribute can be checked in a condition to determine class membership of a parent entity occurrence. This chapter does not use discriminating attributes although such an attribute can be defined. In Visio Professional, a discriminating attribute is an alternative to a disjoint constraint.
- The Visio notation supports completeness constraints, but the symbol (double lines below the circle) is different than shown in this chapter.
- Visio Professional does not support inheritance as the attributes of *Person* are not part of its child entity types (*Student* and *Faculty*).

Visio Professional supports most of the diagram rules in Table 5-4 (Section 5.4.2). Visio Professional does not force entity types to participate in a relationship so completeness rules 4 and 5 are not enforced. Visio Professional allows attribute names in a subtype to have the same names as attributes in a parent entity type of a category so consistency rule 3 requiring unique attribute names is not enforced. For consistency rule 9, forbidding redundant foreign keys, Visio Professional requires that a foreign key attribute be defined in an entity type to specify a participating relationship. Visio Professional optionally displays foreign keys.

5.5.5 ERD Notation in Visual Paradigm

Visual Paradigm (www.visual-paradigm.com) supports entity relationship diagrams in addition to diagrams for project management, enterprise architecture, system modeling, business modeling, user interface design, and Agile requirements. For database design, Visual Paradigm supports forward and reverse engineering for a range of DBMSs.

The ERD notation in Visual Paradigm provides a variation of the Crow's Foot notation along with support for generalization hierarchies. Visual Paradigm supports M-N relationships but does not allow attributes so Figure 5.31 shows an associative entity type (*Enrollment*) instead of an M-N relationship. For other notation differences, Visual Paradigm does not have a symbol for weak entity types, always displays foreign keys, and displays only a single cardinality symbol when identical. Visual Paradigm does not support inheritance, but it supports names and participation constraints (completeness and disjointness) for generalization hierarchies. In Figure 5.31, the generalization hierarchy (*UnivPersonHierarchy*) shows complete and overlapping participation for the *Student* and *Faculty* entity types.

5.5.6 Class Diagram Notation of the Unified Modeling Language

The Unified Modeling Language has become the standard notation for object-oriented modeling. Object-oriented modeling emphasizes objects rather than processes, as emphasized in traditional systems development approaches. In object-oriented modeling, one defines the objects first, followed by the features (attributes and operations) of the objects, and then the dynamic interaction among objects. The UML contains class diagrams, interface diagrams, and interaction diagrams to support object-oriented modeling. The class diagram notation provides an alternative to the ERD notations presented in this chapter.

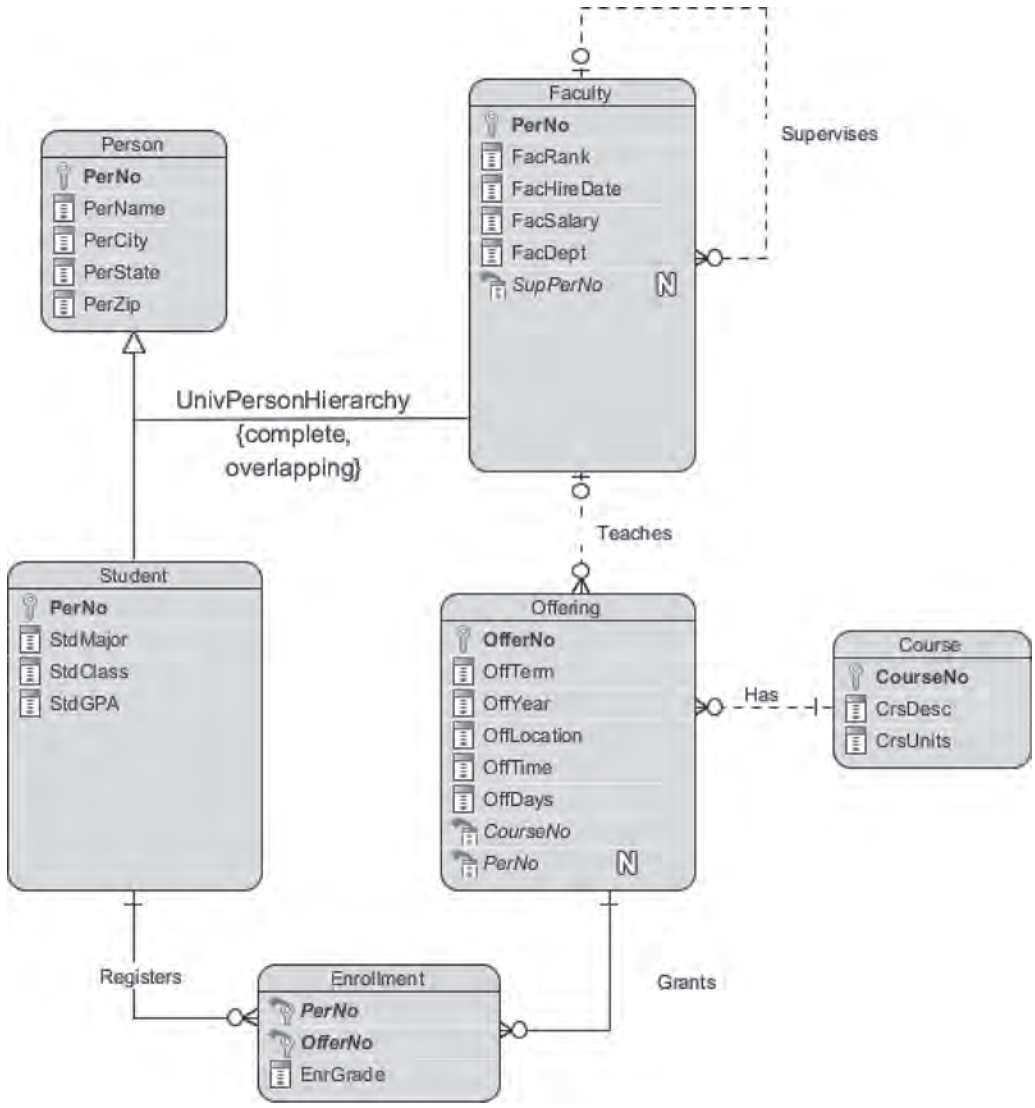


FIGURE 5.31
Extended University Database ERD in Visual Paradigm

Class diagrams contain classes (collections of objects), associations (binary relationships) among classes, and object features (attributes and operations). Figure 5.32 shows a simple class diagram containing the *Offering* and *Faculty* classes. The association in Figure 5.32 represents a 1-M relationship. The UML supports role names and cardinalities (minimum and maximum) for each direction in an association. The 0..1 cardinality means that an offering object can be related to a minimum of zero faculty objects and a maximum of one faculty object. Operations are listed below the attributes. Each operation contains a parenthesized list of parameters along with the data type returned by the operation.

Associations in the UML are similar to relationships in the Crow's Foot notation. Associations can represent binary or unary relationships. To represent an M-way relationship, a class and a collection of associations are required. To represent an M-N relationship with attributes, the UML provides the association class to allow associations to have attributes and operations. Figure 5.33 shows an association class that represents an M-N relationship between the *Student* and the *Offering* classes. The association class contains the relationship attributes.

Unlike most ERD notations, the UML supported generalization from its inception. Most ERD notations added generalization support as an additional feature after a notation was well established. In Figure 5.34, the large empty arrow denotes a classification

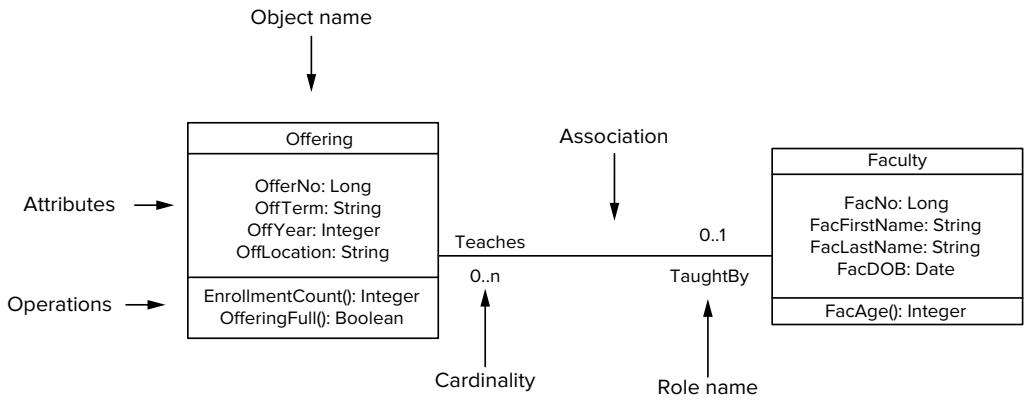


FIGURE 5.32
Simple Class Diagram

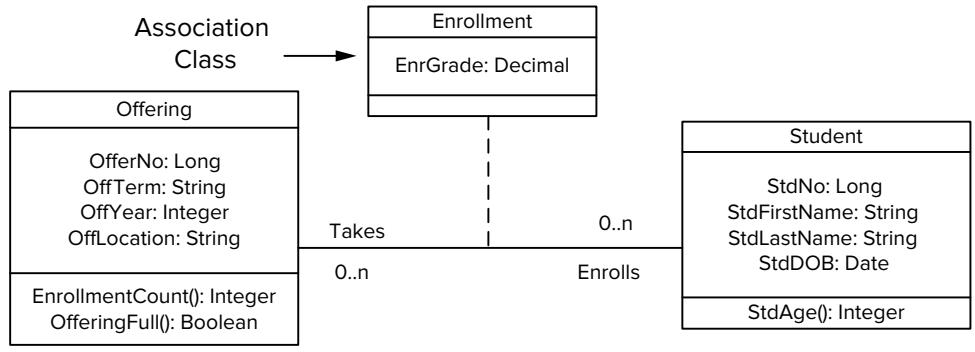


FIGURE 5.33
Association Class Representing an M-N Relationship with Attributes

of the *Student* class into *Undergraduate* and *Graduate* classes. The UML supports generalization names and constraints. In Figure 5.34, the *Status* generalization is complete, meaning that every student must be an undergraduate or a graduate student.

The UML also provides a special symbol for composition relationships, similar to identification dependencies in ERD notations. In a composition relationship, the objects in a child class belong only to objects in the parent class. In Figure 5.35, each *OrdLine* object belongs to one *Order* object. Deletion of a parent object causes deletion of the related child objects. Therefore, child objects usually borrow part of their primary key from the parent object. However, the UML does not require this identification dependency.

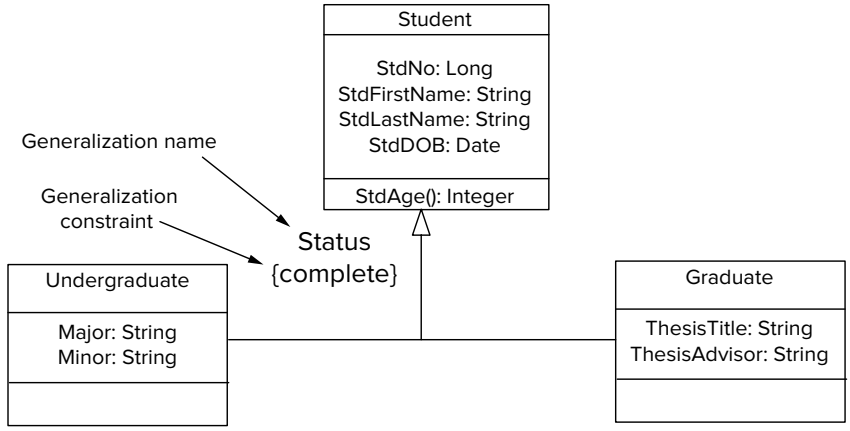
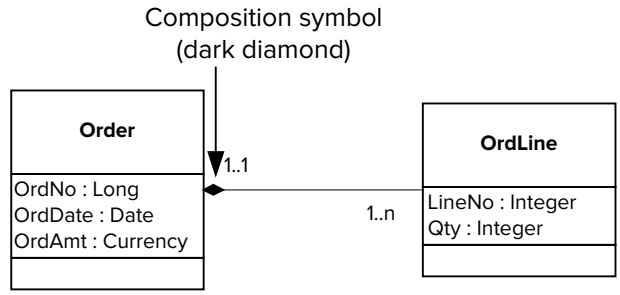


FIGURE 5.34
Class Diagram with a Generalization Relationship

FIGURE 5.35
Class Diagram with a Composition Relationship



UML class diagrams provide many other features not presented in this brief overview. The UML supports different kinds of classes to integrate programming language concerns with data modeling concerns. Other kinds of classes include value classes, stereotype classes, parameterized classes, and abstract classes. For generalization, the UML supports additional constraints such as static and dynamic classification and different interpretations of generalization relationships (subtype and subclass). For data integrity, the UML supports the specification of constraints in a class diagram.

You should note that class diagrams are just one part of the UML. To some extent, class diagrams must be understood in the context of object-oriented modeling and the entire UML. You should expect to devote an entire academic term to understanding object-oriented modeling and the UML.

CLOSING THOUGHTS

This chapter has explained the notation of entity relationship diagrams as a prerequisite to applying entity relationship diagrams in the database development process. Using the Crow's Foot notation, this chapter described the symbols, important relationship patterns, and generalization hierarchies. The basic symbols are entity types, relationships, and attributes along with cardinalities to depict the number of entities participating in a relationship. Four important relationship patterns were described: many-to-many (M-N) relationships with attributes, associative entity types representing M-way relationships, identifying relationships providing primary keys to weak entity types, and self-referencing (unary) relationships. Generalization hierarchies allow classification of entities to depict similarities among entity types.

To help improve your usage of the Crow's Foot notation, business rule representations, diagram rules, and comparisons to other notations were presented. This chapter presented formal and informal representation of business rules in an entity relationship diagram to provide an organizational context for entity relationship diagrams. The diagram rules involve completeness and consistency requirements. The diagram rules ensure that an ERD does not contain obvious errors. To broaden your background of ERD notations, this chapter presented common diagram variations in several CASE tools as well as the Class Diagram notation of the Unified Modeling Language, the standard notation for object-oriented modeling.

This chapter emphasized the notation of ERDs to provide a solid foundation for the more difficult study of applying the notation on business problems. To master data modeling, you need to understand the ERD notation and obtain ample practice building ERDs. Chapter 6 emphasizes the practice of building ERDs for business problems. Applying the notation involves consistent and complete representation of user requirements, generation of alternative designs, and documentation of design decisions. In addition to these skills, Chapter 6 presents rules to convert an ERD into a table design. With careful study, Chapters 5 and 6 provide a solid foundation to perform data modeling on business databases.

REVIEW CONCEPTS

- Basic concepts: entity types, relationships, and attributes
- Minimum and maximum cardinalities to constrain relationship participation
- Classification of cardinalities as optional, mandatory, and functional
- Existence dependency for entities that cannot be stored without storage of related entities
- Informal natural language correspondence for entity types (common nouns) and relationships (transitive verbs with direct objects)
- Identification dependency involving weak entity types and identifying relationships to support entity types that borrow at least part of their primary keys
- M-N relationships with attributes: attributes are associated with the combination of entity types, not just with one of the entity types
- Equivalence between an M-N relationship and an associative entity type with identifying 1-M relationships
- M-way associative entity types to represent M-way relationships among more than two entity types
- Self-referencing (unary) relationships to represent associations among entities of the same entity type
- Instance diagrams to help distinguish between 1-M and M-N self-referencing relationships
- Generalization hierarchies to show similarities among entity types
- Representation of business rules in an ERD: entity identification, connections among business entities, number of related entities, inclusion among entity sets, reasonable values, and data collection completeness
- Diagram rules to prevent obvious data modeling errors
- Common sources of diagram errors: identification dependency and redundant foreign keys
- Support for the diagram rules in data modeling tools through diagram construction and analysis tools
- ERD variations: symbols and diagram rules
- Modified Crow's Foot notation in the data modeling tools of Aqua Data Studio, Oracle SQL Developer, Visio 2010 Professional Edition, and Visual Paradigm
- Class Diagram notation of the Unified Modeling Language as an alternative to the Entity Relationship Model

QUESTIONS

1. What is an entity type?
2. What is an attribute?
3. What is a relationship?
4. What is the natural language correspondence for entity types and relationships?
5. What is the difference between an ERD and an instance diagram?
6. What symbols are the ERD counterparts of foreign keys in the Relational Model?
7. What cardinalities indicate functional, optional, and mandatory relationships?
8. When is it important to convert an M-N relationship into 1-M relationships?

9. How can an instance diagram help to determine whether a self-referencing relationship is a 1-M or an M-N relationship?
10. When should an ERD contain weak entity types?
11. What is the difference between an existence-dependent and a weak entity type?
12. Why is classification important in business?
13. What is inheritance in generalization hierarchies?
14. What is the purpose of disjointness and completeness constraints for a generalization hierarchy?
15. What symbols are used for cardinality in the Crow's Foot notation?
16. What are the two components of identification dependency?
17. How are M-way relationships represented in the Crow's Foot notation?
18. What is an associative entity type?
19. What is the equivalence between an M-N relationship and 1-M relationships?
20. What does it mean to say that part of a primary key is borrowed?
21. What is the purpose of the diagram rules?
22. What are the limitations of the diagram rules?
23. What consistency rules are commonly violated by novice data modelers?
24. Why do novice data modelers violate the identification dependency rules (consistency rules 6 through 8)?
25. Why do novice data modelers violate consistency rule 9 about redundant foreign keys?
26. Why should a CASE tool support diagram rules?
27. How does a data modeling tool typically support consistency rules 4 and 5?
28. How can a data modeling tool support all rules except consistency rules 4 and 5?
29. Why should an analysis tool not require resolution of all diagram errors found in an ERD?
30. How can an analysis tool implement consistency rule 9 about redundant foreign keys?
31. List some symbol differences in ERD notation that you may experience in your career.
32. List some diagram rule differences in ERD notation that you may experience in your career.
33. What is the Unified Modeling Language (UML)?
34. What are the modeling elements in a UML class diagram?
35. What kinds of business rules are formally represented in the Crow's Foot ERD notation?
36. What kinds of business rules are defined through informal documentation in the absence of a rules language for an ERD?
37. How are M-way relationships represented in the Crow's Foot notation?
38. What is a self-referencing relationship?
39. What tool can be useful to distinguish between a 1-M and M-N self-referencing relationship?
40. Please explain the importance of specialized modeling elements including M-way relationships and self-referencing relationships.
41. What is the difference between a weak entity type and an associative entity type?
42. What are the differences between the basic Crow's Foot notation (without generalization support) and the notation supported in the data modeling tool of Aqua Data Studio?

43. What are the differences between the basic Crow's Foot notation (without generalization support) and the notation supported in the data modeling tool of Oracle SQL Developer?
44. What diagram rules presented in Section 5.4.2 are supported by the data modeling tool of the Oracle SQL Developer?
45. What are the differences between the basic Crow's Foot notation and the notation supported in the Entity Relationship stencil of Visio Professional?
46. What diagram rules presented in Section 5.4.2 are supported by the Entity Relationship stencil in Visio Professional?
47. What are the differences between the basic Crow's Foot notation and the ERD notation supported in Visual Paradigm?
48. Which ERD notation in Section 5.5 is closest to the Crow's Foot notation used in Chapter 5? Justify your answer.
49. Which ERD notations in Section 5.5 support inheritance?
50. Which ERD notations in Section 5.5 support participation constraints (complete, disjoint) for generalization hierarchies?

PROBLEMS

The problems emphasize correct usage of the Crow's Foot notation and application of the diagram rules. This emphasis is consistent with the pedagogy of the chapter. The more challenging problems in Chapter 6 emphasize user requirements, diagram transformations, design documentation, and schema conversion. To develop a good understanding of data modeling, you should complete the problems in both chapters.

1. Draw an ERD containing the *Order* and *Customer* entity types connected by a 1-M relationship from *Customer* to *Order*. Choose an appropriate relationship name using your common knowledge of interactions between customers and orders. Define minimum cardinalities so that an order is optional for a customer and a customer is mandatory for an order. For the *Customer* entity type, add attributes *CustNo* (primary key), *CustFirstName*, *CustLastName*, *CustStreet*, *CustCity*, *CustState*, *CustZip*, and *CustBal* (balance). For the *Order* entity type, add attributes for the *OrdNo* (primary key), *OrdDate*, *OrdName*, *OrdStreet*, *OrdCity*, *OrdState*, and *OrdZip*. If you are using a data modeling tool that supports data type specification, choose appropriate data types for the attributes based on your common knowledge.
2. Extend the ERD from problem 1 with the *Employee* entity type and a 1-M relationship from *Employee* to *Order*. Choose an appropriate relationship name using your common knowledge of interactions between employees and orders. Define minimum cardinalities so that an employee is optional to an order and an order is optional to an employee. For the *Employee* entity type, add attributes *EmpNo* (primary key), *EmpFirstName*, *EmpLastName*, *EmpPhone*, *EmpEmail*, *EmpCommRate* (commission rate), and *EmpDeptName*. If you are using a data modeling tool that supports data type specification, choose appropriate data types for the attributes based on your common knowledge.
3. Extend the ERD from problem 2 with a self-referencing 1-M relationship involving the *Employee* entity type. Choose an appropriate relationship name using your common knowledge of organizational relationships among employees. Define minimum cardinalities so that the relationship is optional in both directions.
4. Extend the ERD from problem 3 with the *Product* entity type and an M-N relationship between *Product* and *Order*. Choose an appropriate relationship

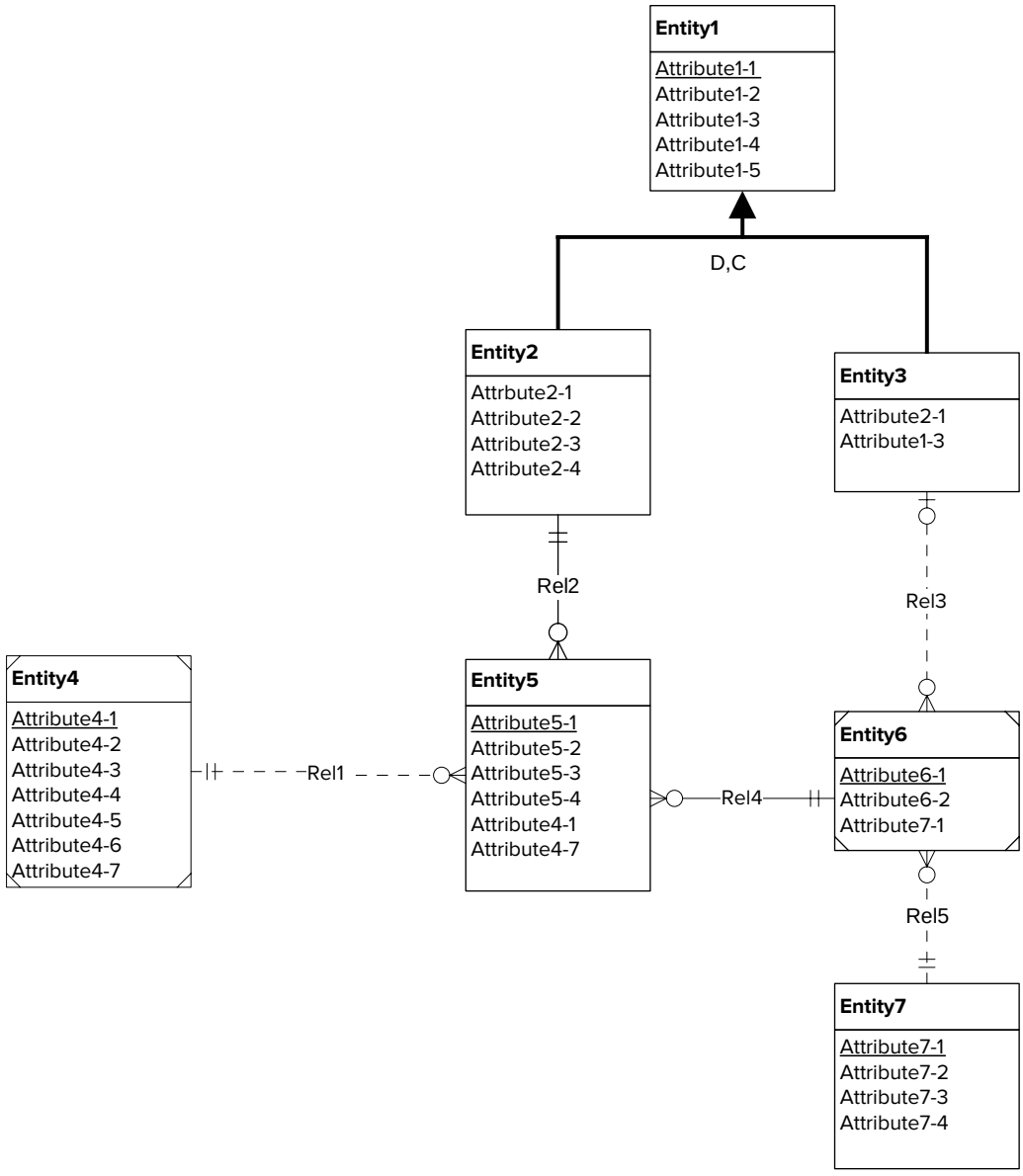
name using your common knowledge of connections between products and orders. Define minimum cardinalities so that an order is optional to a product, and a product is mandatory to an order. For the *Product* entity type, add attributes *ProdNo* (primary key), *ProdName*, *ProdQOH*, *ProdPrice*, and *ProdNextShipDate*. For the M-N relationship, add an attribute for the order quantity. If you are using a data modeling tool that supports data type specification, choose appropriate data types for the attributes based on your common knowledge.

5. Revise the ERD from problem 4 by transforming the M-N relationship into an associative entity type and two identifying, 1-M relationships.
6. Check your ERDs from problems 4 and 5 for violations of the diagram rules. If you followed the problem directions, your diagrams should not have any errors.
7. Using your corrected ERD from problem 6, add violations of consistency rules 6 to 9.
8. Design an ERD for the *Task* entity type and an M-N self-referencing relationship. For the *Task* entity type, add attributes *TaskNo* (primary key), *TaskDesc*, *TaskEstDuration*, *TaskStatus*, *TaskStartTime*, and *TaskEndTime*. Choose an appropriate relationship name using your common knowledge of precedence connections among tasks. Define minimum cardinalities so that the relationship is optional in both directions.
9. Revise the ERD from problem 8 by transforming the M-N relationship into an associative entity type and two identifying, 1-M relationships.
10. Define a generalization hierarchy containing the *Student* entity type, the *UndStudent* entity type, and the *GradStudent* entity type. The *Student* entity type is the supertype and *UndStudent* and *GradStudent* are subtypes. The *Student* entity type has attributes *StdNo* (primary key), *StdName*, *StdGender*, *StdDOB* (date of birth), *StdEmail*, and *StdAdmitDate*. The *UndStudent* entity type has attributes *UndMajor*, *UndMinor*, and *UndClass*. The *GradStudent* entity type has attributes *GradAdvisor*, *GradThesisTitle*, and *GradAsstStatus* (assistantship status). The generalization hierarchy should be complete and disjoint.
11. Define a generalization hierarchy containing the *Employee* entity type, the *Faculty* entity type, and the *Administrator* entity type. The *Employee* entity type is the supertype and *Faculty* and *Administrator* are subtypes. The *Employee* entity type has attributes *EmpNo* (primary key), *EmpName*, *EmpGender*, *EmpDOB* (date of birth), *EmpPhone*, *EmpEmail*, and *EmpHireDate*. The *Faculty* entity type has attributes *FacRank*, *FacPayPeriods*, and *FacTenure*. The *Administrator* entity type has attributes *AdmTitle*, *AdmContractLength*, and *AdmAppointmentDate*. The generalization hierarchy should be complete and overlapping.
12. Combine the generalization hierarchies from problems 10 and 11. The root of the generalization hierarchy is the *UnivPerson* entity type. The primary key of *UnivPerson* is *UnvPerNo*. The other attributes in the *UnivPerson* entity type should be the attributes common to *Employee* and *Student*. You should rename the attributes to be consistent with inclusion in the *UnivPerson* entity type. The generalization hierarchy should be complete and disjoint.
13. Draw an ERD containing the *Patient*, *Physician*, and *Visit* entity types connected by 1-M relationships from *Patient* to *Visit* and *Physician* to *Visit*. Choose appropriate names for the relationships. Define minimum cardinalities so that patients and physicians are mandatory for a visit, but visits are optional for patients and physicians. For the *Patient* entity type, add attributes *PatNo* (primary key), *PatFirstName*, *PatLastName*, *PatStreet*, *PatCity*, *PatState*, *PatZip*, and *PatHealthPlan*. For the *Physician* entity type, add attributes *PhyNo* (primary key), *PhyFirstName*, *PhyLastName*, *PhySpecialty*, *PhyPhone*, *PhyEmail*, *PhyHospital*, and *PhyCertification*. For the *Visit* entity type, add attributes for the *VisitNo* (primary key), *VisitDate*, *VisitPayMethod* (cash, check, or credit card), and *VisitCharge*. If

you are using a data modeling tool that supports data type specification, choose appropriate data types for the attributes based on your common knowledge.

14. Extend the ERD in problem 13 with the *Nurse*, the *Item*, and the *VisitDetail* entity types connected by 1-M relationships from *Visit* to *VisitDetail*, *Nurse* to *VisitDetail*, and *Item* to *VisitDetail*. *VisitDetail* is a weak entity with the 1-M relationship from *Visit* to *VisitDetail* an identifying relationship. Choose appropriate names for the relationships. Define minimum cardinalities so that a nurse is optional for a visit detail, an item is mandatory for a visit detail, and visit details are optional for nurses and items. For the *Item* entity type, add attributes *ItemNo* (primary key), *ItemDesc*, *ItemPrice*, and *ItemType*. For the *Nurse* entity type, add attributes *NurseNo* (primary key), *NurseFirstName*, *NurseLastName*, *NurseTitle*, *NursePhone*, *NurseSpecialty*, and *NursePayGrade*. For the *VisitDetail* entity type, add attributes for the *DetailNo* (part of the primary key) and *DetailCharge*. If you are a design tool that supports data type specification, choose appropriate data types for the attributes based on your common knowledge.
15. Refine the ERD from problem 14 with a generalization hierarchy consisting of *Provider*, *Physician*, and *Nurse*. The root of the generalization hierarchy is the *Provider* entity type. The primary key of *Provider* is *ProvNo* replacing the attributes *PhyNo* and *NurseNo*. The other attributes in the *Provider* entity type should be the attributes common to *Nurse* and *Physician*. You should rename the attributes to be consistent with inclusion in the *Provider* entity type. The generalization hierarchy should be complete and disjoint.
16. Check your ERD from problem 15 for violations of the diagram rules. If you followed the problem directions, your diagram should not have any errors. Apply the consistency and completeness rules to ensure that your diagram does not have errors.
17. Using your corrected ERD from problem 16, add violations of consistency rules 3 and 6 to 9.
18. For each consistency error in Figure 5.P1, identify the consistency rule violated and suggest possible resolutions of the error. The ERD has generic names so that you will concentrate on finding diagram errors rather than focusing on the meaning of the diagram.
19. For each consistency error in Figure 5.P2, identify the consistency rule violated and suggest possible resolutions of the error. The ERD has generic names to help you concentrate on finding diagram errors rather than focusing on the meaning of the diagram.
20. For each consistency error in Figure 5.P3, identify the consistency rule violated and suggest possible resolutions of the error. The ERD has generic names to help will concentrate on finding diagram errors rather than focusing on the meaning of the diagram.
21. Draw an ERD containing the *Employee* and *Appointment* entity types connected by an M-N relationship. Choose an appropriate relationship name using your common knowledge of interactions between employees and appointments. Define minimum cardinalities so that an appointment is optional for an employee and an employee is mandatory for an appointment. For the *Employee* entity type, add attributes *EmpNo* (primary key), *EmpFirstName*, *EmpLastName*, *EmpPosition*, *EmpPhone*, and *EmpEmail*. For the *Appointment* entity type, add attributes for *AppNo* (primary key), *AppSubject*, *AppStartTime*, *AppEndTime*, and *AppNotes*. For the M-N relationship, add an attribute *Attendance* indicating whether the employee attended the appointment.
22. Extend the ERD from problem 21 with the *Location* entity type and a 1-M relationship from *Location* to *Appointment*. Choose an appropriate relationship name using your common knowledge of interactions between locations and

FIGURE 5.P1
ERD for Problem 18



appointments. Define minimum cardinalities so that a location is optional for an appointment and an appointment is optional for a location. For the *Location* entity type, add attributes *LocNo* (primary key), *LocBuilding*, *LocRoomNo*, and *LocCapacity*.

23. Extend the ERD in problem 22 with the *Calendar* entity type and an M-N relationship from *Appointment* to *Calendar*. Choose an appropriate relationship name using your common knowledge of interactions between appointments and calendars. Define minimum cardinalities so that an appointment is optional for a calendar and a calendar is mandatory for an appointment. For the *Calendar* entity type, add attributes *CalNo* (primary key), *CalDate*, and *CalHour*.
24. Revise the ERD from problem 23 by transforming the M-N relationship between *Employe*e and *Appointment* into an associative entity type along with two identifying 1-M relationships.
25. Draw an ERD containing *Student* and *Paper* entity types connected by 1-M relationships. The Student entity type should have attributes for *StdNo*

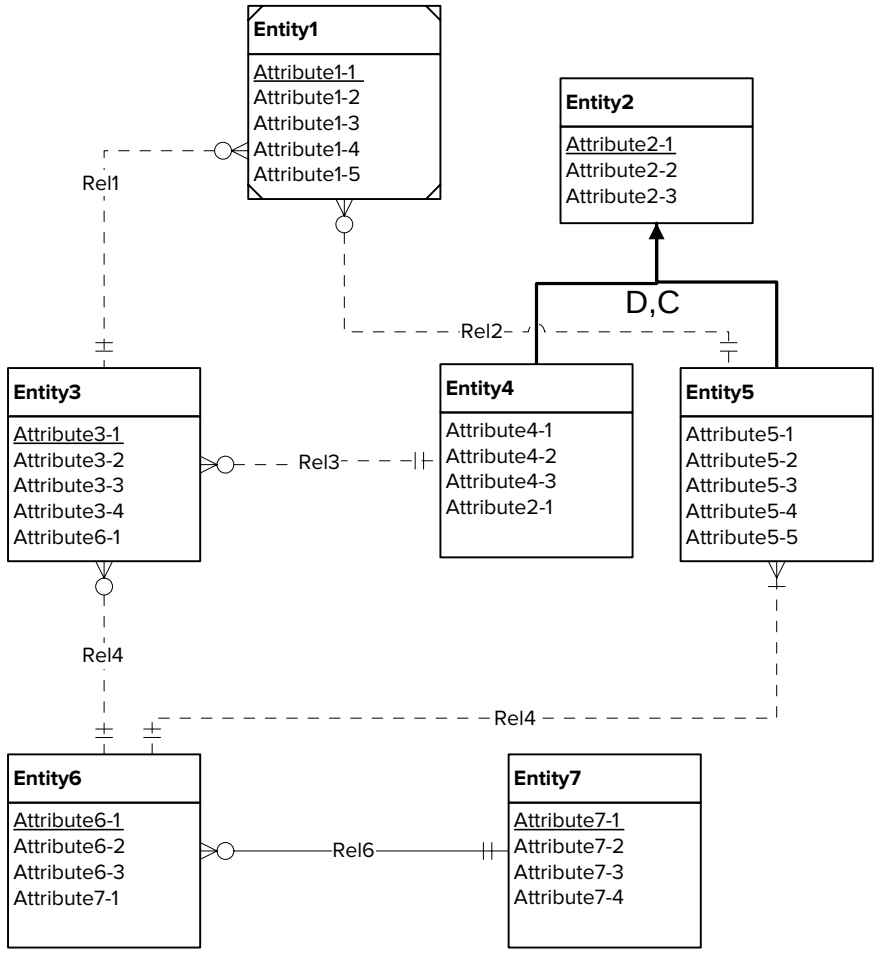


FIGURE 5.P2
ERD for Problem 19

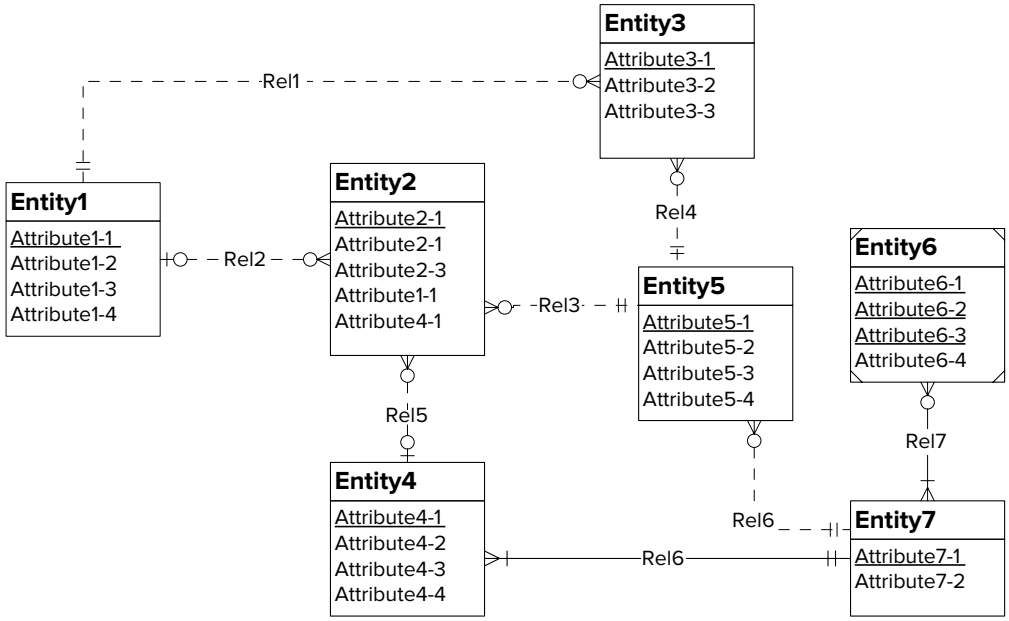


FIGURE 5.P3
ERD for Problem 20

(primary key), *StdFirstName*, *StdLastName*, *StdAdmitSemester*, *StdAdmitYear*, and *StdEnrollStatus* (full or part-time). The Paper entity type should have attributes for *PaperNo* (primary key), *PaperTitle*, *PaperSubmitDate*, *PaperAccepted* (yes or no), and *PaperType* (first, second, proposal, or dissertation). Add a 1-M relationship from *Student* to *Paper*.

26. Extend the ERD with an *Evaluator* entity type and an M-N relationship between *Paper* and *Evaluator*. The Evaluator entity type should have attributes for *EvalNo* (primary key), *EvalFirstName*, *EvalLastName*, *EvalEmail*, and *EvalOffice*. The M-N relationship should have attributes for *EvalDate*, *EvalLitReview* (1 to 5 rating), *EvalProbId* (1 to 5 rating), *EvalTechWriting* (1 to 5 rating), *EvalModelDev* (1 to 5 rating), *EvalOverall* (1 to 5 rating), and *EvalComments*.
27. Transform the M-N relationship from problem 26 into an associative entity type and identifying relationships.
28. Transform the M-N relationship from problem 26 into three 1-M relationships from Evaluator to Paper. Each paper can have up to three evaluations. Each relationship should be optional to both evaluators and papers. The five evaluation attributes should be associated with each 1-M relationship.
29. Consider data modeling choices to represent student classifications by program year (freshman, sophomore, junior, and senior) and degree type (undergraduate, masters, and doctorate). Identify data modeling alternatives to support classification of students by program year and degree type. What additional information would you need to decide on the appropriate representation?
30. Construct an ERD to represent employees and positions. For employees, the ERD should record the unique employee number, first name, last name, department name, office number, hire date, and date of birth. For position, the ERD should record the unique position number, position name, and step number. Positions can be classified as management, associate, or other. Management positions have a salary range (minimum and maximum) and a car allowance amount. Associate positions have an hourly rate range. There are no attributes to record for other position types. An employee must hold one position. A position can be held by many employees.
31. Draw an ERD containing the *Lab*, *LabVisit*, and *Patient* entity types connected by 1-M relationships from *Lab* to *LabVisit* and *Patient* to *LabVisit*. Choose appropriate relationship names using your common knowledge of interactions between labs, lab visits, and patients. Define minimum cardinalities so that a lab is required for a lab visit and a patient is required for a lab visit. For the *Lab* entity type, add attributes *LabNo* (primary key), *LabName*, *LabStreet*, *LabCity*, *LabState*, and *LabZip*. For the *Patient* entity type, add attributes *PatNo* (primary key), *PatLastName*, *PatFirstName*, *PatDOB* (date of birth). For the *LabVisit* entity type, add attributes for the *LVNo* (primary key), *LVDate*, *LVProvNo*, and optional *LVOrdNo* (for orders from physicians). If you are using a data modeling tool that supports data type specification, choose appropriate data types for the attributes based on your common knowledge.
32. Augment your ERD from problem 31 with the *Specimen* entity type. For each specimen collected, the database should record a unique *SpecNo*, *SpecArea* (vaginal, cervical, or endocervical), and *SpecCollMethod* (thin prep or sure path). A 1-M relationship from *LabVisit* to *Specimen*. A lab visit must produce at least one specimen. A specimen is associated with exactly one lab visit.
33. Augment your ERD from problem 32 with the *TestOrder* entity type and a relationship between *TestOrder* and *Specimen*. Multiple test orders can be created for a specimen, but a specimen does not have a test order until a delay, from hours to days. A test order is created for exactly one specimen. A test order contains a *TONo* (primary key), *TOTestName*, *TOTestType* (HPV, CT/GC, CT, or GC), and *TOTestResult* (positive, negative, equivocal, or failure). If a test order

produces a failure, the specimen is given a new test order and tested again until a non-failure result is obtained.

34. Augment your ERD from problem with the *Supply* entity type and a relationship between *TestOrder* and *Supply*. A test can use a collection of supplies (0 or more) and a supply can be used on a collection of tests (0 or more). The *Supply* entity type contains *SuppNo* (primary key), *SuppName*, *SuppLotNo*, and *SuppQOH*.
35. Draw an ERD containing the *Student*, *DegreePlan*, and *Faculty* entity types connected by 1-M relationships from *Student* to *DegreePlan* and *Faculty* to *DegreePlan*. A student submits a degree plan for approval to a faculty adviser. Choose appropriate relationship names using your common knowledge of interactions among students, degree plans, and faculty. Define minimum cardinalities so that both a student and faculty are required for a degree plan. For the *Student* entity type, add attributes *StdNo* (primary key), *StdFirstName*, *StdLastName*, *StdStreet*, *StdCity*, *StdState*, *StdZip*, and *StdAdmitSems*. For the *Faculty* entity type, add attributes *FacNo* (primary key), *FacLastName*, *FacFirstName*, and *FacRank*. For the *DegreePlan* entity type, add attributes *DPNo* (primary key), *DPRevDate*, and *DPComments*. If you are using a data modeling tool that supports data type specification, choose appropriate data types for the attributes based on your common knowledge.
36. Add a *Course* entity type and an M-N relationship (*Enrolls*) connecting *Course* and *DegreePlan*. A degree plan must have at least one associated course. For the *Course* entity type, add attributes for *CrsNo* (primary key), *CrsTitle*, and *CrsUnits*. For the *Enrolls* relationship, add attributes *EnrSems*, *EnrYear*, and *EnrGrade*. Check your diagram for errors and correct errors before completing your solution.
37. Add a *Milestone* entity type and an M-N relationship (*Achieves*) connecting *Milestone* and *DegreePlan*. A degree plan must have at least one milestone. For the *Milestone* entity type, add attributes *MSNo*, *MSName*, *MSDeadline*, and *MSDesc*. For the *Achieves* relationship, you should record *AchDateSubmitted*, *AchDateApproved*, and *AchResult*. Check your diagram for errors and correct errors before completing your solution.
38. Convert the *Achieves* relationship in problem 37 to an associative entity type (*Achievement*) with an M-N relationship (*Evaluates*) from *Faculty* to *Achievement*. An achievement must be associated with at least one faculty member. In the *Evaluates* relationship, add *EvalRole* (advisor, chair, or committee member) as an attribute. Check your diagram for errors and correct errors before completing your solution.

REFERENCES FOR FURTHER STUDY

Four specialized books on database design are Batini, Ceri, and Navathe (1992), Nijssen and Halpin (1989), Teorey et al. (2005), and Carlis and Maguire (2001). The DevX Database Development Zone (www.devx.com) has practical advice about database development and data modeling. If you would like more details about the UML, consult the UML Resource Page of the Object Management Group (www.uml.org).

chapter

6

Developing Data Models for Business Databases

Learning Objectives

This chapter extends your knowledge of data modeling from the notation of entity relationship diagrams (ERDs) to the development of data models for business databases along with rules to convert entity relationship diagrams to relational tables. After this chapter, the student should have acquired the following knowledge and skills:

- Develop ERDs that are consistent with narrative problems
- Use transformations to generate alternative ERDs
- Document design decisions implicit in an ERD
- Analyze an ERD for common design errors
- Convert an ERD to a table design using conversion rules

OVERVIEW

Chapter 5 explained the Crow's Foot notation for entity relationship diagrams. You learned about diagram symbols, relationship patterns, generalization hierarchies, and rules for consistency and completeness. Understanding the notation is a prerequisite for applying it to represent business databases. This chapter explains the development of data models for business databases using the Crow's Foot notation and rules to convert ERDs to table designs.

To become a good data modeler, you need to understand the notation in entity relationship diagrams and get plenty of practice building diagrams. This chapter provides practice with applying the notation. You will learn to analyze a narrative problem, refine a design through transformations, document important design decisions, and analyze a data model for common design errors. After finalizing an ERD, the diagram should be converted to a table design so that it can be implemented with a commercial DBMS. This chapter presents rules to convert an entity relationship diagram to a table design. You will learn about the basic rules to convert common parts of a diagram along with specialized rules for less common parts of an ERD.

With this background, you are ready to build ERDs for moderate-size business situations. You should have confidence in your knowledge of the Crow's Foot notation, ready to apply the notation to narrative problems and convert ERDs to table designs.

6.1 ANALYZING BUSINESS DATA MODELING PROBLEMS

After studying the Crow's Foot notation, you are now ready to apply your knowledge. This section presents guidelines to analyze information needs of business environments. The guidelines involve analysis of narrative problem descriptions as well as challenges of determining information requirements in unstructured business situations. After presenting the guidelines, this chapter applies them to develop an ERD for an example business data modeling problem.

6.1.1 Guidelines for Analyzing Business Information Needs

Data modeling involves the collection and analysis of business requirements resulting in an ERD to represent the requirements. Business requirements are rarely well structured. Rather, as an analyst you will often face an ill-defined business situation in which you need to add structure. You will need to interact with a variety of stakeholders who sometimes provide competing statements about the database requirements. In collecting requirements, you will conduct interviews, review documents and system documentation, and examine existing data. To determine the scope of the database, you will need to eliminate irrelevant details and add missing details. On large projects, you may work on a subset of the requirements and then collaborate with a team of designers to determine the complete data model.

These challenges make data modeling a stimulating and rewarding intellectual activity. A data model provides an essential element to standardize organizational vocabulary, enforce business rules, and ensure adequate data quality. Many users will experience the results of your efforts as they use a database on a daily basis. Because electronic data has become a vital corporate resource, your data modeling efforts can make a significant contribution to an organization's future success.

A textbook cannot provide the experience of designing real databases. The more difficult chapter problems and associated case studies on the textbook's website can provide insights into the difficulties of designing real databases but will not provide you with practice with the actual experience. To acquire this experience, you must interact with organizations through class projects, internships, and job experience. Thus, this chapter emphasizes the more limited goal of analyzing narrative problems as a step to developing data modeling skills for real business situations. Analyzing narrative problems will help you gain confidence in translating a problem statement into an ERD and identifying ambiguous and incomplete parts of problem statements.

The main goal when analyzing narrative problem statements is to create an ERD that is consistent with the narrative. The ERD should not contradict the implied ERD elements in the problem narrative. For example, if the problem statement indicates that entities are related by words indicating more than one, the ERD should have a cardinality of many to match that part of the problem statement. The remainder of this section and Section 6.3.2 provide more details about achieving a consistent ERD.

Goals of Narrative Problem Analysis
strive for a simple design that is consistent with the narrative. Be prepared to follow up with additional requirements collection and consideration of alternative designs.

In addition to the goal of consistency, you should have a bias toward simpler designs at least initially. For example, an ERD with one entity type is less complex than an ERD with two entity types and a relationship. In general, when a choice exists between two ERDs, you should choose the simpler design especially in the initial stages of the design process. As the design process progresses, you can add details and refinements to the original design. Section 6.2 provides a list of transformations that can help you to consider alternative designs.

Identifying Entity Types In a narrative, you should look for nouns representing people, things, places, and events as potential entity types. The nouns may appear as subjects or objects in sentences. For example, the sentence "Students take courses at the university," indicates that student and course may be entity types. You also should

look for nouns that have additional sentences describing their properties. The properties often indicate attributes of entity types. For example, the sentence "Students choose their major and minor in their first year," indicates that major and minor may be attributes of student. The sentence "Courses have a course number, semester, year, and room listed in the catalog," indicates that course number, semester, year, and room are attributes of the course entity type.

You should apply the simplicity principle during the search for entity types in the initial ERD, especially involving choices between attributes and entity types. Unless the problem description contains additional sentences or details about a noun, you should consider it initially as an attribute. For example, if courses have an instructor name listed in the catalog, you should consider instructor name as an attribute of the course entity type rather than as an entity type unless additional details are provided about instructors in the problem statement. If there is confusion between considering a concept as an attribute or entity type, you should follow-up with more requirements collection later.

Determining Primary Keys Determination of primary keys is an important part of entity type identification. Ideally, primary keys should be stable and single purpose. *Stable* means that a primary key should never change after it has been assigned to an entity. For example, phone numbers, email addresses, and names are not good choices for primary keys because they are not stable. *Single purpose* means that a primary key should have no purpose other than entity identification. For example, phone numbers and bank routing numbers are not good choices because they contain location information. Typically, good choices for primary keys are integer values automatically generated by a DBMS. For example, Access has the AutoNumber data type for primary keys and Oracle has Identity columns for primary keys.

You should evaluate proposed primary keys using stability and single purpose as guidelines. If a proposed primary key does not meet either criterion, you should probably reject it as a primary key. If a proposed primary key only meets one criterion, you should explore other attributes for the primary key. Sometimes, industry or organizational practices dictate the choice of a primary key even if the choice is not ideal. For example, email addresses and phone numbers, although not ideal, are sometimes used as primary keys because customers can provide them.

Due to privacy concerns, government created identifiers should be avoided as primary keys. In the U.S., Social Security numbers (SSNs) are widely used for tax purposes and government benefits. Thus, SSNs should be avoided as primary keys. Many organizations prohibit usage of SSNs as primary keys. However, storage of government identifiers such as SSNs may be necessary for compliance with government reporting requirements especially in financial service databases. In addition to privacy concerns, SSNs violate the single-purpose tenet as the first three digits in an SSN identify a geographic region.[1] Another problem with SSNs is fraudulent sharing of numbers as well as creating false numbers. Thus, in a large database, duplicate SSNs and invalid SSNs typically exist.

Besides primary keys, you should also identify other unique attributes (candidate keys). For example, an employee's email address is often unique. The integrity of candidate keys may be important for searching and integration with external databases. Depending on the features of the ERD drawing tool that you are using, you should note that an attribute is unique either in the attribute specification or in free-format documentation. Uniqueness constraints can be enforced after an ERD is converted to a table design.

[1] Prior to 1972, the first three digits (area number) of an SSN indicated the US state of the local office issuing the number. Since 1972, the area number indicates the US residence state on the original application for the SSN.

Adding Relationships Relationships often appear as transitive verbs connecting nouns previously identified as entity types. For example, the sentence, "Students enroll in courses each semester," indicates a relationship between students and courses. For relationship cardinality, you should look at the number (singular or plural) of nouns along with other words that indicate cardinality. For example, the sentence, "A course offering is taught by an instructor," indicates that there is one instructor per course offering. You should also look for words such as "collection" and "set" that indicate a maximum cardinality of more than one. For example, the sentence, "An order contains a collection of items," indicates that an order is related to multiple items. Minimum cardinality can be indicated by words such as "optional" and "required." In the absence of indication of minimum cardinality, the default should be mandatory. Additional requirements collection should be conducted to confirm default choices.

You should be aware that indications of relationships in problem statements may lead to direct or indirect connections in an ERD. A direct connection involves a relationship between entity types. An indirect connection involves a connection through other entity types and relationships. For example, the sentence, "An advisor counsels students about the choice of a major," may indicate direct or indirect relationships between advisor, student, and major.

To help with difficult choices between direct and indirect connections, you should look for entity types that are involved in multiple relationships. These entity types can reduce the number of relationships in an ERD by being placed in the center as a hub connected directly to other entity types as spokes of a wheel. Entity types derived from important documents (orders, registrations, purchase orders, etc.) are often hubs in an ERD. For example, an order entity type can be directly related to customer, employee, and product removing the need for direct connections among all entity types. These choices will be highlighted in the analysis of the water utility information requirements in the following section.

Summary of Analysis Guidelines When analyzing a narrative problem statement, you should develop an ERD that consistently represents the complete narrative. Given a choice among consistent ERDs, you should favor simpler rather than more complex designs, at least initially. You also should note ambiguities and incompleteness in the problem statement. The guidelines discussed in this section can help in your initial analysis of data modeling problems. Sections 6.2 and 6.3 present additional analysis methods to revise and finalize ERDs. To help you recall the guidelines discussed in this section, Table 6-1 presents a summary.

6.1.2 Analysis of Problem Narrative for the Water Utility Database

This section presents requirements for a customer database for a municipal water utility. You can assume that this description is the result of an initial investigation with appropriate personnel at the water utility. After the description, the guidelines presented in Section 6.1.1 are used to analyze the narrative description and develop an ERD.

Problem Narrative The water utility database supports recording of water usage and billing for water consumption. To support these functions, the database should contain data about customers, rates, water usage, and bills. Other functions such as payment processing and customer service are omitted in this description for brevity. The following list describes the data requirements in more detail.

- Customer data include a unique customer number, a name, a billing address, a type (commercial or residential), an applicable rate, and a collection (one or more) of meters.

TABLE 6-1
Summary of Analysis Guidelines for Narrative Problems

Diagram Element	Analysis Guidelines	ERD Effect
Entity type identification	Look for nouns used as subjects or objects along with additional details in other sentences.	Add entity types to the ERD. If a noun does not have supporting details, consider it as an attribute.
Primary key determination	Strive for stable and single-purpose attributes for primary keys. Narrative should indicate uniqueness. Avoid government issued identifiers as primary keys.	Specify primary and candidate keys.
Relationship (direct or indirect) detection	Look for transitive verbs that connect nouns identified as entity types.	Add a direct relationship between entity types or note that an indirect connection must exist between entity types.
Cardinality determination (maximum)	Look for singular or plural designation of nouns in sentences indicating relationships.	Specify cardinalities of 1 and M (many).
Cardinality determination (minimum)	Look for optional or required language in sentences. Set required as the default if problem statement does not indicate minimum cardinality.	Specify cardinalities of 0 (optional) and 1 (mandatory).
Relationship simplification	Look for hub entity types as nouns used in multiple sentences linked to other nouns identified as entity types.	An entity type hub has direct relationships with other entity types. Eliminate other relationships if an indirect connection exists through a hub entity type.

- Meter data include a unique meter number, an address, a size, and a model. The meter number is engraved on a meter before it is placed in service. A meter is associated with one customer at a time.
- An employee periodically reads each meter on a scheduled date. When a meter is read, a meter-reading document is created containing a unique meter reading number, an employee number, a meter number, a timestamp (includes date and time), and a consumption level. When a meter is first placed in service, there are no associated readings for it.
- A rate includes a unique rate number, a description, a fixed dollar amount, a consumption threshold, and a variable amount (dollars per cubic foot). Consumption up to the threshold is billed at the fixed amount. Consumption greater than the threshold is billed at the variable amount. Customers are assigned rates using a number of factors such as customer type, address, and adjustment factors. Many customers can be assigned the same rate. Rates are typically proposed months before approved and associated with customers.
- Water utility bills are based on customers' most recent meter readings and applicable rates. A bill consists of a heading part and a list of detail lines. The heading part contains a unique bill number, a customer number, a preparation date, a payment due date, and a date range for the consumption period. Each detail line contains a meter number, a water consumption level, and an amount. The water consumption level is computed by subtracting the consumption levels in the two most recent meter readings. The amount is computed by multiplying the consumption level by the customer's rate.

Identifying Entity Types and Primary Keys Prominent nouns in the narrative are customer, meter, bill, reading (for meter reading), and rate. For each of these nouns, the narrative describes associated attributes. Figure 6.1 shows a preliminary ERD with entity types for nouns and associated attributes. Note that collections of things are not attributes. For example, the fact that a customer has a collection of meters will be shown as a relationship, rather than as an attribute of the *Customer* entity type. In addition, references between these entity types will be shown as relationships rather

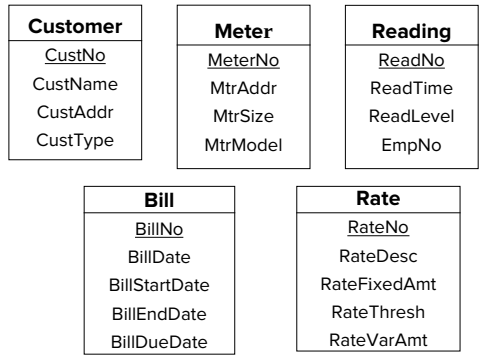


FIGURE 6.1
Preliminary Entity Types and Attributes for the Water Utility Database

than as attributes. For example, the fact that a reading contains a meter number will be recorded as a relationship.

The narrative specifically mentions uniqueness of customer number, meter number, reading number, bill number, and rate number. The bill number, reading number, and meter number seem stable and single purpose as they are imprinted on physical objects. Additional investigation should be conducted to determine if customer number and rate number are stable and single purpose. Since the narrative does not describe additional uses of these attributes, the initial assumption in the ERD is that these attributes are suitable as primary keys.

Adding Relationships After identifying entity types and attributes, the analysis continues by connecting entity types with relationships as shown in Figure 6.2. To reduce the drawing space of the ERD, Figure 6.2 only shows primary keys. The following list explains the derivation of relationships with a focus on parts of the narrative that indicate relationships among entity types.

- For the *Assigned* relationship, the narrative states that a customer has a rate, and many customers can be assigned the same rate. These two statements indicate a 1-M relationship from *Rate* to *Customer*. For the minimum cardinalities, the narrative indicates that a rate is required for a customer, and that rates are proposed before being associated with customers.
- For the *Uses* relationship, the narrative states that a customer includes a collection of meters and a meter is associated with one customer at a time. These two statements indicate a 1-M relationship from *Customer* to *Meter*. For the minimum cardinalities, the narrative indicates that a customer must have at least one meter. The narrative does not indicate the minimum cardinality for a meter so either 0 or 1 can be chosen. The documentation should note this incompleteness in the specifications.
- For the *ReadBy* relationship, the narrative states that a meter reading contains a meter number, and meters are periodically read. These two statements indicate

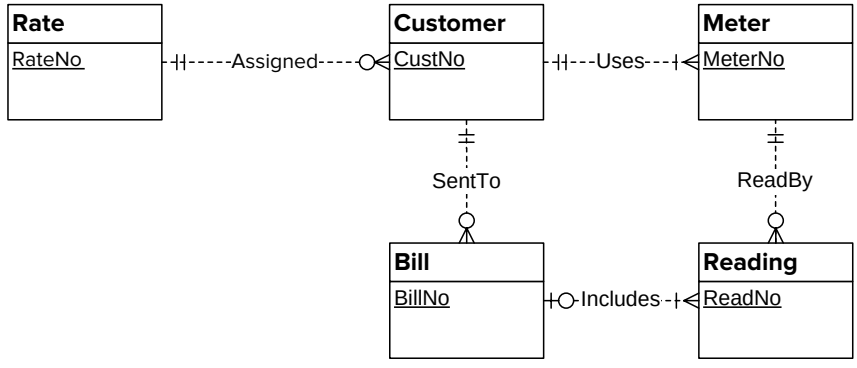


FIGURE 6.2
Entity Types Connected by Relationships

a 1-M relationship from *Meter* to *Reading*. For the minimum cardinalities, the narrative indicates that a meter is required for a reading, and a new meter does not have any associated readings.

- For the *SentTo* relationship, the narrative indicates that the heading part of a bill contains a customer number and bills are periodically sent to customers. These two statements indicate a 1-M relationship from *Customer* to *Bill*. For the minimum cardinalities, the narrative indicates that a customer is required for a bill, and a customer does not have an associated bill until the customer's meters are read.

The *Includes* relationship between the *Bill* and the *Reading* entity types is subtle. The *Includes* relationship is 1-M because a bill may involve a collection of readings (one on each detail line), and a reading relates to one bill. The consumption level and amount on a detail line are calculated values. The *Includes* relationship connects a bill to its most recent meter readings, thus supporting the consumption and amount calculations. These values can be stored if it is more efficient to store them rather than compute them when needed. If the values are stored, attributes can be added to the *Includes* relationship or the *Reading* entity type.

6.2 REFINEMENTS TO AN ERD

Data modeling is usually an iterative or repetitive process. You construct a preliminary data model and then refine it many times. In refining a data model, you should generate feasible alternatives and evaluate them according to user requirements. You typically need to gather additional information from users to evaluate alternatives. This process of refinement and evaluation may continue many times for large databases. To depict the iterative nature of data modeling, this section describes some refinements to the initial ERD design of Figure 6.2.

6.2.1 Expanding Attributes

Attribute expansion, a common transformation, adds detail for an attribute. When an entity type should contain more than just the identifier of an entity, this transformation is useful. This transformation involves the replacement of an attribute with an entity type and a 1-M relationship. In the water utility ERD, the *Reading* entity type contains the *EmpNo* attribute. If other data about an employee are needed, *EmpNo* can be expanded into an entity type and 1-M relationship as shown in Figure 6.3a. Figure 6.3b shows transformation of *ProvNo* into an entity type and 1-M relationship. The minimum cardinalities require additional requirements collection as the transformation permits either 0 or 1 as minimum cardinalities.

6.2.2 Splitting Compound Attributes

Compound attribute split, another common refinement, decomposes compound attributes into smaller attributes. A compound attribute embeds multiple attributes. For example, the *CustAddr* attribute in Figure 6.4a embeds components for street, city, state, and postal code. In Figure 6.4b, the *EmpPhone* attribute embeds components for area code, prefix, and line number. Splitting compound attributes can facilitate search of the embedded data. Splitting the address attribute supports searches by street, city, state, and postal code, while splitting the phone attribute supports convenient search by country code, area code, prefix, and line number.

6.2.3 Expanding Entity Types

Entity type expansion provides a finer level of detail about an entity, typically adding another level of detail. This transformation expands an entity type into two entity

FIGURE 6.3
Attribute Expansion Examples

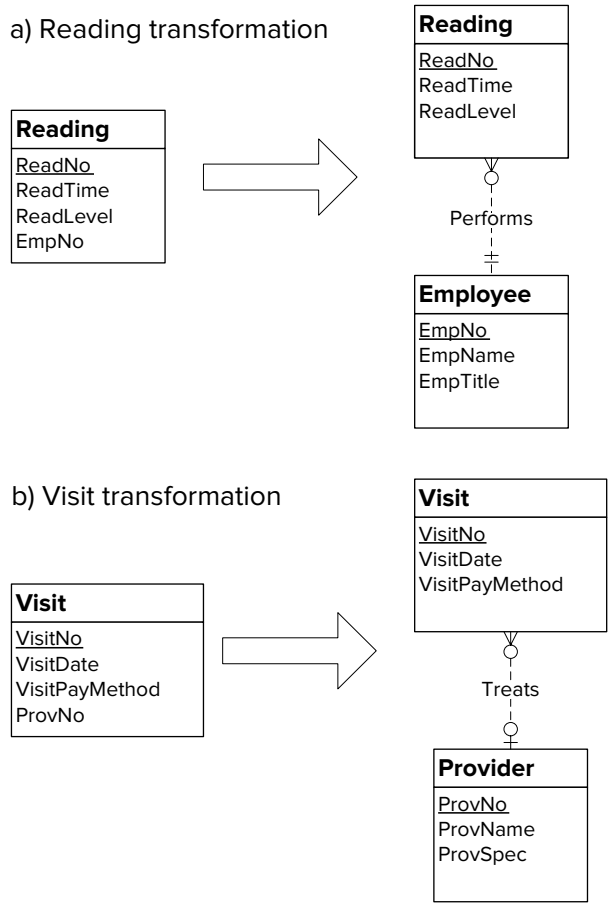
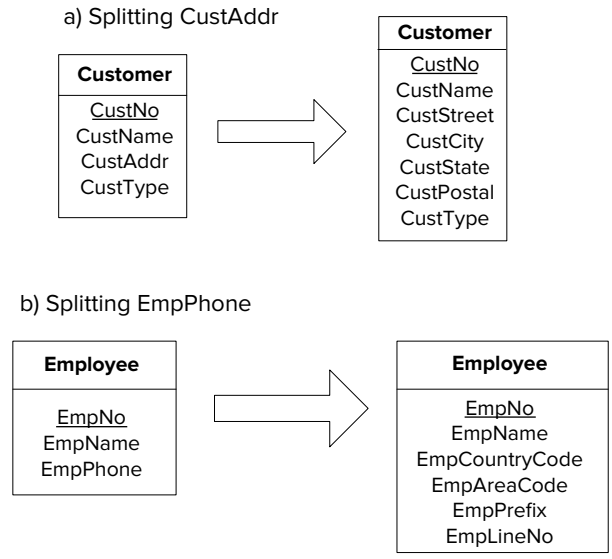


FIGURE 6.4
Examples of Compound Attribute Splits



types and a relationship. Figure 6.5a shows a transformation to the *Rate* entity type to support a more detailed rate structure. The *RateSet* entity type represents a set of rates approved by the utility's governing commission. The primary key of the *Rate* entity type borrows from the *RateSet* entity type. Identification dependency is not required when transforming an entity type into two entity types and a relationship.

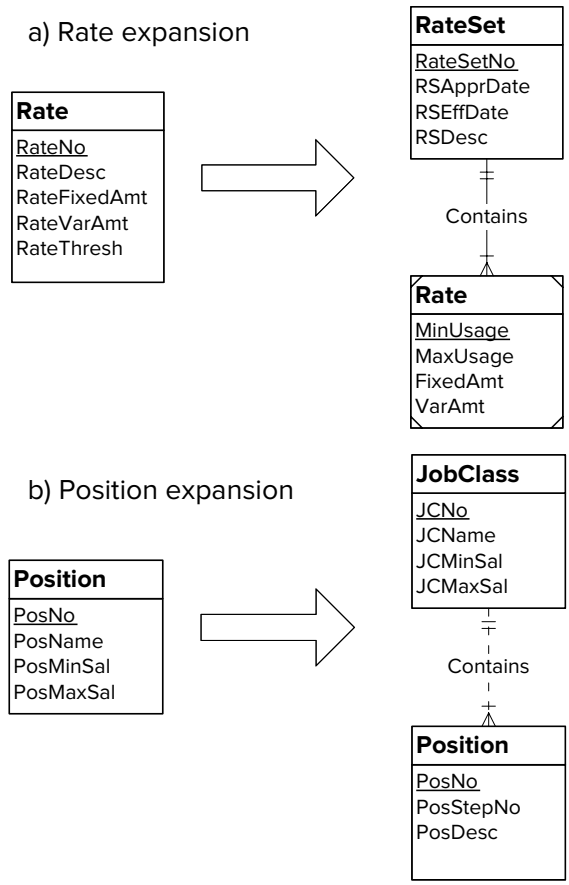


FIGURE 6.5
Entity Type Expansions for Rates (a) and Positions (b)

In this situation, identification dependency is useful, but in other situations, it may not be appropriate.

A similar transformation can be applied to employment positions. A single level of positions can be expanded to a two level structure with job classes and positions as shown in Figure 6.5b. The entity type expansion in Figure 6.5b does not involve identification dependency in contrast to the identification dependency in Figure 6.5a.

6.2.4 Transforming a Weak Entity Type into a Strong Entity Type

The weak-strong entity type transformation makes it easier to reference an entity type after conversion to a table design. After conversion, a reference to a weak entity type involves a combined foreign key with more than one column. This transformation changes a weak entity type into a regular entity type and converts associated identifying relationships into regular (non-identifying) relationships. This transformation is most useful for associative entity types, especially associative entity types representing M-way relationships.

Figure 6.6 depicts two weak-strong entity type transformations. In Figure 6a, *Rate* is transformed into a strong entity type. The transformation involves changing the weak entity type to a strong entity type and changing each identifying relationship to a non-identifying relationship. In addition, it may be necessary to add a new attribute to serve as the primary key. In Figure 6.6a, the new attribute *RateNo* is the primary key as *MinUsage* does not uniquely identify rates. The designer should note that the combination of *RateSetNo* and *MinUsage* is unique in design documentation so that a candidate key constraint can be specified after conversion to a table design. In Figure 6.6b, *RoomId* is added as the primary key of the strong entity type *Room*. The designer should note that the combination of *BldgId* and *RoomNo* is also unique for *Room*.

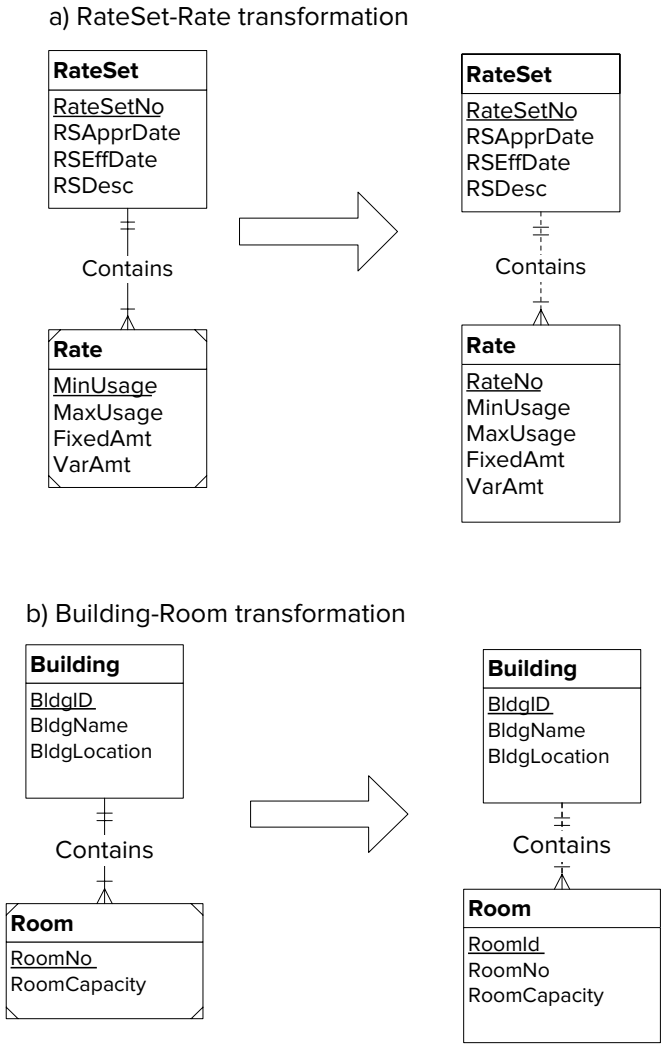


FIGURE 6.6
Examples of Weak-Strong Entity Type Transformations

6.2.5 Adding History

History transformations add historical details to a data model. Historical details may be necessary for legal requirements as well as strategic reporting requirements. History transformations can be applied to attributes and relationships. The attribute history transformation is similar to the attribute expansion. For example, to maintain a history of employee titles, the *EmpTitle* attribute is replaced with an entity type and a 1-M relationship. The new entity type typically contains a version number as part of its primary key and borrows from the original entity type for the remaining part of its primary key, as shown in Figure 6.7a. The beginning and ending dates indicate the effective dates for a change. A similar transformation is shown for student majors in Figure 6.7b.

When applied to a relationship, the history transformation typically involves changing a 1-M relationship into an associative entity type and a pair of identifying 1-M relationships. Figure 6.8a depicts the transformation of the 1-M *Uses* relationship into an associative entity type with attributes for the version number and effective dates. The associative entity type is necessary because the combination of customer and meter may not be unique without a version number. Figure 6.8b depicts a similar transformation for office assignments. The minimum cardinalities of 0 are necessary because offices and employees may exist without assignments.

When applied to an M-N relationship, the history transformation can be more complex. The appropriate transformation depends on the ability of the associated

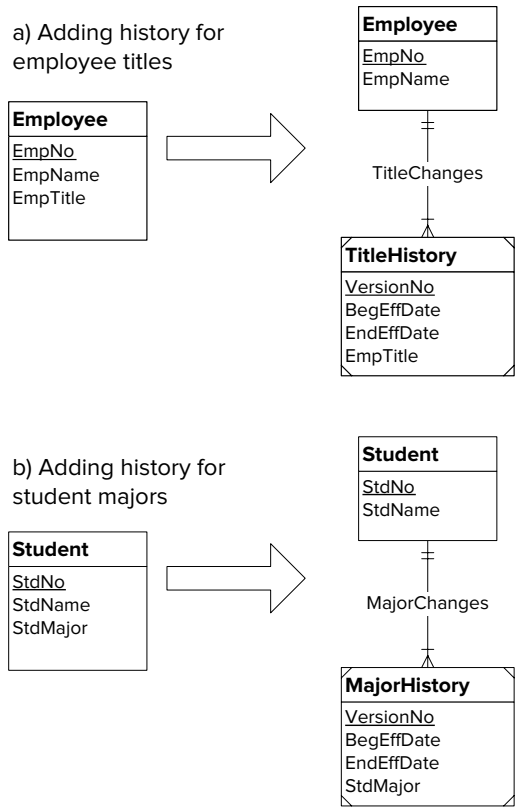


FIGURE 6.7
Examples of Attribute History Transformations

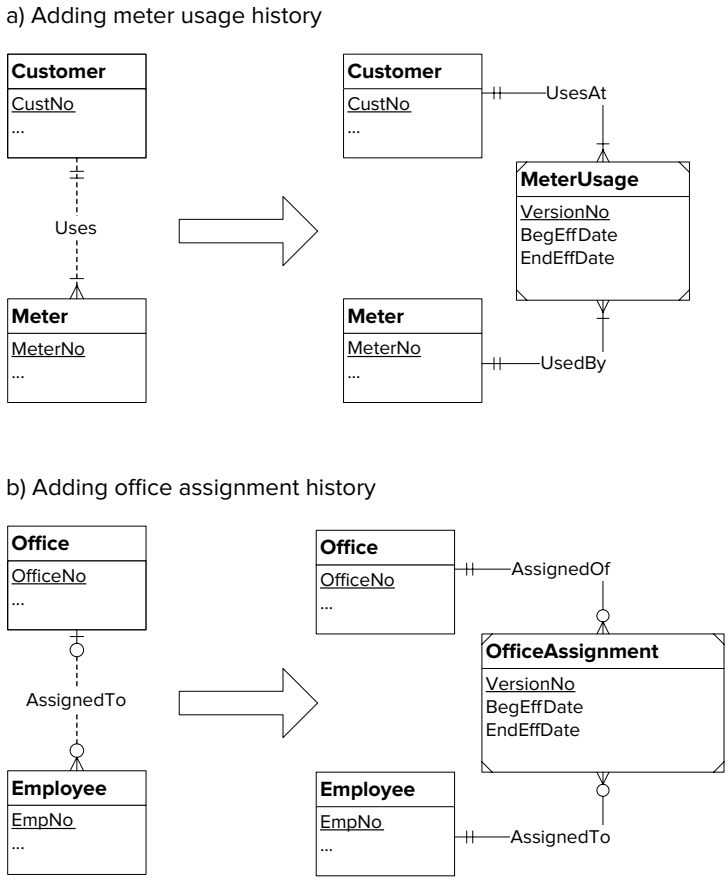


FIGURE 6.8
Examples of History Transformations for 1-M Relationships

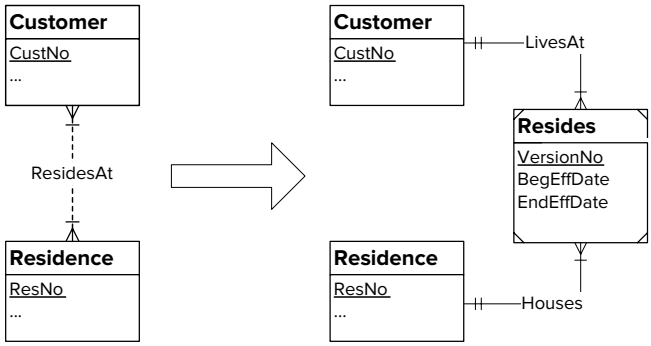


FIGURE 6.9
Adding History to a M-N Relationship with Independent Change

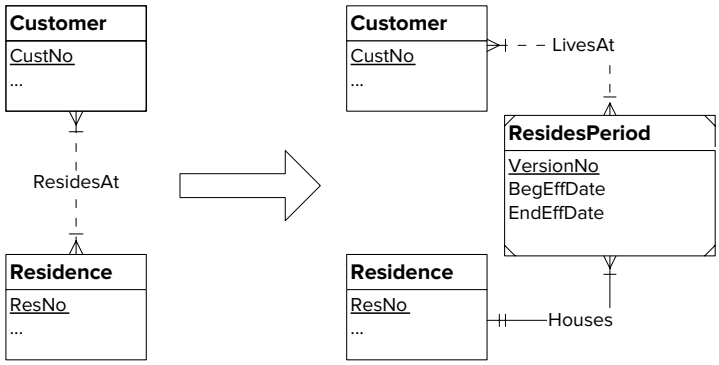


FIGURE 6.10
Adding History to a M-N Relationship with Dependent Change

entities to change independently. In Figure 6.9, the transformation allows the entities to change independently connected by the effective date attributes. For example, if customer C1 resides at residence R1 from 2015 to 2017 and C2 resides at R1 from 2016 to 2017, there will be two *Resides* entities with different version numbers but overlap among the effective date attributes for the two entities. In Figure 6.10, the transformation does not allow the entities to change independently. The *ResidesPeriod* entity type connects all customers that share a residence during the same time period using the *LivesAt* M-N relationship. The transformation in Figure 6.9 is appropriate for property rental in which roommates can change independently. The transformation in Figure 6.10 is appropriate for home ownership in which home owners change as a unit.

The transformations in Figures 6.7 to 6.10 support an unlimited history. For a limited history, a fixed number of attributes can be added to the same entity type. For example to maintain a history of the current and the last employee titles, two attributes (*EmpCurrTitle* and *EmpPrevTitle*) can be used as depicted in Figure 6.11. To record the change dates for employee titles, two effective date attributes per title attribute can be added.

FIGURE 6.11
Adding Limited History to the *Employee* Entity Type

Employee
EmpNo
EmpName
EmpCurrTitle
EmpCurrTitleBegEffDate
EmpCurrTitleEndEffDate
EmpPrevTitle
EmpPrevTitleBegEffDate
EmpPrevTitleEndEffDate

6.2.6 Adding Generalization Hierarchies

A sixth transformation is to make an entity type into a generalization hierarchy. This transformation should be used sparingly because the generalization hierarchy is a specialized modeling tool. If there are multiple attributes that do not apply to all entities and there is an accepted classification of entities, a generalization hierarchy may be useful. For example, water utility customers can be classified as commercial or residential. The attributes specific to commercial customers (*TaxPayerID* and *EnterpriseZone*)

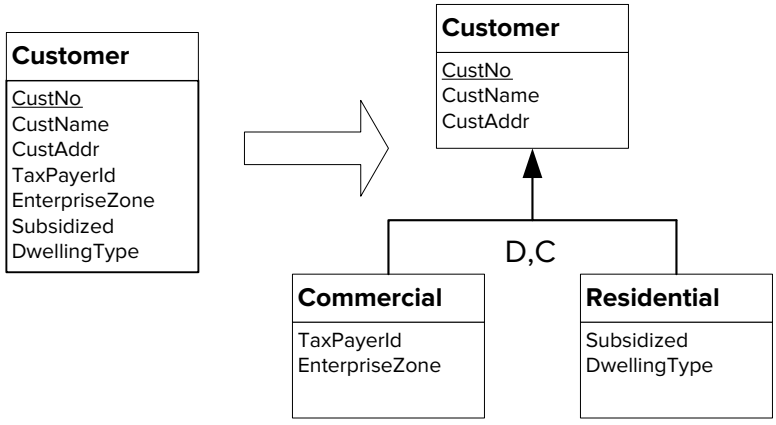


FIGURE 6.12
Generalization Hierarchy Transformation for Water Utility Customers

do not apply to residential customers and vice-versa. In Figure 6.12, the attributes specific to commercial and residential customers have been moved to the subtypes. An additional benefit of this transformation is the avoidance of null values. For example, entities in the *Commercial* and the *Residential* entity types will not have null values. In the original *Customer* entity type, residential customers would have had null values for *TaxPayerID* and *EnterpriseZone*, while commercial customers would have had null values for *Subsidized* and *DwellingType*.

The generalization transformation can also be applied to a collection of entity types. In this situation, the transformation involves the addition of a supertype and a generalization hierarchy as shown in Figure 6.13. In addition, the common attributes in the collection of entity types are moved to the supertype.

6.2.7 Summary of Transformations

When designing a database, you should carefully explore alternative designs. The transformations discussed in this section can help you consider alternative designs. To help you recall the transformations shown in this section, Table 6-2 presents a convenient summary.

The possible transformations are not limited to those discussed in this section. You can reverse the transformations as shown in Table 6-3 although the reversed transformations are less frequently used. The reversed transformations mostly simplify a data model rather than expand it with more details. The reversed transformations are more useful when a data model has been refined several times. Sometimes refinements make a data model too complex so removal of some detail is useful.

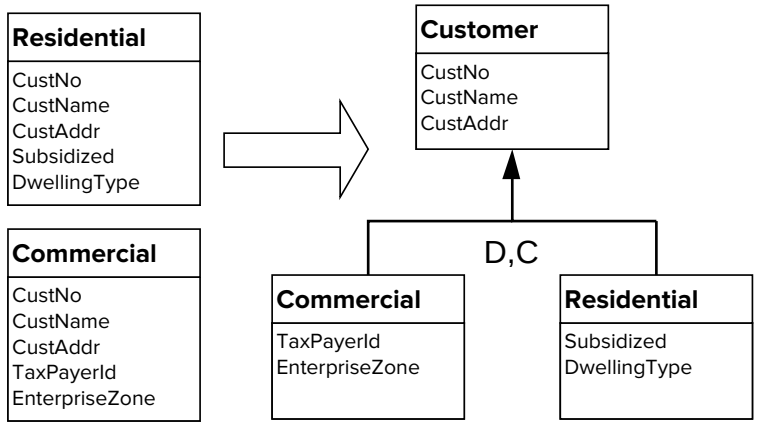


FIGURE 6.13
Generalization Hierarchy Transformation for Similar Entity Types

TABLE 6-2
Summary of Transformations

Transformation	Details	When to Use
Expand attribute	Replace an attribute with an entity type and a 1-M relationship.	Additional details about an attribute are needed.
Split a compound attribute	Replace an attribute with a collection of attributes.	Standardize the components in an attribute.
Expand entity type	Add a new entity type and a 1-M relationship.	Add a finer level of detail about an entity, usually a two-level structure.
Weak entity type to strong entity type	Remove identification dependency symbols and possibly add a primary key.	Remove combined foreign keys after conversion to tables.
Add history	For attribute history, replace an attribute with a weak entity type and an identifying 1-M relationship. For 1-M relationship history, change the relationship into an associative entity type along with identifying relationships. For a limited history, you should add attributes to the entity type.	Add detail for legal requirements or strategic reporting.
Add generalization hierarchy	Starting from a supertype: add subtypes, a generalization hierarchy, and redistribute attributes to subtypes. Starting from subtypes: add a supertype, a generalization hierarchy, and redistribute common attributes and relationships to the supertype.	Accepted classification of entities; specialized attributes and relationships for the subtypes; avoid excessive null values.

TABLE 6-3
Summary of Common Reversed Transformations

Transformation	Details	When to Use
Contract entity type	Replace an entity type and 1-M relationship with an attribute. Opposite of attribute expansion.	Simplify design when no attributes for an entity type are needed.
Group attributes	Combine a collection of attributes into a compound attribute. Opposite of attribute split.	Reduce number of attributes when data collection process cannot parse a compound attribute.
Contract entity type collection	Remove an entity type and a 1-M relationship. Opposite of entity type expansion.	Eliminate details about an entity when more complex structure is not necessary.
Strong entity type to weak entity type	Add identification dependency symbols and possibly a part of primary key. Opposite of weak entity type to strong entity type transformation.	Entity is physically contained in another entity such as building-rooms or order-detail lines.
Remove history	Eliminate all history or reduce from unlimited to fixed history. Opposite of add history expansion.	Legal and practical requirements may allow less history.
Remove generalization hierarchy	Combine a generalization hierarchy into a single entity type. Opposite of add generalization hierarchy expansion.	Not enough specialized attributes to justify a generalization hierarchy.

6.3 FINALIZING AN ERD

After iteratively evaluating alternative ERDs using transformations presented in Section 6.2, you are ready to finalize your data model. Your data model is not complete without adequate design documentation and careful consideration of design errors. You should strive to write documentation and perform design error checking throughout the design process. Even with due diligence during the design process, you will still need to conduct final reviews to ensure adequate design documentation

and lack of design errors. Often these reviews are conducted with a team of designers to ensure completeness. This section presents guidelines to aid you when writing design documentation and checking design errors.

6.3.1 Documenting an ERD

Chapter 5 (Section 5.4.1) prescribed informal documentation for business rules involving uniqueness of attributes, attribute value restrictions, null values, and default values. It is important to document these kinds of business rules because they can be converted to a formal specification in SQL as described in Chapters 11 and 16. You should use informal documentation associated with entity types, attributes, and relationships to record these kinds of business rules.

Resolving Specification Problems Beyond informal representation of business rules, documentation plays an important role in resolving questions about a specification and in communicating a design to others. In the process of revising an ERD, you should carefully document inconsistency and incompleteness in a specification. A large specification typically contains many points of inconsistency and incompleteness. Recording each point allows systematic resolution through additional requirements-gathering activities.

As an example of inconsistency, the water utility requirements would be inconsistent if one part indicated that a meter is associated with one customer, but another part stated that a meter can be associated with multiple customers. In resolving an inconsistency, a user can indicate that an inconsistency is an exception. In this example, a user may indicate the circumstances in which a meter can be associated with multiple customers. The designer must decide on the resolution in the ERD such as permitting multiple customers for a meter, allowing a second responsible customer, or prohibiting more than one customer. The designer should carefully document the resolution of each inconsistency, including a justification for the chosen solution.

As an incompleteness example, the narrative does not specify the minimum cardinality for a meter in the *Uses* relationship of Figure 6.2. The designer should gather additional requirements to resolve the incomplete specification. Incomplete parts of a specification are common for relationships as complete specification involves two sets of cardinalities. It is easy to omit one part of a cardinality specification in an initial requirements effort.

Improving Communication Besides identifying problems in a specification, documentation should be used to communicate a design to others. Databases can have a long lifetime owing to the economics of information systems. An information system can undergo a long cycle of repair and enhancement before there is sufficient justification to redesign the system. Good documentation enhances an ERD by communicating the justification for important design decisions. Your documentation should not repeat the constraints in an ERD. For example, you do not need to document that a customer can use many meters as the ERD contains this information.

You should document decisions in which there is more than one feasible choice. For example, you should carefully document alternative designs for rates (a single consumption level versus multiple consumption levels) as depicted in Figure 6.5. You should document your decision by recording the recommender and justification for the alternative. Although all transformations presented in the previous section can lead to feasible choices, you should focus on transformations most relevant to a specification.

You also should document decisions that might be unclear to others even if there are no feasible alternatives. For example, the minimum cardinality of 0 from the *Reading* entity type to the *Bill* entity type might be unclear. You should document the need for this cardinality because of the time difference between the creation of a bill and its associated readings. A meter may be read days before an associated bill is created.

Design Documentation: include justification for design decisions involving multiple feasible choices and explanations of subtle design choices. Do not use documentation just to repeat the information already contained in an ERD. You also should provide a description for each attribute especially where an attribute's name does not indicate its purpose. As an ERD is developed, you should document incompleteness and inconsistency in the requirements.

Example Design Documentation Design documentation should be incorporated into your ERD. If you are using a drawing tool that has a data dictionary, you should include design justifications in the data dictionary. Most data modeling tools support design justifications as well as comments associated with each item on a diagram. You can use the comments to describe the meaning of attributes. If you are not using a tool that supports documentation, you can list the justifications on a separate page and annotate your ERD as shown in Figure 6.14. The circled numbers in Figure 6.14 refer to explanations in Table 6-4. Note that some of the refinements shown previously were not used in the revised ERD.

6.3.2 Detecting Common Design Errors

As indicated in Chapter 5, you should use the diagram rules to ensure that there are no obvious errors in your ERD. You also should use the guidelines in this section to check for design errors. Design errors are more difficult to detect and resolve than diagram errors because design errors involve the meaning of elements on a diagram, not just

FIGURE 6.14
Revised Water Utility ERD with Annotations

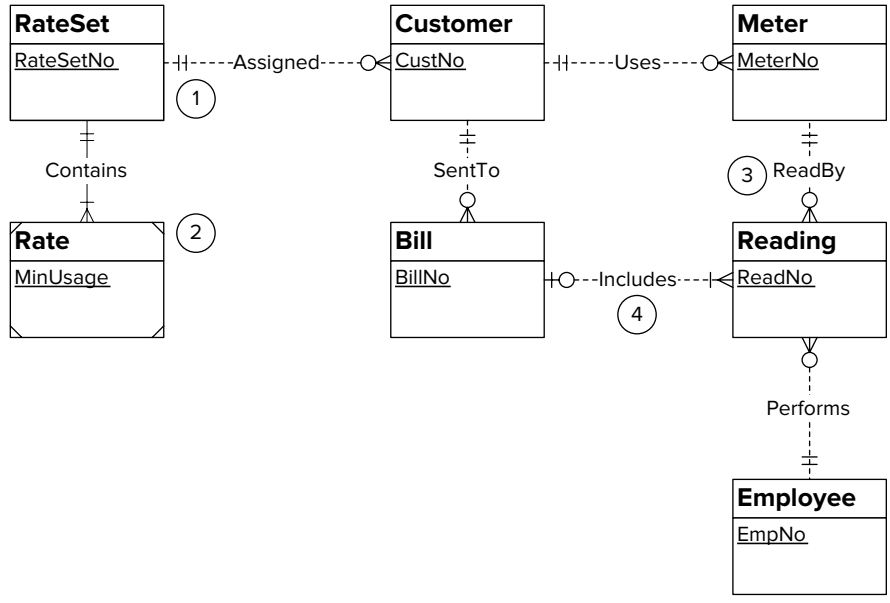


TABLE 6-4
List of Design Justifications for the Revised ERD

1.	A rate set is a collection of rates approved by the governing commission of the utility.
2.	Rates are similar to lines on a tax table. An individual rate is identified by the rate set identifier along with the minimum consumption level of the rate.
3.	The minimum cardinality indicates that a meter must always be associated with a customer. For new property, the developer is initially responsible for the meter. If a customer forecloses on a property, the financial institution holding the deed will be responsible.
4.	A reading is not associated with a bill until the bill is prepared. A reading may be created several days before the associated bill.

TABLE 6-5
Summary of Design Errors

Design Error	Description	Resolution
Misplaced relationship	Wrong entity types connected.	Consider all queries that the database should support.
Missing relationship	Entity types should be connected directly.	Examine implications of requirements.
Incorrect cardinality	1-M relationship instead of an M-N relationship; Reversed cardinality for 1-M relationship.	For 1-M relationship instead of M-N relationship, you should not make inferences beyond incomplete requirements; For reversed 1-M relationships, ensure that specification window is consistent with intended direction.
Overuse of generalization hierarchies	Generalization hierarchies are uncommon. A typical novice mistake is to use them inappropriately.	Ensure that subtypes have specialized attributes and relationships. Generalization indicates the essence of an entity whereas attributes indicate the functions performed by an entity or state of an entity.
Overuse of M-way associative entity types	M-way relationships are uncommon. A typical novice mistake is to use them inappropriately.	Ensure that a database records combinations of three or more entities.
Redundant relationship	Relationship derived from other relationships	Examine each relationship cycle to see if a relationship can be derived from other relationships.

a diagram's structure. The following subsections explain common design problems, while Table 6-5 summarizes them.

Misplaced and Missing Relationships In a large ERD, it is easy to connect the wrong entity types or omit a necessary relationship. You can connect the wrong entity types if you do not consider all of the queries that a database should support. For example in Figure 6.14, if *Customer* is connected directly to *Reading* instead of being connected to *Meter*, the control of a meter cannot be established unless the meter has been read for the customer. Queries that involve meter control cannot be answered except through consideration of meter readings.

If the requirements do not directly indicate a relationship, you should consider indirect implications to detect whether a relationship is required. For example, the requirements for the water utility database do not directly indicate the need for a relationship from *Bill* to *Reading*. However, careful consideration of the consumption calculation reveals the need for a relationship. The *Includes* relationship connects a bill to its most recent meter readings, thus supporting the consumption calculation.

Incorrect Cardinalities The typical error involves the usage of a 1-M relationship instead of an M-N relationship. This error can be caused by an omission in the requirements. For example, if requirements just indicate that work assignments involve a collection of employees, you should not assume that an employee can be related to just one work assignment. You should gather additional requirements to determine if an employee can be associated with multiple work assignments.

Other incorrect cardinality errors that you should consider are reversed cardinalities (1-M relationship should be in the opposite direction) and errors on a minimum cardinality. The error of reversed cardinality is typically an oversight. You may overlook incorrect cardinalities indicated in a relationship specification after the ERD is displayed. You should carefully check all relationships after specification to ensure consistency with your intent. Errors on minimum cardinality are typically the result of overlooking key words in problem narratives such as "optional" and "required."

Overuse of Specialized Data Modeling Constructs Generalization hierarchies and M-way associative entity types are specialized data modeling constructs. A typi-

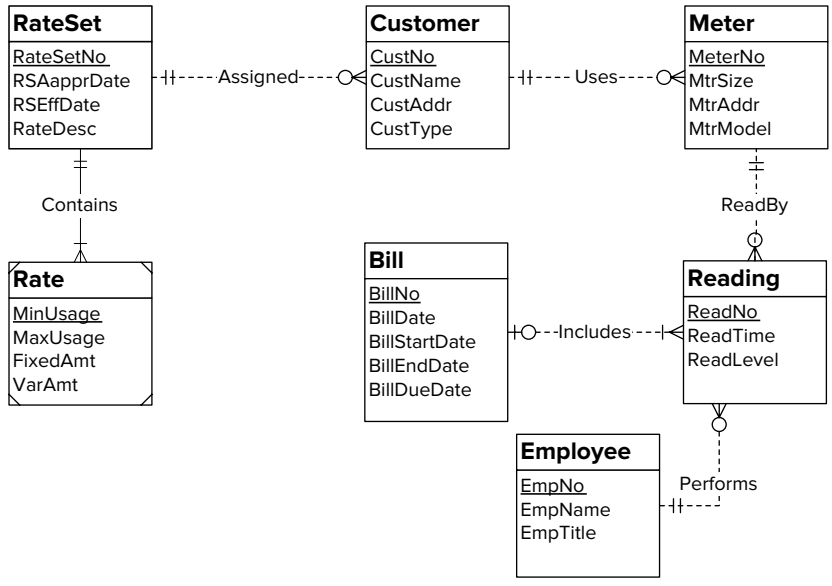
cal novice mistake is to use them inappropriately. You should not use generalization hierarchies just because an entity can exist in multiple states or roles. For example, the requirement that a project task can be started, in progress, or complete, does not indicate the need for a generalization hierarchy. If there is an established classification and specialized attributes and relationships for subtypes, a generalization hierarchy is an appropriate tool.

A rough guideline is that generalization indicates the essence of an entity whereas attributes indicate the functions performed by an entity or state of an entity. Thus, generalization implies relative stability while attributes imply planned changes in state. For example, generalization seems appropriate to represent distinctions between students and faculty, commercial and residential customers, and stocks and bonds. Since these distinctions are relatively permanent, attributes specific to each subtype are useful. In contrast, attributes are appropriate to represent the completion of an order, academic progress of a student, and pay grade of an employee.

An M-way associative entity type (an associative entity type representing an M-way relationship) should be used when the database is to record combinations of three (or more) objects rather than just combinations of two objects. In most cases, only combinations of two objects should be recorded. For example, if a database needs to record the skills provided by an employee and the skills required by a project, binary relationships should be used. If a database needs to record the skills provided by employees for specific projects, an M-way associative entity type is needed. Note that the former situation with binary relationships is much more common than the latter situation represented by an M-way associative entity type.

Redundant Relationships Cycles in an ERD may indicate redundant relationships. A cycle involves a collection of relationships arranged in a loop starting and ending with the same entity type. For example in Figure 6.14, there is a cycle of relationships connecting *Customer*, *Bill*, *Reading*, and *Meter*. In a cycle, a relationship is redundant if it can be derived from other relationships. For the *SentTo* relationship, the bills associated with a customer can be derived from the relationships *Uses*, *ReadBy*, and *Includes*. In the opposite direction, the customer associated with a bill can be derived from

FIGURE 6.15
Final Water Utility ERD



[2] Although the relationship is redundant, deriving it will require 3 joins after table conversion. The possible problem with redundancy may be outweighed by the difficulty of finding the customer for a bill without the relationship. Bills are constructed by starting with a customer, then finding all readings for a meter so all readings on a bill will relate to the same customer by the process of constructing the bill.

the *Includes*, *ReadBy*, and *Uses* relationships. Although a bill can be associated with a collection of readings, each associated reading must be associated with the same customer. Because the *SentTo* relationship can be derived,[2] it is removed in the final ERD (see Figure 6.15).

Another example of a redundant relationship would be a relationship between *Meter* and *Bill*. The meters associated with a bill can be derived using the *Includes* and the *ReadBy* relationships. Note that using clusters of entity types such as *Reading* in the center connected to *Meter*, *Employee*, and *Bill* avoids redundant relationships.

You should take care when removing redundant relationships, as removing a necessary relationship is a more serious error than retaining a redundant relationship. When in doubt, you should retain the relationship.

6.4 CONVERTING AN ERD TO A TABLE DESIGN

Conversion from the ERD notation to table design is important because of industry practice. Computer-aided software engineering (CASE) tools support varying notations for ERDs. It is common practice to use a CASE tool as an aid in developing an ERD. Because most commercial DBMSs use the Relational Model, you must convert an ERD to a table design to implement your database design.

Even if you use a CASE tool to perform conversion, you should still have a basic understanding of the conversion process. Understanding the conversion rules improves your understanding of the ER model, particularly the difference between the Entity Relationship Model and the Relational Model. Some typical errors by novice data modelers are due to confusion between the models. For example, usage of foreign keys in an ERD is due to confusion about relationship representation in the two models.

This section describes the conversion process in two parts. First, the basic rules to convert entity types, relationships, and attributes are described. Second, specialized rules to convert optional 1-M relationships, generalization hierarchies, and 1-1 relationships are shown. The CREATE TABLE statements in this section conform to SQL:2016 syntax.

6.4.1 Basic Conversion Rules

The basic rules convert everything on an ERD except generalization hierarchies. You should apply these rules until everything in your ERD is converted except for generalization hierarchies. You should use the first two rules before the other rules. As you apply these rules, you can use a check mark to indicate the converted parts of an ERD.

1. **Entity Type Rule:** Each entity type (except a subtype) becomes a table. The primary key of the entity type (if not weak) becomes the primary key of the table. The attributes of the entity type become columns in the table. This rule should be used first before the relationship rules.
2. **1-M Relationship Rule:** Each 1-M relationship becomes a foreign key in the table corresponding to the child entity type (the entity type near the Crow's Foot symbol). If the minimum cardinality on the parent side of the relationship is one, the foreign key cannot accept null values.
3. **M-N Relationship Rule:** Each M-N relationship becomes a separate table. The primary key of the table is a combined key consisting of the primary keys of the entity types participating in the M-N relationship.
4. **Identifying Relationship Rule:** Each identifying relationship (denoted by a solid relationship line) adds a component to a primary key. The primary key of the table corresponding to the weak entity type consists of (i) the underlined local key (if any) in the weak entity type and (ii) the primary key(s) of the entity type(s) connected by identifying relationship(s).

FIGURE 6.16
ERD with 1-M Relationship

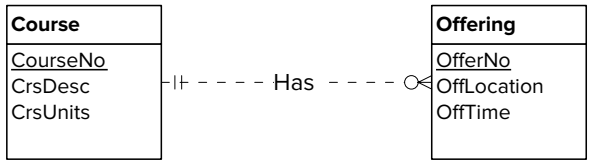


FIGURE 6.17
Conversion of Figure 6.15 (SQL:2016 Syntax)

```
CREATE TABLE Course
( CourseNo CHAR(6),
  CrsDesc   VARCHAR(30),
  CrsUnits  SMALLINT,
CONSTRAINT PKCourse PRIMARY KEY (CourseNo) )

CREATE TABLE Offering
( OfferNo        INTEGER,
 OffLocation           CHAR(20),
 CourseNo         CHAR(6)    NOT NULL,
 OffTime          TIMESTAMP,
 …
CONSTRAINT PKOffering PRIMARY KEY (OfferNo),
CONSTRAINT FKCourseNo FOREIGN KEY (CourseNo) REFERENCES Course )
```

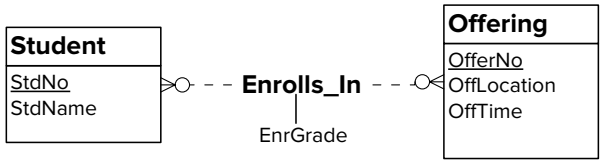
To understand these rules, you can apply them to some of the ERDs used in Chapter 5. Using Rules 1 and 2, you can convert Figure 6.16 into the CREATE TABLE statements shown in Figure 6.17. Rule 1 is applied to convert the *Course* and *Offering* entity types to tables. Then, Rule 2 is applied to convert the *Has* relationship to a foreign key (*Offering.CourseNo*). The *Offering* table contains the foreign key because the *Offering* entity type is the child entity type in the *Has* relationship.

Next, you can apply the M-N relationship rule (Rule 3) to convert the ERD in Figure 6.18. Following this rule leads to the *Enrolls_In* table in Figure 6.19. The primary key of *Enrolls_In* is a combination of the primary keys of the *Student* and *Offering* entity types.

To gain practice with the identification dependency rule (Rule 4), you can use it to convert the ERD in Figure 6.20. The result of converting Figure 6.20 is identical to Figure 6.19 except that the *Enrolls_In* table is renamed *Enrollment*. The ERD in Figure 6.20 requires two applications of the identification dependency rule. Each application of the identification dependency rule adds a component to the primary key of the *Enrollment* table.

You can also apply the rules to convert self-referencing relationships. For example, you can apply the 1-M and M-N relationship rules to convert the self-referencing relationships in Figure 6.21. Using the 1-M relationship rule, the *Supervises* relationship converts to a foreign key in the *Faculty* table, as shown in Figure 6.22. Using the M-N relationship rule, the *Prereq_To* relationship converts to the *Prereq_To* table with a combined primary key of the course number of the prerequisite course and the course number of the dependent course.

FIGURE 6.18
M-N Relationship with an Attribute



```
CREATE TABLE Student
(     StdNo             CHAR(11),
      StdName           VARCHAR(30),
      …
CONSTRAINT PKStudent PRIMARY KEY (StdNo)  )

CREATE TABLE Offering
(     OfferNo           INTEGER,
      OffLocation       VARCHAR(30),
      OffTime           TIMESTAMP,
      …
CONSTRAINT PKOffering PRIMARY KEY (OfferNo) )

CREATE TABLE Enrolls_In
(     OfferNo           INTEGER,
      StdNo             CHAR(11),
      EnrGrade          DECIMAL(2,1),
CONSTRAINT PKEnrolls_In PRIMARY KEY (OfferNo, StdNo),
CONSTRAINT FKOfferNo FOREIGN KEY (OfferNo) REFERENCES Offering,
CONSTRAINT FKStdNo FOREIGN KEY (StdNo) REFERENCES Student )
```

FIGURE 6.19
Conversion of Figure 6.18 (SQL:2016 Syntax)

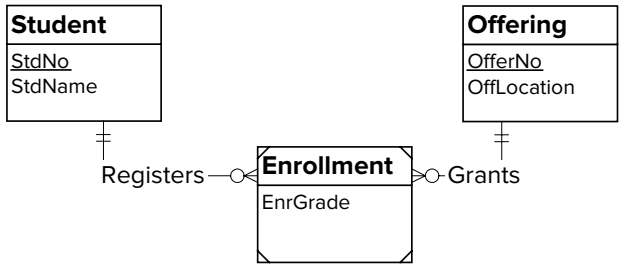


FIGURE 6.20
Enrolls_in M-N Relationship Transformed into 1-M Relationships

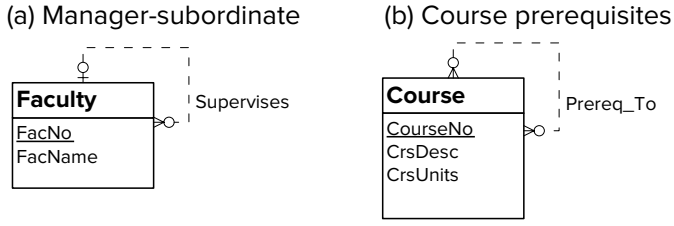


FIGURE 6.21
Examples of 1-M and M-N Self-Referencing Relationships

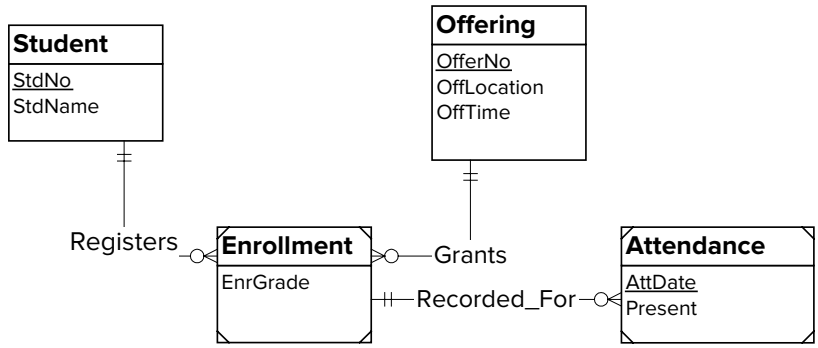
You also can apply conversion rules to more complex identification dependencies as depicted in Figure 6.23. The first part of the conversion is identical to the conversion of Figure 6.20. Application of the 1-M rule makes the combination of *StdNo* and *OfferNo* foreign keys in the *Attendance* table (Figure 6.24). Note that the foreign keys in *Attendance* refer to *Enrollment*, not to *Student* and *Offering*. Finally, one application of the identification dependency rule makes the combination of *StdNo*, *OfferNo*, and *AttDate* the primary key of the *Attendance* table.

The conversion in Figure 6.24 depicts a situation in which the transformation of a weak to a strong entity may apply (Section 6.2.3). In the conversion, the *Attendance* table contains a combined foreign key (*OfferNo*, *StdNo*). Changing *Enrollment* into a strong entity will eliminate the combined foreign key in the *Attendance* table.

FIGURE 6.22
Conversion of Figure 6.20 (SQL:2016 Syntax)

```
CREATE TABLE Faculty
(     FacNo              CHAR(11),
      FacName            VARCHAR(30),
      FacSupervisor      CHAR(11),
      …
CONSTRAINT PKFaculty PRIMARY KEY (FacNo),
CONSTRAINT FKSupervisor FOREIGN KEY (FacSupervisor) REFERENCES
Faculty )

CREATE TABLE Course
(     Courseno           CHAR(6),
      CrsDesc            VARCHAR(30),
      CrsUnits           SMALLINT,
CONSTRAINT PKCourse PRIMARY KEY (CourseNo) )

CREATE TABLE Prereq_To
(     PrereqCNo          CHAR(6),
      DependCNo          CHAR(6),
CONSTRAINT PKPrereq_To PRIMARY KEY (PrereqCNo, DependCNo),
CONSTRAINT FKPrereqCNo FOREIGN KEY (PrereqCNo) REFERENCES Course,
CONSTRAINT FKDependCNo FOREIGN KEY (DependCNo) REFERENCES Course )
```

FIGURE 6.23
ERD with Two Weak Entity Types



Summary of Basic Conversion Rules Figure 6.25 shows a convenient summary of the basic conversion rules. You should apply the rules shown in Figure 6.25 starting with the entity type rule. The 1-M relationship rule embodies the basic difference between the entity relationship model with named relationships and relational model with foreign keys. The M-N relationship rule generates an associative table, foreign keys, and combined primary key for each relationships. The identifying relationship rule generates a component of a primary key for each identifying relationship.

6.4.2 Converting Optional 1-M Relationships

When you use the 1-M relationship rule for optional relationships, the resulting foreign key contains null values. Recall that a relationship with a minimum cardinality of 0 is optional. For example, the *Teaches* relationship (Figure 6.26) is optional to *Offering* because an *Offering* entity can be stored without being related to a *Faculty* entity. Converting Figure 6.26 results in two tables (*Faculty* and *Offering*) as well as a foreign key (*FacNo*) in the *Offering* table. The foreign key should allow null values

```
CREATE TABLE Attendance
(     OfferNo           INTEGER,
      StdNo             CHAR(11),
      AttDate           DATE,
      Present           BOOLEAN,
CONSTRAINT PKAttendance PRIMARY KEY (OfferNo, StdNo, AttDate),
CONSTRAINT FKOfferNoStdNo FOREIGN KEY (OfferNo, StdNo)
 REFERENCES Enrollment )
```

FIGURE 6.24
Conversion of the Attendance Entity Type in Figure 6.23 (SQL:2016 Syntax)

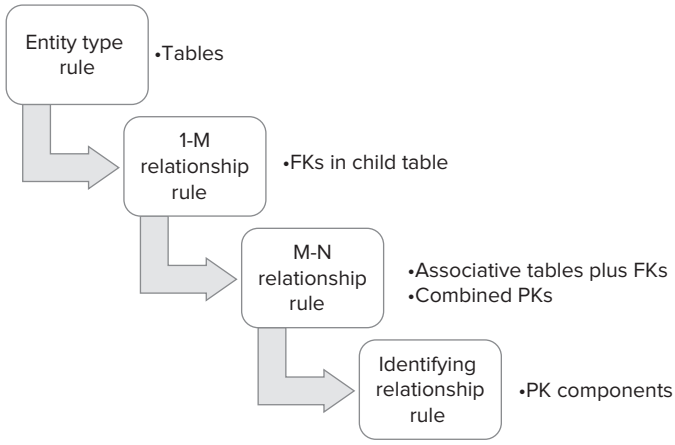


FIGURE 6.25
Application of Basic Conversion Rules

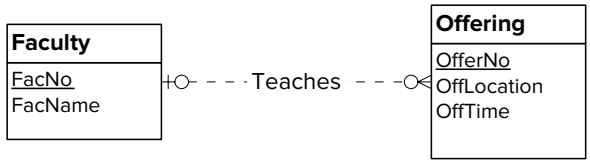


FIGURE 6.26
Optional 1-M Relationship

because the minimum cardinality of the *Offering* entity type in the relationship is optional (0). However, null values can lead to complications in evaluating the query results.

To avoid null values when converting an optional 1-M relationship, you can apply Rule 5 to convert an optional 1-M relationship into a table instead of a foreign key. Figure 6.27 shows an application of Rule 5 to the ERD in Figure 6.26. The *Teaches* table contains the foreign keys *OfferNo* and *FacNo* with null values not allowed for both columns. In addition, the *Offering* table no longer has a foreign key referring to the *Faculty* table. Figures 6.28 and 6.29 depict an example of converting an optional 1-M relationship with an attribute. Note that the *Lists* table contains the *Commission* column.

5. **Optional 1-M Relationship Rule**: Each 1-M relationship with 0 for the minimum cardinality on the parent side becomes a new table. The primary key of the new table is the primary key of the entity type on the child (many) side of the relationship. The new table contains foreign keys for the primary keys of both entity types participating in the relationship. Both foreign keys in the new table do not permit null values. The new table also contains the attributes of the optional 1-M relationship.

Rule 5 is controversial. Using Rule 5 in place of Rule 2 (1-M Relationship Rule) avoids null values in foreign keys. However, the use of Rule 5 results in more tables. Query formulation can be more difficult with additional tables. In addition, query execution can be slower due to extra joins. The choice of using Rule 5 in place of Rule

FIGURE 6.27
Conversion of Figure 6.25 (SQL:2016 Syntax)

```
CREATE TABLE Faculty
(     FacNo             CHAR(11),
      FacName           VARCHAR(30),
      …
CONSTRAINT PKFaculty PRIMARY KEY (FacNo)  )

CREATE TABLE Offering
(     OfferNo           INTEGER,
      OffLocation       VARCHAR(30),
      OffTime           TIMESTAMP,
      …
CONSTRAINT PKOffering PRIMARY KEY (OfferNo) )

CREATE TABLE Teaches
(     OfferNo           INTEGER,
      FacNo             CHAR(11)    NOT NULL,
CONSTRAINT PKTeaches PRIMARY KEY (OfferNo),
CONSTRAINT FKFacNo FOREIGN KEY (FacNo) REFERENCES Faculty,
CONSTRAINT FKOfferNo FOREIGN KEY (OfferNo) REFERENCES Offering )
```

FIGURE 6.28
Optional 1-M Relationship with an Attribute

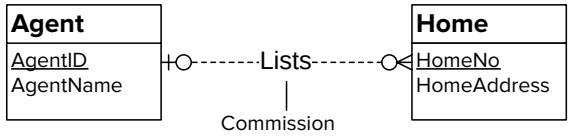


FIGURE 6.29
Conversion of Figure 6.28 (SQL:2016 Syntax)

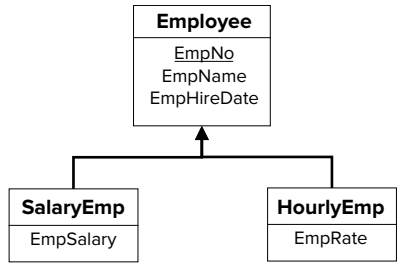
```
CREATE TABLE Agent
(     AgentId           CHAR(10),
      AgentName         VARCHAR(30),
      …
CONSTRAINT PKAgent PRIMARY KEY (AgentId)  )

CREATE TABLE Home
(     HomeNo            INTEGER,
      HomeAddress       VARCHAR(50),
      …
CONSTRAINT PKHome PRIMARY KEY (HomeNo) )

CREATE TABLE Lists
(     HomeNo            INTEGER,
      AgentId           CHAR(10)    NOT NULL,
      Commission        DECIMAL(10,2),
CONSTRAINT PKLists PRIMARY KEY (HomeNo),
CONSTRAINT FKAgentId FOREIGN KEY (AgentId) REFERENCES Agent,
CONSTRAINT FKHomeNo FOREIGN KEY (HomeNo) REFERENCES Home )
```

2 depends on the importance of avoiding null values versus avoiding extra tables. In many situations, avoiding extra tables may be more important than avoiding null values.

6.4.3 Converting Generalization Hierarchies

The approach to convert generalization hierarchies mimics the entity relationship notation as much as possible. Rule 6 converts each entity type of a generalization hierarchy into a table. The only columns that are different from attributes in the associated ERD are the inherited primary key attributes. In Figure 6.30, *EmpNo* is a column in the *SalaryEmp* and *HourlyEmp* tables because it is the primary key of the parent entity type (*Employee*). In addition, the *SalaryEmp* and *HourlyEmp* tables have a foreign key constraint referring to the *Employee* table. The CASCADE delete option is set in both foreign key constraints (see Figure 6.31).

FIGURE 6.30
Generalization Hierarchy for Employees



6. **Generalization Hierarchy Rule**: Each entity type of a generalization hierarchy becomes a table. The columns of a table are the attributes of the corresponding entity type plus the primary key of the parent entity type. For each table representing a subtype, define a foreign key constraint that references the table corresponding to the parent entity type. Use the CASCADE option for deletions of referenced rows.

Rule 6 also applies to generalization hierarchies of more than one level. To convert the generalization hierarchy of Figure 6.32, five tables are produced (see Figure 6.33).

```
CREATE TABLE Employee
(     EmpNo             INTEGER,
      EmpName           VARCHAR(30),
      EmpHireDate       DATE,
CONSTRAINT PKEmployee PRIMARY KEY (EmpNo) )

CREATE TABLE SalaryEmp
(     EmpNo             INTEGER,
      EmpSalary         DECIMAL(10,2),
CONSTRAINT PKSalaryEmp PRIMARY KEY (EmpNo),
CONSTRAINT FKSalaryEmp FOREIGN KEY (EmpNo) REFERENCES Employee
  ON DELETE CASCADE )

CREATE TABLE HourlyEmp
(     EmpNo             INTEGER,
      EmpRate           DECIMAL(10,2),
CONSTRAINT PKHourlyEmp PRIMARY KEY (EmpNo),
CONSTRAINT FKHourlyEmp FOREIGN KEY (EmpNo) REFERENCES Employee
  ON DELETE CASCADE   )
```

FIGURE 6.31
Conversion of the Generalization Hierarchy in Figure 6.30 (SQL:2016 Syntax)

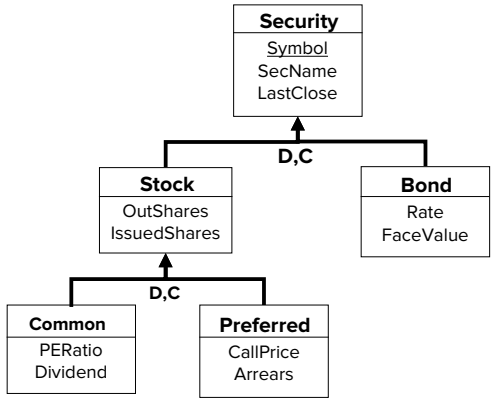


FIGURE 6.32
Multiple Levels of Generalization Hierarchies

FIGURE 6.33
Conversion of the Generalization Hierarchy in Figure 6.32 (SQL:2016 Syntax)

```
CREATE TABLE Security
(     Symbol            CHAR(6),
      SecName           VARCHAR(30),
      LastClose         DECIMAL(10,2),
CONSTRAINT PKSecurity PRIMARY KEY (Symbol) )

CREATE TABLE Stock
(     Symbol            CHAR(6),
      OutShares         INTEGER,
      IssuedShares      INTEGER,
CONSTRAINT PKStock PRIMARY KEY (Symbol),
CONSTRAINT FKStock FOREIGN KEY (Symbol) REFERENCES Security
  ON DELETE CASCADE )

CREATE TABLE Bond
(     Symbol            CHAR(6),
      Rate              DECIMAL(12,4),
      FaceValue         DECIMAL(10,2),
CONSTRAINT PKBond PRIMARY KEY (Symbol),
CONSTRAINT FKBond FOREIGN KEY (Symbol) REFERENCES Security
  ON DELETE CASCADE )
CREATE TABLE Common
(     Symbol            CHAR(6),
      PERatio           DECIMAL(12,4),
      Dividend          DECIMAL(10,2),
CONSTRAINT PKCommon PRIMARY KEY (Symbol),
CONSTRAINT FKCommon FOREIGN KEY (Symbol) REFERENCES Stock
  ON DELETE CASCADE )

CREATE TABLE Preferred
(     Symbol            CHAR(6),
      CallPrice         DECIMAL(12,2),
      Arrears           DECIMAL(10,2),
CONSTRAINT PKPreferred PRIMARY KEY (Symbol),
CONSTRAINT FKPreferred FOREIGN KEY (Symbol) REFERENCES Stock
  ON DELETE CASCADE )
```

In each table, the primary key of the parent (*Security*) is included. In addition, foreign key constraints are added in each table corresponding to a subtype.

Because the Relational Model does not directly support generalization hierarchies, there are several other ways to convert generalization hierarchies. The other approaches vary depending on the number of tables and the placement of inherited columns. Rule 6 may result in extra joins to gather all data about an entity, but there are no null values and only small amounts of duplicate data. For example, to collect all data about a common stock, you should join the *Common*, *Stock*, and *Security* tables. Other conversion approaches may require fewer joins, but result in more redundant data and null values. The references at the end of this chapter discuss the pros and cons of several approaches to convert generalization hierarchies.

You should note that generalization hierarchies for tables are directly supported in SQL:2016, the standard for object relational databases presented in Chapter 19. In the SQL:2016 standard, subtable families provide a direct conversion from generalization hierarchies avoiding the loss of semantic information when converting to the traditional Relational Model. However, few commercial DBMS products fully support

the object relational features in SQL:2016. Thus, usage of the generalization hierarchy conversion rule will likely be necessary.

To support usage of the Generalization Hierarchy Rule, Chapter 11 presents triggers to support operations on tables in a generalization hierarchy. The triggers support propagation among tables when inserting and updating rows in a generalization hierarchy as well as enforcement of generalization hierarchy constraints.

6.4.4 Converting 1-1 Relationships

Outside of generalization hierarchies, 1-1 relationships are not common. They can occur when entities with separate identifiers are closely related. For example, Figure 6.34 shows the *Employee* and *Office* entity types connected by a 1-1 relationship. Separate entity types seem intuitive, but a 1-1 relationship connects the entity types. Rule 7 converts 1-1 relationships into two foreign keys unless many null values will result. In Figure 6.34, most employees will not manage offices. Thus, the conversion in Figure 6.35 eliminates the foreign key (*OfficeNo*) in the employee table.

7. **1-1 Relationship Rule**: Each 1-1 relationship is converted into two foreign keys. If the relationship is optional with respect to one of the entity types, the corresponding foreign key may be dropped to eliminate null values.

6.4.5 Comprehensive Conversion Example

This section presents a larger example to integrate your knowledge of the conversion rules. Figure 6.36 shows an ERD similar to the final ERD for the water utility problem discussed in Section 6.1. For brevity, some attributes have been omitted. Figure 6.37 shows the relational tables derived through the conversion rules. Table 6-6 lists the conversion rules used along with brief explanations.

6.4.6 Conversion Practices in Commercial CASE Tools

Commercial CASE tools convert ERDs so that a table design closely matches its associated ERD. This philosophy dictates the dominant usage of the basic rules in the

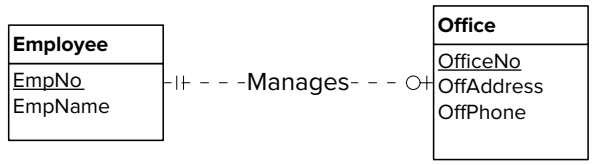


FIGURE 6.34
1-1 Relationship

```
CREATE TABLE Employee
(     EmpNo             INTEGER,
      EmpName           VARCHAR(30),
CONSTRAINT PKEmployee PRIMARY KEY (EmpNo) )

CREATE TABLE Office
(     OfficeNo          INTEGER,
      OffAddress        VARCHAR(30),
      OffPhone          CHAR(10),
      EmpNo             INTEGER,
CONSTRAINT PKOffice PRIMARY KEY (OfficeNo),
CONSTRAINT FKEmpNo FOREIGN KEY (EmpNo) REFERENCES Employee,
CONSTRAINT EmpNoUnique UNIQUE (EmpNo) )
```

FIGURE 6.35
Conversion of the 1-1 Relationship in Figure 6.34 (SQL:2016 Syntax)

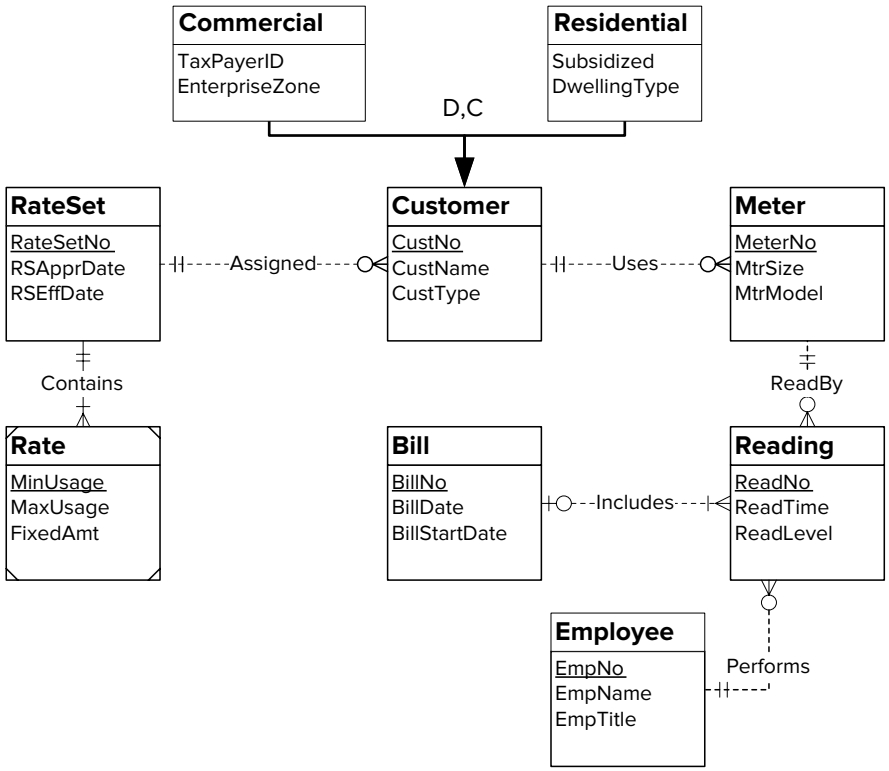


FIGURE 6.36
Water Utility ERD with a Generalization Hierarchy

FIGURE 6.37
Conversion of the ERD in Figure 6.36 (SQL:2016 Syntax)

```
CREATE TABLE Customer
(     CustNo            INTEGER,
      CustName          VARCHAR(30),
      CustType          CHAR(6),
      RateSetNo         INTEGER     NOT NULL,
CONSTRAINT PKCustomer PRIMARY KEY (CustNo),
CONSTRAINT FKRateSetNo FOREIGN KEY (RateSetNo)
  REFERENCES RateSet  )

CREATE TABLE Commercial
(     CustNo            INTEGER,
      TaxPayerID        CHAR(20)    NOT NULL,
      EnterpiseZone     BOOLEAN,
CONSTRAINT PKCommercial PRIMARY KEY (CustNo),
CONSTRAINT FKCommercial FOREIGN KEY (CustNo)
  REFERENCES Customer ON DELETE CASCADE  )

CREATE TABLE Residential
(     CustNo            INTEGER,
      Subsidized        BOOLEAN,
      DwellingType      CHAR(6),
CONSTRAINT PKResidential PRIMARY KEY (CustNo),
CONSTRAINT FKResidential FOREIGN KEY (CustNo)
   REFERENCES Customer ON DELETE CASCADE   )

CREATE TABLE RateSet
(     RateSetNo         INTEGER,
```

```
      RSApprDate        DATE,
      RSEffDate         DATE,
CONSTRAINT PKRateSet PRIMARY KEY (RateSetNo)  )

CREATE TABLE Rate
(     RateSetNo         INTEGER,
      MinUsage          INTEGER,
      MaxUsage          INTEGER,
      FixedAmt          DECIMAL(10,2),
CONSTRAINT PKRate PRIMARY KEY (RateSetNo, MinUsage),
CONSTRAINT FKRateSetNo2 FOREIGN KEY(RateSetNo)
   REFERENCES RateSet  )

CREATE TABLE Meter
(     MeterNo           INTEGER,
      MtrSize           INTEGER,
      MtrModel          CHAR(6),
      CustNo            INTEGER           NOT NULL,
CONSTRAINT PKMeter PRIMARY KEY (MeterNo),
CONSTRAINT FKCustNo FOREIGN KEY (CustNo)
  REFERENCES Customer  )

CREATE TABLE Reading
(     ReadNo            INTEGER,
      ReadTime          TIMESTAMP,
      ReadLevel         INTEGER,
      MeterNo           INTEGER           NOT NULL,
      EmpNo             INTEGER           NOT NULL,
      BillNo            INTEGER,
CONSTRAINT PKReading PRIMARY KEY (ReadNo),
CONSTRAINT FKEmpNo FOREIGN KEY (EmpNo) REFERENCES Employee,
CONSTRAINT FKMeterNo FOREIGN KEY (MeterNo) REFERENCES Meter,
CONSTRAINT FKBillNo FOREIGN KEY (BillNo) RERERENCES Bill  )

CREATE TABLE Bill
(     BillNo            INTEGER,
      BillDate          DATE,
      BillStartDate     DATE,
CONSTRAINT PKBill PRIMARY KEY (BillNo) )

CREATE TABLE Employee
(     EmpNo             INTEGER,
      EmpName           VARCHAR(50),
      EmpTitle          VARCHAR(20),
CONSTRAINT PKEmployee PRIMARY KEY (EmpNo) )
```

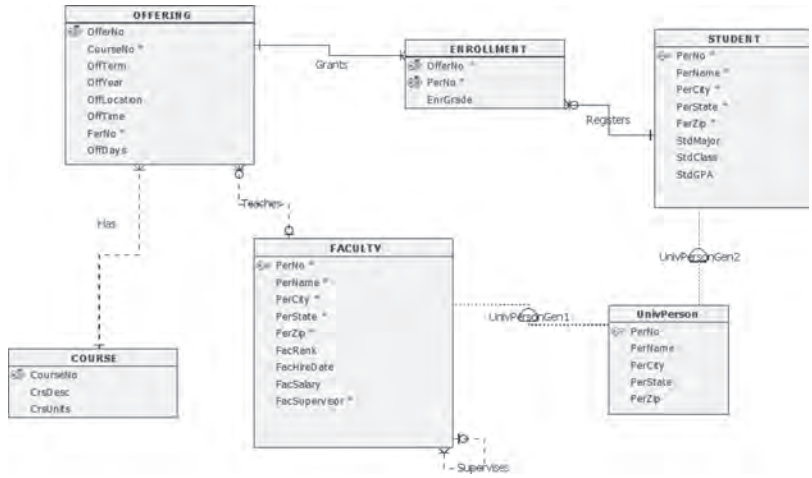
FIGURE 6.37
(Continued)

conversion process. The optional 1-M relationship rule is not used although a table design can be manually changed after the conversion to apply the optional 1-M relationship rule. Since representation of generalization hierarchies varies widely among commercial CASE tools, it is not possible to make a general statement about conversion practices. Typically, the conversion would result in the same number of tables as entity types. However, since the basic relational model does not support generalization hierarchies, the attributes in converted subtype tables may be just the direct attributes or all attributes (direct and inherited).

TABLE 6-6
Conversion Rules Used for Figure 6.36

Rule	Usage
1	All entity types except subtypes converted to tables with primary keys.
2	1-M relationships converted to foreign keys: *Contains* relationship to *Rate.RateSetNo; Uses* relationship to *Meter.CustNo; ReadBy* relationship to *Reading.MeterNo; Includes* relationship to *Reading.BillNo; Performs* relationship to *Reading.EmpNo; Assigned* relationship to *Customer.RateSetNo*
3	Not used because there are no M-N relationships.
4	Primary key of *Rate* table is a combination of *RateSetNo* and *MinUsage*.
5	Not used although it could have been used for the *Includes* relationship.
6	Subtypes (*Commercial* and *Residential*) converted to tables. Primary key of *Customer* is added to the *Commercial* and *Residential* tables. Foreign key constraints with CASCADE DELETE options added to tables corresponding to the subtypes.

FIGURE 6.38
Extended University Database in the Data Modeling Tool of Aqua Data Studio



To depict a commercial conversion tool, Figure 6.38 shows the extended university database in the ERD notation of Aqua Data Studio. The ERD has six entity types with generalization relationships from *UnivPerson* to *Student* and *UnivPerson* to *Faculty*. The *Student* and *Faculty* entity types show direct and inherited attributes (*PerNo*, *PerName*, *PerCity*, *PerState*, and *PerZip*). The conversion feature in Aqua Data Studio (known as Generate Scripts) creates one table per entity type. The *Student* and *Faculty* tables resulting from the conversion have all attributes shown in the ERD, both direct and inherited. Foreign key constraints are created as part of the conversion of generalization relationships. Thus, the *Faculty* and *Student* tables have foreign key constraints referencing the *UnivPerson* table.

As another example, Figure 6.39 shows the extended university database in the Entity Relationship stencil of Visio 2010 Professional Edition. Like the conversion in Aqua Data Studio, each entity type in an ERD corresponds to a table. However, Visio has an implicit conversion feature in contrast to the explicit feature in Aqua Data Studio. In Visio Professional 2010, you set display options to see the conversion in the ERD. In Figure 6.39, Visio displays foreign keys in the ERD because the foreign key display option is set. The table properties window shows additional properties including triggers (see Chapter 11) and check constraints (see Chapter 16). Visio's implicit conversion feature does not support generalization hierarchies in the relational model. In the conversion result, child entity types (*Student* and *Faculty*) only have the direct attributes, not the inherited attributes.

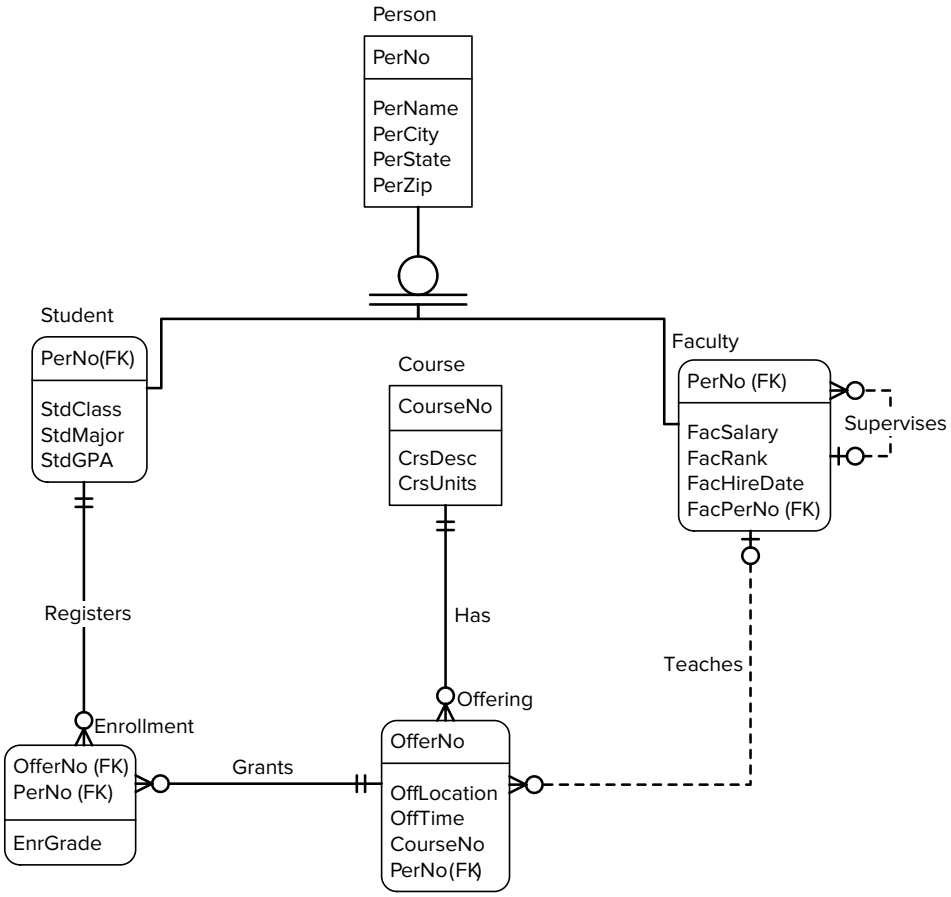


FIGURE 6.39
Extended University Database in Visio 2010 Professional Edition

CLOSING THOUGHTS

This chapter has described the practice of data modeling, building on your understanding of the Crow's Foot notation presented in Chapter 5. To master data modeling, you need to understand the notation used in entity relationship diagrams (ERDs) and get plenty of practice building ERDs. This chapter described techniques to derive an initial ERD from a narrative problem, refine the ERD through transformations, document important design decisions, and check the ERD for design errors. To apply these techniques, a practice problem for a water utility database was presented. You are encouraged to apply these techniques using the problems at the end of the chapter.

The remainder of this chapter presented rules to convert an ERD into relational tables and alternative ERD notations. The rules will help you convert modest-size ERDs into tables. For large problems, you should use a good CASE tool. Even if you use a CASE tool, understanding the conversion rules provides insight into the differences between the Entity Relationship Model and the Relational Model.

This chapter emphasized data modeling skills for constructing ERDs using narrative problems, refining ERDs, and converting ERDs into relational tables. The next chapter presents normalization, a technique to remove redundancy from a table design. Together, data modeling and normalization are fundamental skills for database development.

After you master these database development skills, you are ready to apply them to database design projects. An additional challenge of applying your skills is requirements definition. It is a lot of work to collect requirements from users with diverse interests and backgrounds. You may spend more time gathering requirements than performing data modeling and normalization. With careful study and practice, you will find database development to be a challenging and highly rewarding activity.

REVIEW CONCEPTS

- Identifying entity types and attributes in a narrative
- Criteria for primary keys: stable and single purpose
- Avoidance of government issued identification attributes as primary keys due to privacy concerns
- Identifying relationships in a narrative
- Transformations to add detail to an ERD: expanding an attribute, expanding an entity type, adding history
- Splitting an attribute to standardize information content and improve query results
- Changing a weak entity type to a strong entity type to remove combined foreign keys after conversion
- Adding a generalization hierarchy to avoid null values
- Reversed transformations to simplify a design with excessive detail
- Documentation practices for important design decisions: justification for design decisions involving multiple feasible choices and explanations of subtle design choices.
- Poor documentation practices: repeating the information already contained in an ERD
- Common design errors: misplaced relationships, missing relationships, incorrect cardinalities, overuse of generalization hierarchies, overuse of associative entity types representing M-way relationships, and redundant relationships
- Basic rules to convert entity types and relationships
- Specialized conversion rules to convert optional 1-M relationships, generalization hierarchies, and 1-1 relationships
- Simplified conversion practices in Aqua Data Studio and Visio Professional, one table per entity type with little or no support for generalization hierarchies

QUESTIONS

1. What does it mean to say that constructing an ERD is an iterative process?
2. Why decompose a compound attribute into smaller attributes?
3. When is it appropriate to expand an attribute?
4. Why transform an entity type into two entity types and a relationship?
5. Why transform a weak entity type to a strong entity type?
6. Why transform an entity type into a generalization hierarchy?
7. Why add history to an attribute or relationship?
8. What changes to an ERD are necessary when expanding an attribute?
9. What changes to an ERD are necessary when splitting a compound attribute?
10. What changes to an ERD are necessary when expanding an entity type?
11. What changes to an ERD are necessary when transforming a weak entity type to a strong entity type?
12. What changes to an ERD are necessary when adding unlimited history to an attribute or a relationship?

13. What changes to an ERD are necessary when replacing an entity type with a generalization hierarchy?
14. What should you document about an ERD?
15. What should you omit in ERD documentation?
16. Why are design errors more difficult to detect and resolve than diagram errors?
17. What is a misplaced relationship and how is it resolved?
18. What is an incorrect cardinality and how is it resolved?
19. What is a missing relationship and how is it resolved?
20. What is overuse of a generalization hierarchy and how is it resolved?
21. What is a relationship cycle?
22. What is a redundant relationship and how is it resolved?
23. How is an M-N relationship converted to a table design?
24. How is a 1-M relationship converted to a table design?
25. What is the difference between the 1-M relationship rule and the optional 1-M relationship rule?
26. How is a weak entity type converted to a table design?
27. How is a generalization hierarchy converted to a table design?
28. How is a 1-1 relationship converted to a table design?
29. What are the criteria for choosing a primary key?
30. What should you do if a proposed primary key does not meet selection criteria?
31. Why should you understand the conversion process even if you use a CASE tool to perform the conversion?
32. What are the goals of narrative problem analysis?
33. What are some difficulties with collecting information requirements to develop a business data model?
34. How are entity types identified in a problem narrative?
35. How should the simplicity principle be applied during the search for entity types in a problem narrative?
36. How are relationships and cardinalities identified in a problem narrative?
37. How can you reduce the number of relationships in an initial ERD?
38. What changes to an ERD are necessary when adding limited history to an attribute?
39. How can design documentation help in resolving specification problems?
40. How can design documentation help in improving communication?
41. Why should you not use government identifiers as primary keys?
42. Why are government identifiers sometimes important to store in a database?
43. How does stability help as a guideline for the appropriateness of using generalization?
44. What changes to an ERD are necessary when adding limited history to an attribute?
45. What changes to an ERD are necessary when replacing a collection of entity types with a generalization hierarchy?
46. When should you consider reversed transformations? Are the reversed transformations less frequently used than the normal transformations?
47. What is a hub entity type? How can a hub entity type be used to simplify an ERD?
48. Compare and contrast attribute expansion and entity type expansion?

PROBLEMS

The problems are divided between data modeling problems and conversion problems. Additional conversion problems are found in Chapter 7, where conversion is followed by normalization.

Data Modeling Problems

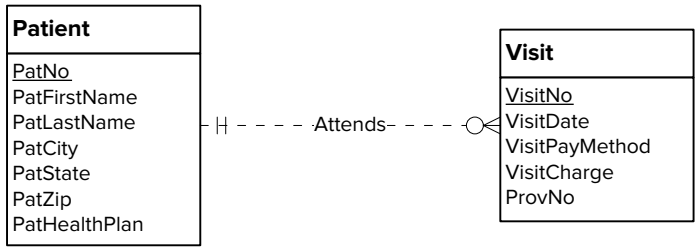
1. Define an ERD for the following narrative. The database should track homes and owners. A home has a unique home identifier, a street address, a city, a state, a zip, a number of bedrooms, a number of bathrooms, and square feet. A home is either owner occupied or rented. An owner has a unique owner number, a Social Security number (used for government reporting requirements), a name, an optional spouse name, a profession, an optional spouse profession, and an optional spouse Social Security number. An owner can possess one or more homes. Each home has only one owner.
2. Refine the ERD from problem 1 by adding an agent entity type. Agents represent owners in the sale of a home. An agent can list many homes, but only one agent can list a home. An agent has a unique agent identifier, a name, an office identifier, a Social Security number (for government reporting only) and a phone number. When an owner agrees to list a home with an agent, a commission (percentage of the sales price) and a selling price are determined.
3. In the ERD from problem 2, transform the attribute, office identifier, into an entity type. Data about an office include the phone number, the manager name, and the address.
4. In the ERD from problem 3, add a buyer entity type. A buyer entity type has a unique buyer identifier, a Social Security number (used for government reporting only), a name, an address, a phone number, optional spouse attributes (name and Social Security number), and preferences for the number of bedrooms and bathrooms, and a price range. An agent can work with many buyers, but a buyer works with only one agent.
5. Refine the ERD from problem 4 with a generalization hierarchy to depict similarities between buyers and owners.
6. Revise the ERD from problem 5 by adding an offer entity type. A buyer makes an offer on a home for a specified sales price. The offer starts on the submission date and time and expires on the specified date and time. A unique offer number identifies an offer. A buyer can submit multiple offers for the same home.
7. Construct an ERD to represent accounts in a database for personal financial software. The software supports checking accounts, credit cards, and two kinds of investments (mutual funds and stocks). No other kinds of accounts are supported, and every account must fall into one of these account types. For each kind of account, the software provides a separate data entry screen. The following list describes the fields on the data entry screens for each kind of account:
   - For all accounts, the software requires the unique account identifier, the account name, date established, and the balance.
   - For checking accounts, the software supports attributes for the bank name, the bank address, the checking account number, and the routing number.
   - For credit cards, the software supports attributes for the credit card number, the expiration date, and the credit card limit.
   - For stocks, the software supports attributes for the stock symbol, the stock type (common or preferred), the last dividend amount, the last dividend

date, the exchange, the last closing price, and the number of shares (a whole number).

- For mutual funds, the software supports attributes for the mutual fund symbol, the share balance (a real number), the fund type (stock, bond, or mixed), the last closing price, the region (domestic, international, or global), and the tax-exempt status (yes or no).

8. Construct an ERD to represent categories in a database for personal financial software. A category has a unique category identifier, a name, a type (expense, asset, liability, or revenue), and a balance. Categories are organized hierarchically so that a category can have a parent category and one or more subcategories. For example, the category "household" can have subcategories for "cleaning" and "maintenance." A category can have any number of levels of subcategories. Make an instance diagram to depict the relationships among categories.
9. Design an ERD for parts and relationships among parts. A part has a unique identifier, a name, and a color. A part can have multiple subparts and multiple parts that use it. The quantity of each subpart should be recorded. Make an instance diagram to depict relationships among parts.
10. Design an ERD to represent a credit card statement. The statement has two parts: a heading containing the unique statement number, the account number of the credit card holder, and the statement date; and a detail section containing a list of zero or more transactions for which the balance is due. Each detail line contains a line number, a transaction date, a merchant name, and the amount of the transaction. The line number is unique within a statement.
11. Modify your ERD from problem 10. Everything is the same except that each detail line contains a unique transaction number in place of the line number. Transaction numbers are unique across statements.
12. Using the ERD in Figure 6.P1, transform the *ProvNo* attribute into an entity type (*Provider*) and a 1-M relationship (*Treats*). A provider has a unique provider number, a first name, a last name, a phone, a specialty, a hospital name in which the provider practices, an e-mail address, a certification, a pay grade, and a title. A provider is required for a visit. New providers do not have associated visits.
13. In the result for problem 12, expand the *Visit* entity type to record details about a visit. A visit detail includes a detail number, a detail charge, an optional provider number, and an associated item. The combination of the visit number and visit detail number is unique for a visit detail. An item includes a unique item number, an item description, an item price, and an item type. An item can be related to multiple visit details. New items may not be related to any visit details. A provider can be related to multiple visit details. Some providers may not be associated to any visit details. In addition, a provider can be related to multiple visits as indicated in problem 12. The provider for a visit detail (usually a nurse or a lab technician) is typically different than the provider for the visit (usually a physician).
14. In the result for problem 13, add a generalization hierarchy to distinguish between nurse and physician providers. A nurse has a pay grade and a title. A physician has a residence hospital, e-mail address, and a certification. The other attributes of provider apply to both physicians and nurses. A visit involves a physician provider while a visit detail may involve a nurse provider.
15. In the result for problem 14, transform *VisitDetail* into a strong entity type with *VisitDetailNo* as the primary key.
16. In the result for problem 15, add a history of item prices. Your solution should support the current price along with the two most recent prices. Include change dates for each item price.

FIRE 6.P1
ERD for Problem 12



17. In the result for problem 15, add a history of item prices. Your solution should support an unlimited number of prices and change dates.
18. Design an ERD with entity types for projects, specialties, and contractors. Add relationships and/or entity types as indicated in the following description. Each contractor has exactly one specialty, but many contractors can provide the same specialty. A contractor can provide the same specialty on multiple projects. A project can use many specialties, and a specialty can be used on many projects. Each combination of project and specialty should have at least two contractors.
19. For the following problem, define an ERD for the initial requirements and then revise the ERD for the new requirements. Your solution should have an initial ERD, a revised ERD, and a list of design decisions for each ERD. In performing your analysis, you may want to follow the approach presented in Section 6.1.

    The database supports the placement office of a leading graduate school of business. The primary purpose of the database is to schedule interviews and facilitate searches by students and companies. Consider the following requirements in your initial ERD:

    - Student data include a unique student identifier, a name, a phone number, an e-mail address, a web address, a major, a minor, and a GPA.
    - The placement office maintains a standard list of positions based on the Labor Department's list of occupations. Position data include a unique position identifier and a position description.
    - Company data include a unique company identifier, a company name, and a list of positions and interviewers. Each company must map its positions into the position list maintained by the placement office. For each available position, the company lists the cities in which positions are available.
    - Interviewer data include a unique interviewer identifier, a name, a phone, an e-mail address, and a web address. Each interviewer works for one company and conducts interviews at the placement office.
    - An interview includes a unique interview identifier, a date, a time, a location (building and room), an interviewer, and a student. A student may have multiple interviews.

    After reviewing your initial design, the placement office decides to revise the requirements. Make a separate ERD to show your refinements. Refine your original ERD to support the following new requirements:

    - Allow companies to use their own language to describe positions. The placement office will not maintain a list of standard positions.
    - Allow companies to indicate availability dates and number of openings for positions.
    - Allow companies to reserve blocks of interview time. The interview blocks will not specify times for individual interviews. Rather a company will request a block of *X* hours during a specified week. Companies reserve interview blocks before the placement office schedules individual

interviews. Thus, the placement office needs to store interviews as well as interview blocks.

- Allow students to submit bids for interview blocks. Students receive a set amount of bid dollars that they can allocate among bids. The bid mechanism is a pseudo-market approach to allocating interviews, a scarce resource. A bid contains a unique bid identifier, a bid amount, and a company. A student can submit many bids and an interview block can receive many bids.

20. For the following problem, define an ERD for the initial requirements and then revise the ERD for the new requirements. Your solution should have an initial ERD, a revised ERD, and a list of design decisions for each ERD. In performing your analysis, you may want to follow the approach presented in Section 6.1.

    Design a database for managing the task assignments on a work order. A work order records the set of tasks requested by a customer at a specified location.

    - A customer has a unique customer identifier, a name, a billing address (street, city, state, and zip), and a collection of submitted work orders.
    - A work order has a unique work order number, a creation date, a date required, a completion date, a customer, an optional supervising employee, a work address (street, city, state, zip), and a set of tasks.
    - Each task has a unique task identifier, a task name, an hourly rate, and estimated hours. Tasks are standardized across work orders so that the same task can be performed on many work orders.
    - Each task on a work order has a status (not started, in progress, or completed), actual hours, and a completion date. The completion date is not entered until the status changes to complete.

    After reviewing your initial design, the company decides to revise the requirements. Make a separate ERD to show your refinements. Refine your original ERD to support the following new requirements:

    - The company wants to maintain a list of materials. The data about materials include a unique material identifier, a name, and an estimated cost. A material can appear on multiple work orders.
    - Each work order uses a collection of materials. A material used on a work order includes the estimated quantity of the material and the actual quantity of the material used.
    - The estimated number of hours for a task depends on the work order and task, not on the task alone. Each task of a work order includes an estimated number of hours.

21. For the following problem, define an ERD for the initial requirements and then revise the ERD for the new requirements. Your solution should have an initial ERD, a revised ERD, and a list of design decisions for each ERD. In performing your analysis, you may want to follow the approach presented in Section 6.1.

    Design a database to assist physical plant personnel in managing assignments of keys to employees. The primary purpose of the database is to ensure proper accounting for all keys.

    - An employee has a unique employee number, a name, a position, and an optional office number.
    - A building has a unique building number, a name, and a location within the campus.
    - A room has a room number, a size (physical dimensions), a capacity, a number of entrances, and a description of equipment in the room. Because

each room is located in exactly one building, the identification of a room depends on the identification of a building.

- Key types (also known as master keys) are designed to open one or more rooms. A room may have one or more key types that open it. A key type has a unique key type number, a date designed, and the employee authorizing the key type. A key type must be authorized before it is created.
- A copy of a key type is known as a key. Keys are assigned to employees. Each key is assigned to exactly one employee, but an employee can hold multiple keys. The key type number plus a copy number uniquely identify a key. The date the copy was made should be recorded in the database.

After reviewing your initial design, the physical plant supervisor decides to revise the requirements. Make a separate ERD to show your refinements. Refine your original ERD to support the following new requirements:

- The physical plant supervisor needs to know not only the current holder of a key but the past holders of a key. For past key holders, the date range that a key was held should be recorded.
- The physical plant supervisor needs to know the current status of each key: in use by an employee, in storage, or reported lost. If lost, the date reported lost should be stored.

22. Define an ERD that supports the generation of product explosion diagrams, assembly instructions, and parts lists. These documents are typically included in hardware products sold to the public. Your ERD should represent the final products as well as the parts comprising final products. The following points provide more details about the documents.

    - Your ERD should support the generation of product explosion diagrams as shown in Figure 6.P2 for a wheelbarrow with a hardwood handle. Your ERD should store the containment relationships along with the quantities required for each subpart. For line drawings and geometric position specifications, you can assume that image and position data types are available to store attribute values.
    - Your ERD should support the generation of assembly instructions. Each product can have a set of ordered steps for instruction. Table 6-P1 shows some of the assembly instructions for a wheelbarrow. The numbers in the instructions refer to the parts diagram.
    - Your ERD should support the generation of a parts list for each product. Table 6-P2 shows the parts list for the wheelbarrow.

23. For the Expense Report ERD shown in Figure 6.P3, identify and resolve errors and note incompleteness in the specifications. Your solution should include a list of errors and a revised ERD. For each error, identify the type of error (diagram or design) and the specific error within each error type. Note that the ERD may have both diagram and design errors. Specifications for the ERD appear below:

    - The Expense Reporting database tracks expense reports and expense report items along with users, expense categories, status codes, and limits on expense category spending.
    - For each user, the database records the unique user number, the first name, the last name, the phone number, the e-mail address, the spending limit, the organizational relationships among users, the submitted expense reports (0 or more), and the expense categories (at least one) available to the user. A user can manage other users but have at most one manager. For each expense category available to a user, there is a limit amount.

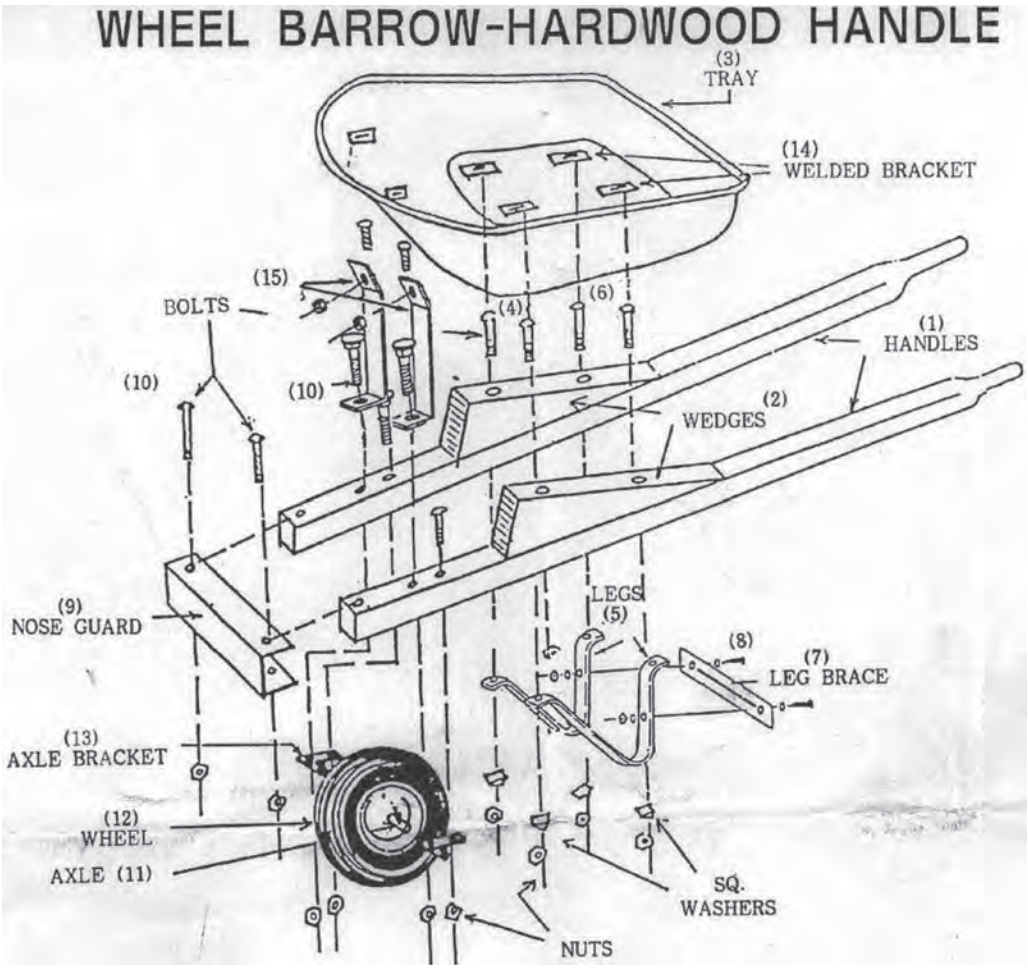


FIGURE 6.P2
Product Explosion Diagram

Step	Instructions
1	Assembly requires a few hand tools, screw driver, box, or open wrench to fit the nuts.
2	Do NOT wrench-tighten nuts until entire wheelbarrow has been assembled.
3	Set the handles (1) on two boxes or two saw horses (one at either end).
4	Place a wedge (2) on top of each handle and align the bolt holes in the wedge with corresponding bolt holes in the handle.

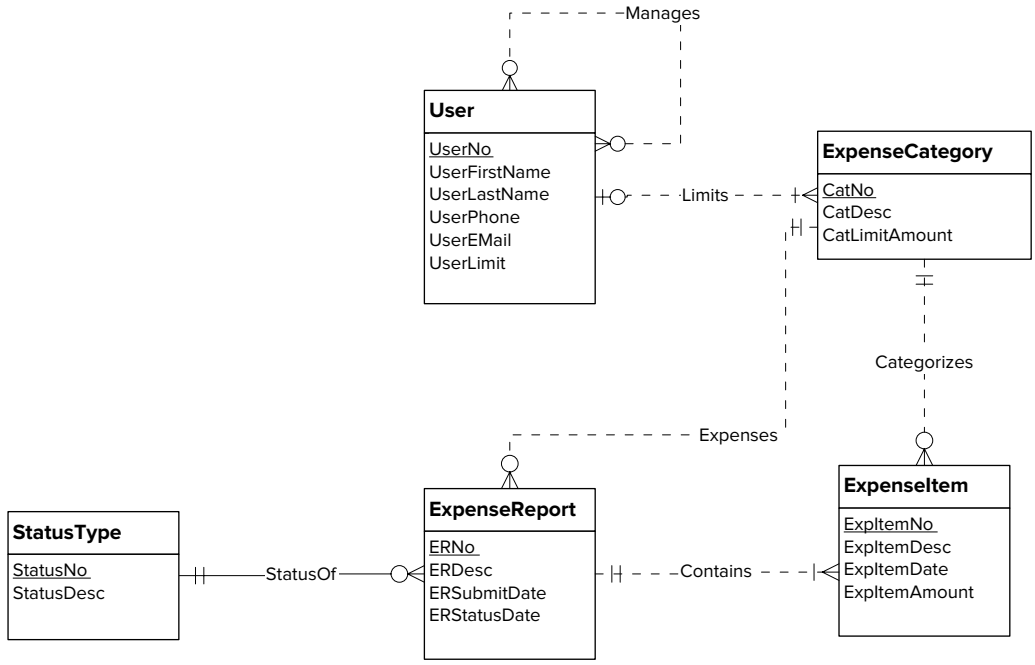
TABLE 6-P1
Sample Assembly Instructions for the Wheelbarrow

- For each expense category, the database records the unique category number, the category description, the spending limit, and the users permitted to use the expense category. When an expense category is initially created, there may not be related users.
- For each status code, the database records the unique status number, the status description, and the expense reports using the status code.
- For each expense report, the database records the unique expense report number, the description, the submitted date, the status date, the status code (required), the user number (required), and the related expense items.
- For each expense item, the database records the unique item number, the description, the expense date, the amount, the expense category (required), and the expense report number (required).

TABLE 6-P2
Partial Parts List for the Wheelbarrow

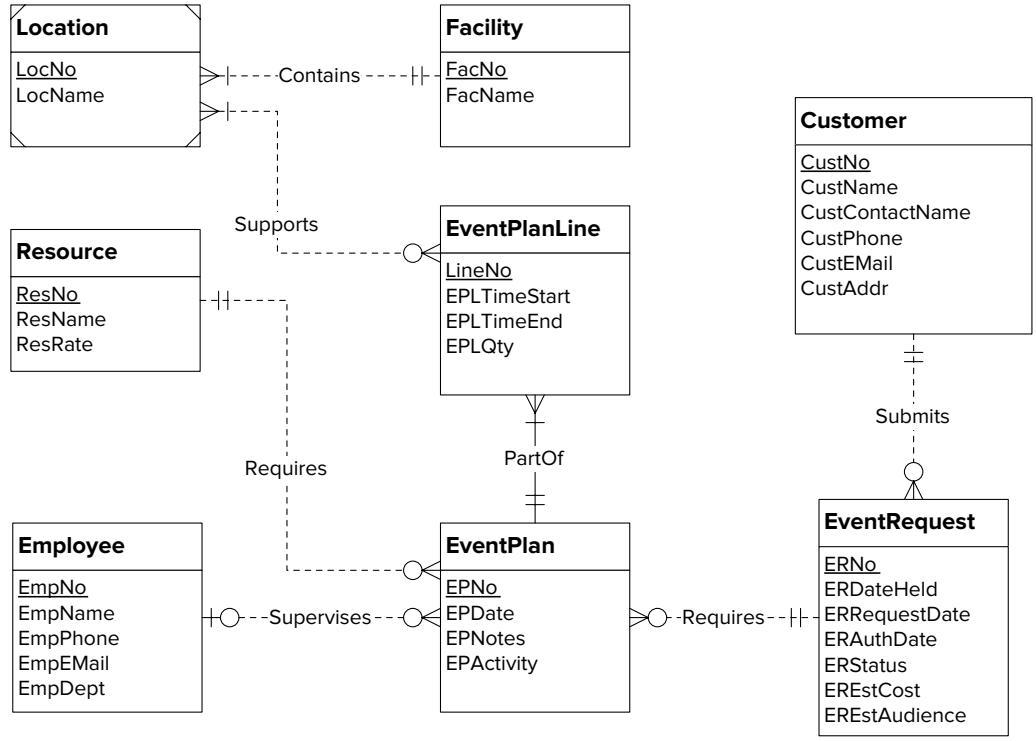
Quantity	Part Description
1	Tray
2	Hardwood handle
2	Hardwood wedge
2	Leg

FIGURE 6.P3
ERD for the Expense Reporting Database



24. For the Intercollegiate Athletic ERD shown in Figure 6.P4, identify and resolve errors and note incompleteness in the specifications. Your solution should include a list of errors and a revised ERD. For each error, identify the type of error (diagram or design) and the specific error within each error type. Note that the ERD may have both diagram and design errors. Specifications for the ERD are as follows:
    - The Intercollegiate Athletic database supports the scheduling and the operation of events along with tracking customers, facilities, locations

FIGURE 6.P4
ERD for the Intercollegiate Athletic Database



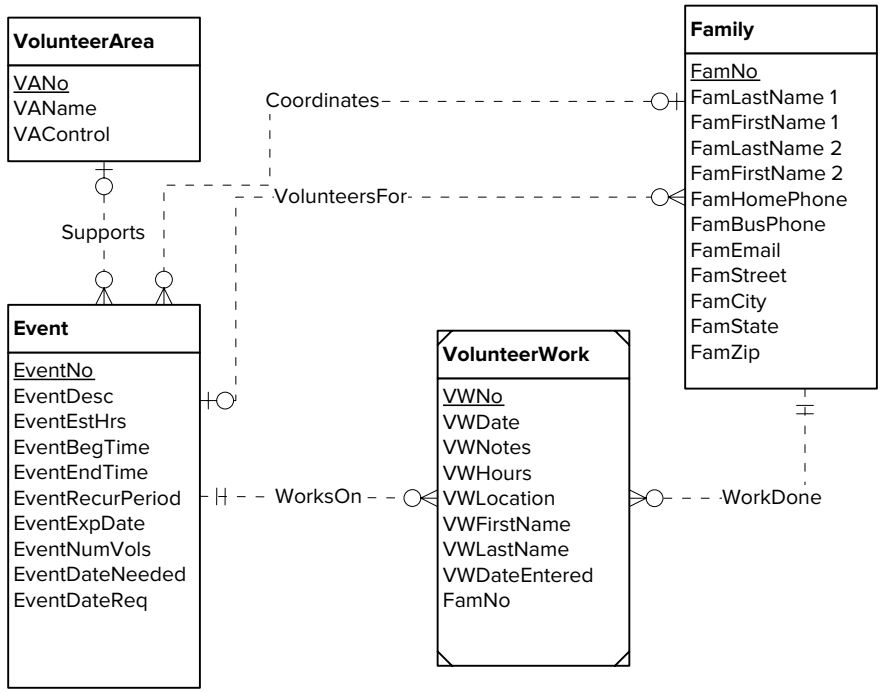
within facilities, employees, and resources to support events. To schedule an event, a customer initiates an event request with the Intercollegiate Athletic Department. If an event request is approved, one or more event plans are made. Typically, event plans are made for the setup, the operation, and the cleanup of an event. An event plan consists of one or more event plan lines.

- For each event request, the database records the unique event number, the date held, the date requested, the date authorized, the status, an estimated cost, the estimated audience, the facility number (required), and the customer number (required).
- For each event plan, the database records the unique plan number, notes about the plan, the work date, the activity (setup, operation, or cleanup), the employee number (optional), and the event number (required).
- For each event plan line, the database records the line number (unique within a plan number), the plan number (required), the starting time, the ending time, the resource number (required), the location number (required), and the quantity of resources required.
- For each customer, the database records the unique customer number, the name, the address, the contact name, the phone, the e-mail address, and the list of events requested by the customer. A customer is not stored in the database until submitting an event request.
- For each facility, the database records the unique facility number, the facility name, and the list of events in which the facility is requested.
- For each employee, the database records the unique employee number, the name, the department name, the email address, the phone number, and the list of event plans supervised by the employee.
- For each location, the database records the related facility number, the location number (unique within a facility), the name, and the list of event plan lines in which the location is used.
- For each resource, the database records the unique resource number, the name, the rental rate, and the list of event plan lines in which the resource is needed.

25. For the Volunteer Information System ERD shown in Figure 6.P5, identify and resolve errors and note incompleteness in the specifications. Your solution should include a list of errors and a revised ERD. For each error, identify the type of error (diagram or design) and the specific error within each error type. Note that the ERD may have both diagram and design errors. Specifications for the ERD are as follows:

    - The Volunteer Information System supports organizations that need to track volunteers, volunteer areas, events, and hours worked at events. The system will be initially developed for charter schools that have mandatory parent participation as volunteers. Volunteers register as a dual or single-parent family. Volunteer coordinators recruit volunteers for volunteer areas. Event organizers recruit volunteers to work at events. Some events require a schedule of volunteers while other events do not use a schedule. Volunteers work at events and record the time worked.
    - For each family, the database records the unique family number, the first and last name of each parent, the home and business phones, the mailing address (street, city, state, and zip), and an optional e-mail address. For single parent households, information about only one parent is recorded.
    - For each volunteer area, the database records the unique volunteer area, the volunteer area name, the group (faculty senate or parent teacher association) controlling the volunteer area, the family coordinating

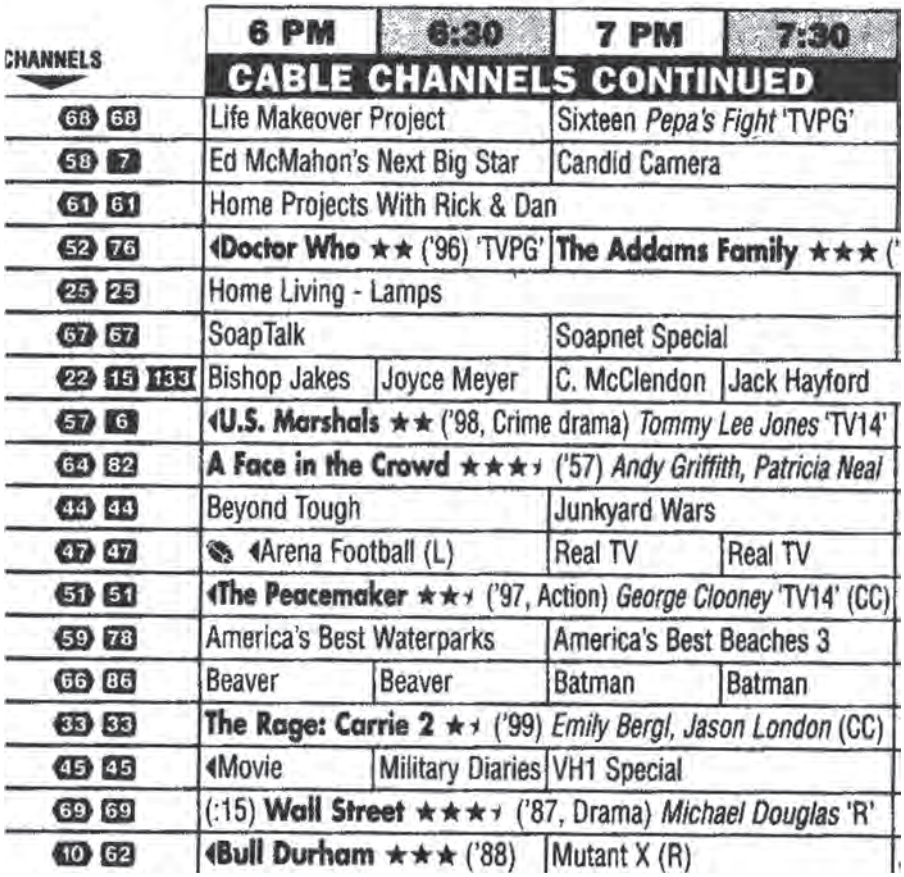
FIGURE 6.P5
ERD for the Volunteer Information System



the volunteer area. In some cases, a family coordinates more than one volunteer area.

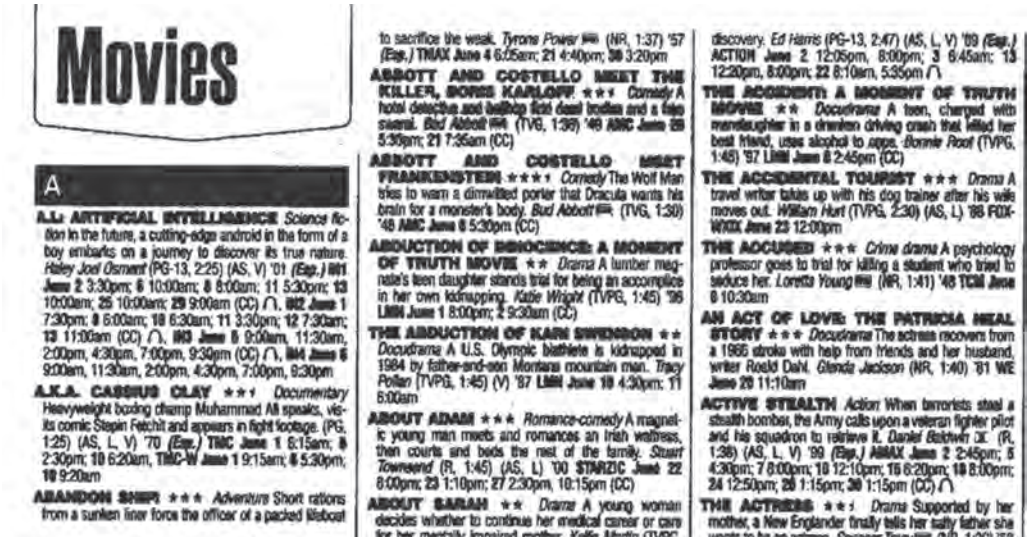
- For events, the database records the unique event number, the event description, the event date, the beginning and ending time of the event, the number of required volunteers, the event period and expiration date if the event is a recurring event, and the list of family volunteers for the event. Families can volunteer in advance for a collection of events.
- After completing a work assignment, hours worked are recorded. The database contains the first and last name of the volunteer, the family in which the volunteer represents, the number of hours worked, the optional event, the date worked, the location of the work, and optional comments. The event is optional to allow volunteer hours for activities not considered as events.

26. Define an ERD that supports the generation of television viewing guides, movie listings, sports listings, public access listings, and cable conversion charts. These documents are typically included in television magazines bundled with Sunday newspapers. In addition, these documents are available online. The following points provide more details about the documents.
    - A television viewing guide lists the programs available during each time slot of a day as depicted in Figure 6.P6. For each program in a channel/time slot, a viewing guide may include some or all of these attributes: a program title, a television content rating, a description, a rerun status (yes or no), a duration, a closed caption status (yes or no), and a starting time if a program does not begin on a half-hour increment. For each movie, a guide also may include some or all of these attributes: an evaluative rating (number of stars from 1 to 4, with half-star increments), a list of major actors, an optional brief description, a motion picture content rating, and a release year. Public access programs are shown in a public access guide, not in a television viewing guide.
    - A movie listing contains all movies shown in a television guide as depicted in Figure 6.P7. For each movie, a listing may include some or all of these



CHANNELS	6 PM	6:30	7 PM	7:30
	CABLE CHANNELS CONTINUED			
68 68	Life Makeover Project		Sixteen *Pepa's Fight* 'TVPG'	
58 7	Ed McMahon's Next Big Star		Candid Camera	
61 61	Home Projects With Rick & Dan			
52 76	◂**Doctor Who** ★★ ('96) 'TVPG'		**The Addams Family** ★★★ ('	
25 25	Home Living - Lamps			
67 67	SoapTalk		Soapnet Special	
22 15 133	Bishop Jakes	Joyce Meyer	C. McClendon	Jack Hayford
57 6	◂**U.S. Marshals** ★★ ('98, Crime drama) *Tommy Lee Jones* 'TV14'			
64 82	**A Face in the Crowd** ★★★½ ('57) *Andy Griffith, Patricia Neal*			
44 44	Beyond Tough		Junkyard Wars	
47 47	◂Arena Football (L)		Real TV	Real TV
51 51	◂**The Peacemaker** ★★½ ('97, Action) *George Clooney* 'TV14' (CC)			
59 78	America's Best Waterparks		America's Best Beaches 3	
66 86	Beaver	Beaver	Batman	Batman
33 33	**The Rage: Carrie 2** ★½ ('99) *Emily Bergl, Jason London* (CC)			
45 45	◂Movie	Military Diaries	VH1 Special	
69 69	(:15) **Wall Street** ★★★½ ('87, Drama) *Michael Douglas* 'R'			
10 62	◂**Bull Durham** ★★★ ('88)		Mutant X (R)	

FIGURE 6.P6
Sample Television Viewing Guide



Movies

A

A.I.: ARTIFICIAL INTELLIGENCE *Science fiction* In the future, a cutting-edge android in the form of a boy embarks on a journey to discover its true nature. *Haley Joel Osment* (PG-13, 2:25) (AS, V) '01 *(Exp.)* WH June 2 3:30pm; 6 10:00am; 8 8:00am; 11 5:30pm; 13 10:00am; 25 10:00am; 29 9:00am (CC) ∩. WZ June 1 7:30pm; 8 6:00am; 10 6:30am; 11 3:30pm; 12 7:30am; 13 11:00am (CC) ∩. WB June 5 9:00am, 11:30am, 2:00pm, 4:30pm, 7:00pm, 9:30pm (CC) ∩. WA June 6 9:00am, 11:30am, 2:00pm, 4:30pm, 7:00pm, 9:30pm

A.K.A. CASSIUS CLAY ★★½ *Documentary* Heavyweight boxing champ Muhammad Ali speaks, visits comic Stepin Fetchit and appears in fight footage. (PG, 1:25) (AS, L, V) '70 *(Exp.)* TMC June 1 8:15am; 8 2:30pm; 10 6:20am, TMC-W June 1 9:15am; 8 5:30pm; 10 9:20am

ABANDON SHIP ★★★ *Adventure* Short rations from a sunken liner force the officer of a packed lifeboat to sacrifice the weak. *Tyrone Power* (NR, 1:37) '57 *(Exp.)* TMAX June 4 6:05am; 21 4:40pm; 30 3:20pm

ABBOTT AND COSTELLO MEET THE KILLER, BORIS KARLOFF ★★½ *Comedy* A hotel detective and bellhop fight dead bodies and a fake sawai. *Bud Abbott* (TVG, 1:36) '49 ABC June 29 5:30pm; 21 7:35am (CC)

ABBOTT AND COSTELLO MEET FRANKENSTEIN ★★★½ *Comedy* The Wolf Man tries to warn a dimwitted porter that Dracula wants his brain for a monster's body. *Bud Abbott* (TVG, 1:30) '48 ABC June 6 5:30pm (CC)

ABDUCTION OF INNOCENCE: A MOMENT OF TRUTH MOVIE ★★ *Drama* A lumber magnate's teen daughter stands trial for being an accomplice in her own kidnapping. *Katie Wright* (TVPG, 1:45) '96 LMN June 1 8:00pm; 2 9:30am (CC)

THE ABDUCTION OF KARI SWENSON ★★ *Docudrama* A U.S. Olympic biathlete is kidnapped in 1984 by father-and-son Montana mountain man. *Tracy Pollan* (TVPG, 1:45) (V) '97 LMN June 10 4:30pm; 11 6:00am

ABOUT ADAM ★★★ *Romance-comedy* A magnetic young man meets and romances an Irish waitress, then courts and beds the rest of the family. *Stuart Townsend* (R, 1:45) (AS, L) '00 STARZC June 22 6:00pm; 23 1:10pm; 27 2:30pm, 10:15pm (CC)

ABOUT SARAH ★★ *Drama* A young woman decides whether to continue her medical career or care

discovery. *Ed Harris* (PG-13, 2:47) (AS, L, V) '99 *(Exp.)* ACTION June 2 12:05pm, 8:00pm; 3 6:45am; 13 12:20pm, 8:00pm; 22 8:10am, 5:35pm ∩

THE ACCIDENT: A MOMENT OF TRUTH MOVIE ★★ *Docudrama* A teen, charged with manslaughter in a drunken driving crash that killed her best friend, uses alcohol to oppo. *Bonnie Root* (TVPG, 1:45) '97 LMN June 8 2:45pm (CC)

THE ACCIDENTAL TOURIST ★★★ *Drama* A travel writer takes up with his dog trainer after his wife moves out. *William Hurt* (TVPG, 2:30) (AS, L) '88 FOX-WHX June 23 12:00pm

THE ACCUSED ★★★ *Crime drama* A psychology professor goes to trial for killing a student who tried to seduce her. *Loretta Young* (NR, 1:41) '48 TCM June 8 10:30am

AN ACT OF LOVE: THE PATRICIA NEAL STORY ★★★ *Docudrama* The actress recovers from a 1965 stroke with help from friends and her husband, writer Roald Dahl. *Glenda Jackson* (NR, 1:40) '81 WE June 29 11:10am

ACTIVE STEALTH *Action* When terrorists steal a stealth bomber, the Army calls upon a veteran fighter pilot and his squadron to retrieve it. *Daniel Baldwin* (R, 1:38) (AS, L, V) '99 *(Exp.)* ABMX June 2 2:45pm; 5 4:30pm; 7 8:00pm; 10 12:10pm; 16 6:20pm; 18 8:00pm; 24 12:50pm; 26 1:15pm; 30 1:15pm (CC) ∩

THE ACTRESS ★★½ *Drama* Supported by her mother, a New Englander finally tells her salty father she

FIGURE 6.P7
Sample Movie Listing

attributes: a title, a release year, an evaluative rating, a content rating, a channel abbreviation, a list of days of the week/time combinations, a list of major actors, and a brief description. A movie listing is organized in ascending order by movie titles.

- A sports listing contains all sports programming in a television guide as depicted in Figure 6.P8. A sports listing is organized by sport and day within a sport. Each item in a sports listing may include some or all of these

FIGURE 6.P8
Sample Sports Listing

Sports

8:00pm **GOLF** Golf Murphy's Irish Open, First Round (R)
11:00pm **GOLF** Golf Murphy's Irish Open, First Round (R)

FRIDAY, JUNE 28
10:00am **GOLF** Golf Murphy's Irish Open, Second Round (L)
12:00pm **ESPN** U.S. Senior Open, Second Round (L) (CC)
2:00pm **ESPN** PGA FedEx St. Jude Classic, Second Round (L) (CC)
3:00pm **GOLF** Golf ShopRite LPGA Classic, First Round (L)
4:00pm **ESPN** Golf U.S. Senior Open, Second Round (L) (CC)
5:30pm **GOLF** Scorecard Report
8:00pm **GOLF** Golf ShopRite LPGA Classic, First Round (R)
10:00pm **GOLF** Scorecard Report
11:00pm **GOLF** Golf Murphy's Irish Open, Second Round (R)

SATURDAY, JUNE 29
10:00am **GOLF** Golf Murphy's Irish Open, Third Round (L)
3:00pm **NBC-WLWT** Golf U.S. Senior Open, Third Round (L) (CC)
4:00pm **ABC-WCPO** PGA FedEx St. Jude Classic, Third Round (L)
4:30pm **GOLF** Golf ShopRite LPGA Classic, Second Round (L)
7:00pm **GOLF** Scorecard Report
8:00pm **GOLF** Haskins Award
8:30pm **GOLF** Golf ShopRite LPGA Classic, Second Round (R)
10:00pm **GOLF** Scorecard Report
11:00pm **GOLF** Haskins Award
11:30pm **GOLF** Golf Murphy's Irish Open, Third Round (R)

HORSE EVENTS

SUNDAY, JUNE 2
2:00pm **ESPN2** Equestrian Del Mar National (CC)

WEDNESDAY, JUNE 5
2:00pm **ESPN2** Wire to Wire

FRIDAY, JUNE 7
5:00pm **ESPN2** Horse Racing Acorn Stakes (L)

SATURDAY, JUNE 8
2:00pm **ESPN** Horse Racing Belmont Stakes Special (L) (CC)
5:00pm **NBC-WLWT** Horse Racing Belmont Stakes (L) (CC)

WEDNESDAY, JUNE 12
2:00pm **ESPN2** Wire to Wire

SATURDAY, JUNE 15
5:00pm **CBS-WKRC** Horse Racing Stephen Foster Handicap (L)

WEDNESDAY, JUNE 19
2:00pm **ESPN2** Wire to Wire

WEDNESDAY, JUNE 26
2:00pm **ESPN2** Wire to Wire

SATURDAY, JUNE 29
3:00pm **ESPN2** Budweiser Grand Prix of Devon
5:00pm **CBS-WKRC** Horse Racing The Mothergoose (L) (CC)
11:00pm **ESPN2** 2Day at the Races (L)

MARTIAL ARTS

SATURDAY, JUNE 1
10:00pm **IN2** World Championship Kickboxing Bad to the Bone (L)

MONDAY, JUNE 3
9:00pm **IN2** World Championship Kickboxing Bad to the Bone (R)

SUNDAY, JUNE 16
9:00pm **IN1** Ultimate Fighting Championship: Ultimate Royce Gracie

MONDAY, JUNE 17
1:00am **IN2** Ultimate Fighting Championship: Ultimate Royce Gracie
11:30pm **IN2** Ultimate Fighting Championship: Ultimate Royce Gracie

attributes: an event title, a time, a duration, a channel, an indicator for closed-captioning, an indicator if live, and an indicator if a rerun.

- A public access listing shows public access programming that does not appear elsewhere in a television guide as depicted in Figure 6.P9. A public access listing contains a list of community organizations (title, area, street address, city, state, zip code, and phone number). After the listing of community organizations, a public access listing contains programming for each day/time slot. Because public access shows do not occupy all time slots and are available on one channel only, there is a list of time slots for each day, not a grid as for a complete television guide. Each public access program has a title and an optional sponsoring community organization.

PUBLIC ACCESS

Public Access listings for Channel 24 in all Time Warner franchises in Greater Cincinnati:
Media Bridges Cincinnati, 1100 Race St., Cincinnati 45210. 651-4171.
Waycross Community Media (Forest Park-Greenhills-Springfield Twp.), 2086 Waycross Road, Forest Park 45240. 825-2429.
Intercommunity Cable Regulatory Commission, 2492 Commodity Circle, Cincinnati 45241. 772-4272.
Norwood Community Television, 2020 Sherman Ave., Norwood 45212. 396-5573.

SUNDAY
7 a.m. – Heart of Compassion
7:30 a.m. – Community Pentecostal
8 a.m. – ICRC Programming
8:30 a.m. – ICRC Programming
9 a.m. – St. John Church of Christ
10 a.m. – Beulah Missionary Baptist

MONDAY
6 a.m. – Sonshine Gospel Hour
7 a.m. – Latter Rain Ministry
8 a.m. – Dunamis of Faith
8:30 a.m. – In Jesus' Name
9 a.m. – Happy Gospel Time TV
10 a.m. – Greek Christian Hour
10:30 a.m. – Armor of God
11 a.m. – Delhi Christian Center
Noon – Humanist Perspective
12:30 p.m. – Waterway Hour
1:30 p.m. – Country Gospel Jubilee
2:30 p.m. – Know Your Government
4:30 p.m. – House of Yisrael
5:30 p.m. – Living Vine Presents
6:30 p.m. – Family Dialogue
7 p.m. – Goodwill Talks
8 p.m. – Pastor Nadie Johnson
9 p.m. – Delta Kings Barbershop Show
Midnight – Basement Flava 2
1 a.m. – Total Chaos Hour
2 a.m. – Commissioned by Christ
3 a.m. – From the Heart
3:30 a.m. – Words of Farrakhan
4:30 a.m. – Skyward Bound
11:30 p.m. – Fire Ball Ministry Church of God
12:30 a.m. – Second Peter Pentecostal Church
1:30 a.m. – Road to Glory Land
3:30 a.m. – Shadows of the Cross

WEDNESDAY
6 a.m. – Pure Gospel
8 a.m. – ICRC Programming
8:30 a.m. – Way of the Cross
9 a.m. – Church of Christ Hour
10 a.m. – A Challenge of Faith
10:30 a.m. – Miracles Still Happen
11 a.m. – Deerfield Digest
11:30 a.m. – Bob Schuler
Noon – Friendship Baptist Church
2 p.m. – Business Talk
2:30 p.m. – ICRC Programming
3 p.m. – ICRC Programming
3:30 p.m. – Temple Fitness
4 p.m. – Church of God
5 p.m. – Around Cincinnati
5:30 p.m. – Countering the Silence
6 p.m. – Community Report
6:30 p.m. – ICRC Programming
7 p.m. – Inside Springdale
8 p.m. – ICRC Sports

FIGURE 6.P9
Sample Public Access Listing

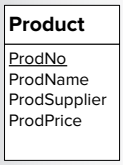
- A cable/conversion chart shows the mapping of channels across cable systems as depicted in Figure 6.P10. For each channel, a conversion chart shows a number on each cable system in the local geographic area.

CABLE CONVERSION CHART

	Time Warner standard	Time Warner upgrade cable ready	Insight	Adelphia Amelia	Fairfield, Middletown	Time Warner Hamilton,	Adelphia Delhi
5	5	6	7	5	5		6
9	9	7	8	9	9		10
12	12	13	3	12	12		13
19	3	3	4	3	13		2
25	20	20–	25	25	20		15
48	13	8	13	6	6		8
64	11	11	11	11	11		11
2	–	–	–	–	2		–
7		–	–	–	7		–
14	14	14	14	14	14		14
16	16	16	–	16	16		16
22	–	–	–	–	8		–
43	–	–	–	–	3		–
45	–	–	–	–	10		–
54	21	21	2	–	–		4
A&E	39	39	52	28	27		46
AMC	46	46	31	29	26		40

FIGURE 6.P10
Sample Conversion Chart

FIGURE 6.P11
Product Entity Type without Price History



27. Transform the ERD in Figure 6.P11 by adding unlimited history for the *ProdPrice* attribute.
28. Transform the ERD in Figure 6.P11 by adding limited history for the *ProdPrice* attribute. The transformed ERD should support the current price and the two most recent prices.
29. Transform the ERD in Figure 6.P12 by adding unlimited history for the *WorksAt* 1-M relationship.
30. Transform the ERD in Figure 6.P13 by adding unlimited history for the *Shares* M-N relationship. The Shares relationship represents a timesharing situation in which owners have fractional ownership for a number of consecutive weeks of a property per year.
31. This problem involves relationships among bookings, vehicles, and customers for auto maintenance. Customers own a collection of vehicles but a vehicle is owned by exactly one customer. A booking involves a vehicle brought to an auto shop by a customer. A vehicle may have a collection of bookings over time. The customer that makes a booking is always the owner of the vehicle. Draw an ERD to represent the relationships among vehicles, customers, and bookings. You can assume that each entity type has its own primary key.
32. Draw an ERD for the situation in problem 31 except that the customer who makes the booking may be different than the owner of the vehicle for the booking.
33. Draw an ERD to track lab tests performed by a medical laboratory on clients. The database should track basic client details including a unique client identifier, client name, client insurance provider (if any), client address, client date of birth, and client sex. The database should track the unique identifier for the lab test, the test type identifier, the date and time when the lab test was administered, and the identifier of the lab employee performing the lab test. A client can request multiple tests in a visit to the lab. The database only contains clients who have had lab tests performed. Each lab test is administered to one client.
34. Revise the ERD from problem 33 with more details about test types. A test type includes a unique test type identifier, a test type name, a test cost, and a test type code. A test for a client is associated with one test type. A test type can be used in multiple tests given to clients. A test type can exist in the database without ever being used in a test given to a client.
35. Revise the ERD from problem 34 with test type items. A test type item includes a unique test item identifier, a test item name, test item unit of measure, and a test item description. A test type includes one or more test items. A test item can be part of one or more test types.
36. Revise the ERD from problem 35 to include lab test results. Each lab test given to a client has a result for each item on the test type. A test item result includes a unique test item result identifier, the analysis date, the employee identifier of

FIGURE 6.P12
WorksAt Relationship without History

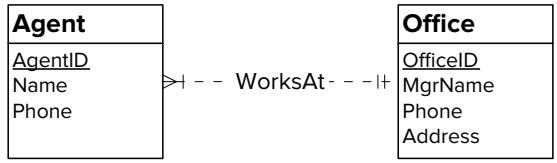


FIGURE 6.P13
WorksAt Relationship without History

the lab technician measuring the result, the measured value of the test item, and an optional description of the test item's appearance. An lab test has one or more associated test item results although there are no associated results until lab analysis is complete. A test item can have many associated test item results. Test items never administered will not have any associated test item results.

37. Analyze the ERD from problem 36 for cycles among entity types. Identify any cycles in the ERD. Does the cycle contain redundant relationships? Explain your answer.
38. For the Entertainment Viewing ERD shown in Figure 6.P14, identify and resolve errors and note incompleteness in the specifications. Your solution should include a list of errors and a revised ERD. For each error, identify the type of error (diagram or design) and the specific error within each error type. Note that the ERD may have both diagram and design errors. Specifications for the ERD are as follows:
    - The Entertainment Viewing database supports entry of the viewing habits of users and queries about movies, television shows, actors, and user preferences for actors.
    - For movies, the database should record the unique movie identifier, unique title, genre, list of noteworthy actors, director, format, studio, duration, date released to theaters, and list of subtitle languages.
    - For shows, the database should record the unique show identifier, unique title, duration, list of noteworthy actors, network, optional scheduled time (day and start time), date of first viewing, and number of seasons of shows.
    - For actors, the database should record the unique actor identifier, first name, last name, age, sex, and list of awards. An actor can appear in many shows and movies.
    - For users, the database records the unique user identifier, unique email address, first name, last name, postal code, sex, country, and age group.
    - For viewing habits, the database should record the user, show or movie viewed, date/time viewed, and method of viewing (broadcast time or recorded).
39. For the Auto Dealership ERD shown in Figure 6.P15, identify and resolve errors and note incompleteness in the specifications. Your solution should include a list

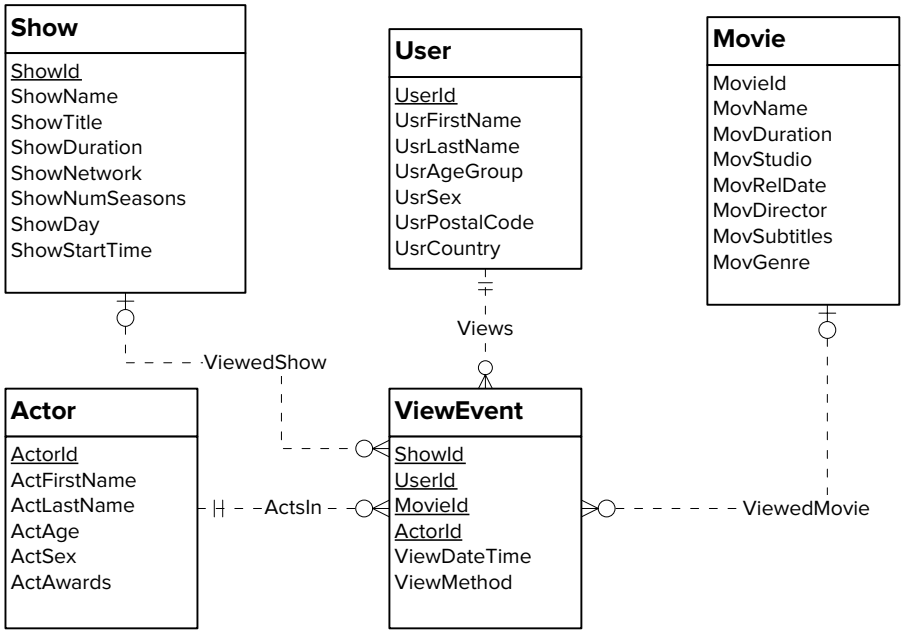
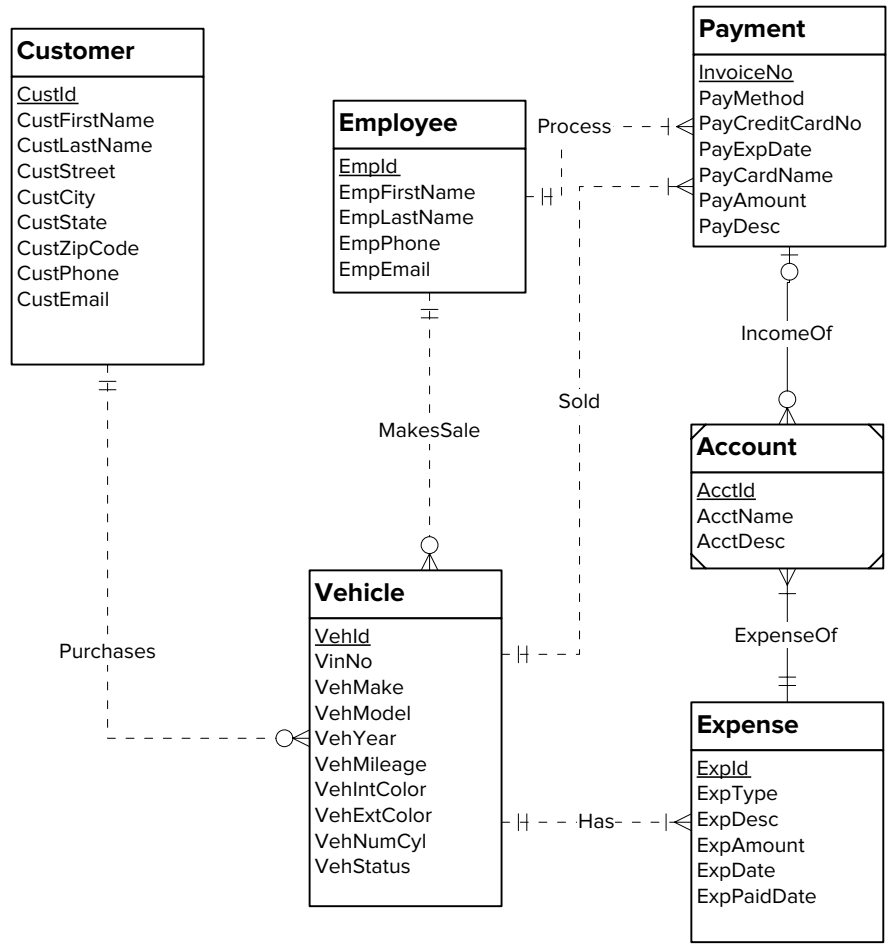


FIGURE 6.P14
ERD for the Entertainment Viewing Database

FIGURE 6.P15
ERD for the Auto Dealership Database



of errors and a revised ERD. For each error, identify the type of error (diagram or design) and the specific error within each error type. Note that the ERD may have both diagram and design errors. Specifications for the ERD are presented in the following narrative.

Mountain High Quality Vehicles serves a metropolitan market with a medium size inventory of pre-owned cars and trucks. The vehicle inventory includes a variety of makes and models such as Acura, Chrysler, BMW, Cadillac, Ford, Chevrolet, Toyota, Honda, Mercedes-Benz, and more. A small staff manages the major functions of the business, purchasing, transporting, marketing, cleaning, maintaining, and selling the vehicles. They carefully inspect and certify the vehicles before they are available to the public for sale.

The dealership would like to develop an inventory management database to improve its tracking of vehicles, sales, and expenses. The dealership also would like to track information about its customers and car(s) sold to its customers.

- **Vehicle Acquisitions:** Periodically the owners attend auctions and purchase pre owned cars seeking reasonable prices and quality vehicles. They also purchase pre-owned vehicles from the wholesale market. The purchased vehicles are transported to the dealership and inspected for mechanical problems. Each vehicle is fixed and cleaned before being placed for sale.
- **Vehicle Improvements:** Apart from purchases, the dealership has additional expenses to prepare vehicles for market. The expenses typically involve transporting the purchased vehicle to the dealership, checking the vehicle for any potential problem, repairs and maintenance if necessary, marketing and cleaning.

- **Sales Details:** Customers purchase vehicles at the dealership. Each sale involves one customer even for married couples. Although customers can purchase more than one vehicle, each vehicle is recorded as a separate sale. When a sale is completed, the employee associated with the sale and payments are recorded. Typically, vehicles remain on the lot for a period of time before sales occur.
- **Vehicle Details:** The database tracks the unique vehicle identifier and vehicle identification number (VIN) to complete a sales transaction. The database also tracks vehicle characteristics such as make, model, year, mileage, exterior and interior colors, transmission type (automatic or manual), and number of cylinders (4 or 6).
- **Customer Details:** The database records the unique customer number, first and last names, address, city, state, postal code, primary phone number, and cellphone number.
- **Expense Details:** Each vehicle expense has a unique expense identifier, expense type, expense description, expense amount, expense paid date, account, and associated vehicle.
- **Account Details:** The database tracks account details such as a unique account identifier, account description, related expenses, and related payments.
- **Payment Details:** The database also tracks the vehicle sale (payment process). Each payment has a unique invoice number and payment method. The payment options are cash, credit card, or external line of credit as no financing is available at the dealership. Typically one payment is made per sale although multiple payments are sometimes made if a customer provides cash for part of the sale. If a customer is paying with a credit card, the payment includes the credit card number, expiration date, name on the credit card, and payment description. The employee that completed the sales transaction and vehicle should be recorded. The same employee works as sales associate and processes the payment to complete the sale. Each payment is associated with one account for company accounting purposes.

40. The ERD in Figure 6.P16 was used in a banking transaction benchmark (TPC-B) developed by the Transaction Processing Council (TPC) in the 1990s. You should analyze the specifications for the benchmark database to determine if there are any design errors. Your solution should include a list of design errors and a revised ERD. Specifications for the ERD are presented in the following narrative.

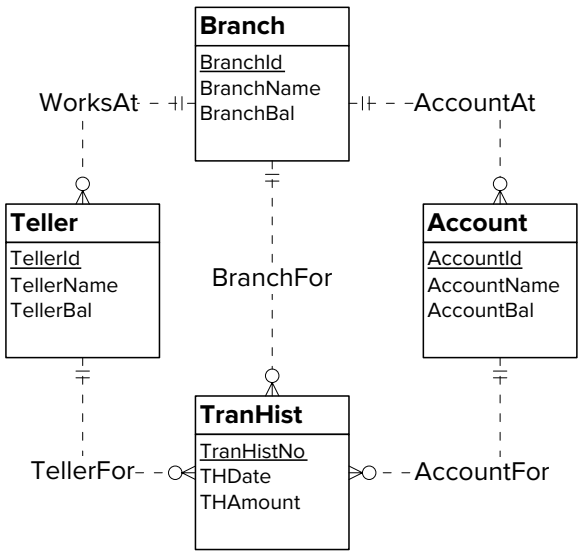


FIGURE 6.P16
ERD for the Bank Transaction Benchmark

- Each branch employs a collection of tellers. A teller is assigned to exactly one branch.
- Each account is opened at exactly one branch. A branch may be the home for many accounts.
- The bank keeps an historical record of each transaction. A transaction record identifies the teller, account, and branch. The branch associated with a transaction always matches the teller's branch. In a home transaction, the branch of the account and teller match. In a remote transaction, the branch of a teller and account are different. Historical transaction records contain both home and remote transactions.

Conversion Problems

1. Convert the ERD shown in Figure 6.CP1 into tables. List the conversion rules used and the resulting changes to the tables.
2. Convert the ERD shown in Figure 6.CP2 into tables. List the conversion rules used and the resulting changes to the tables.
3. Convert the ERD shown in Figure 6.CP3 into tables. List the conversion rules used and the resulting changes to the tables.
4. Convert the ERD shown in Figure 6.CP4 into tables. List the conversion rules used and the resulting changes to the tables.
5. Convert the ERD shown in Figure 6.CP5 into tables. List the conversion rules used and the resulting changes to the tables.
6. Convert the ERD shown in Figure 6.CP6 into tables. List the conversion rules used and the resulting changes to the tables.
7. Convert the ERD shown in Figure 6.CP7 into tables. List the conversion rules used and the resulting changes to the tables.
8. Convert the ERD shown in Figure 6.CP8 into tables. List the conversion rules used and the resulting changes to the tables.
9. Convert the ERD shown in Figure 6.CP9 into tables. List the conversion rules used and the resulting changes to the tables.

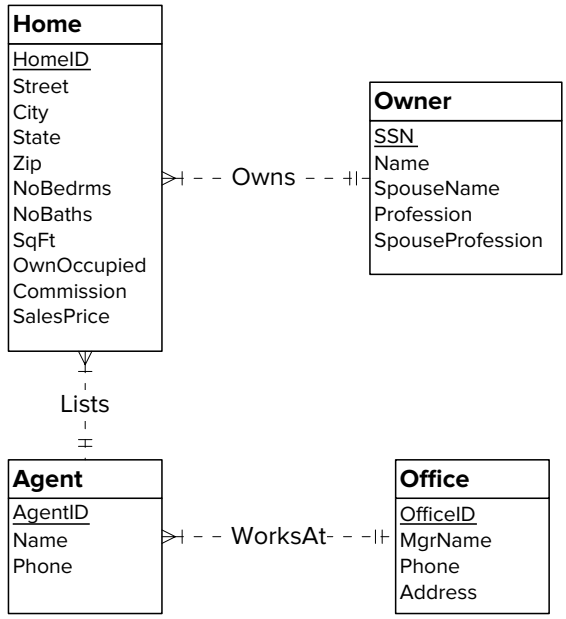


FIGURE 6.CP1
ERD for Conversion Problem 1

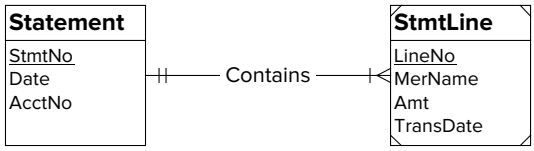


FIGURE 6.CP2
ERD for Conversion Problem 2

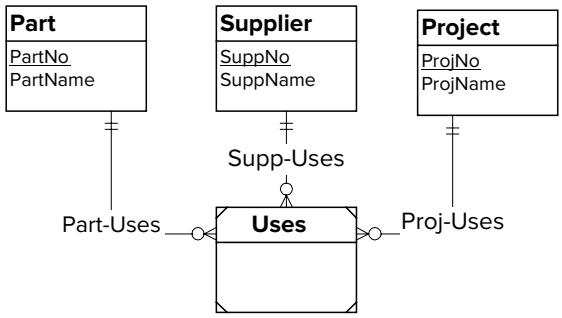


FIGURE 6.CP3
ERD for Conversion Problem 3

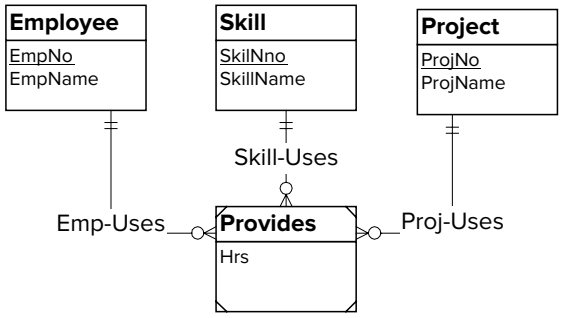


FIGURE 6.CP4
ERD for Conversion Problem 4

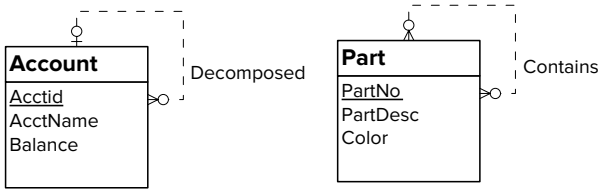


FIGURE 6.CP5
ERD for Conversion Problem 5

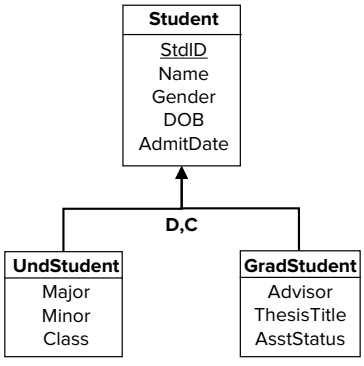


FIGURE 6.CP6
ERD for Conversion Problem 6

FIGURE 6.CP7
ERD for Conversion Problem 8

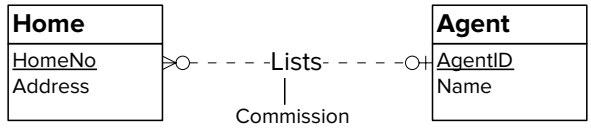


FIGURE 6.CP8
ERD for Conversion Problem 8

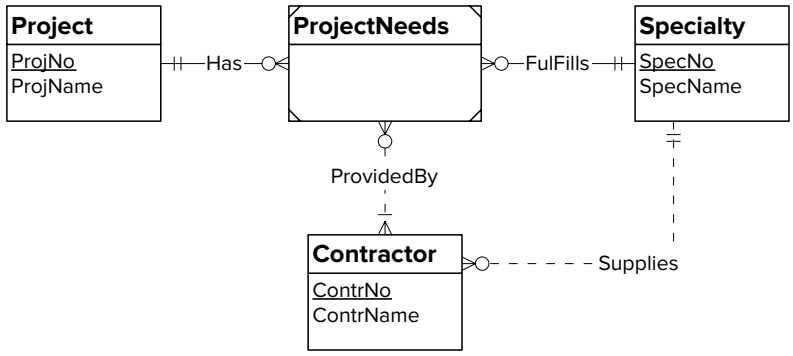
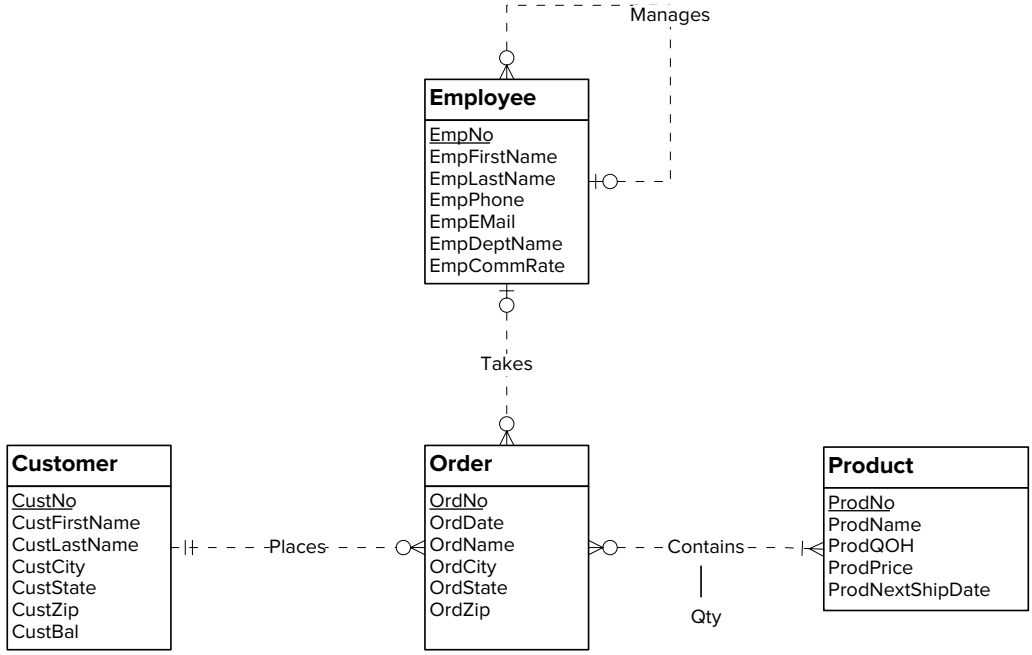


FIGURE 6.CP9
ERD for Conversion Problem 9



10. Convert the ERD shown in Figure 6.CP10 into tables. List the conversion rules used and the resulting changes to the tables.
11. Convert the ERD shown in Figure 6.CP11 into tables. List the conversion rules used and the resulting changes to the tables.

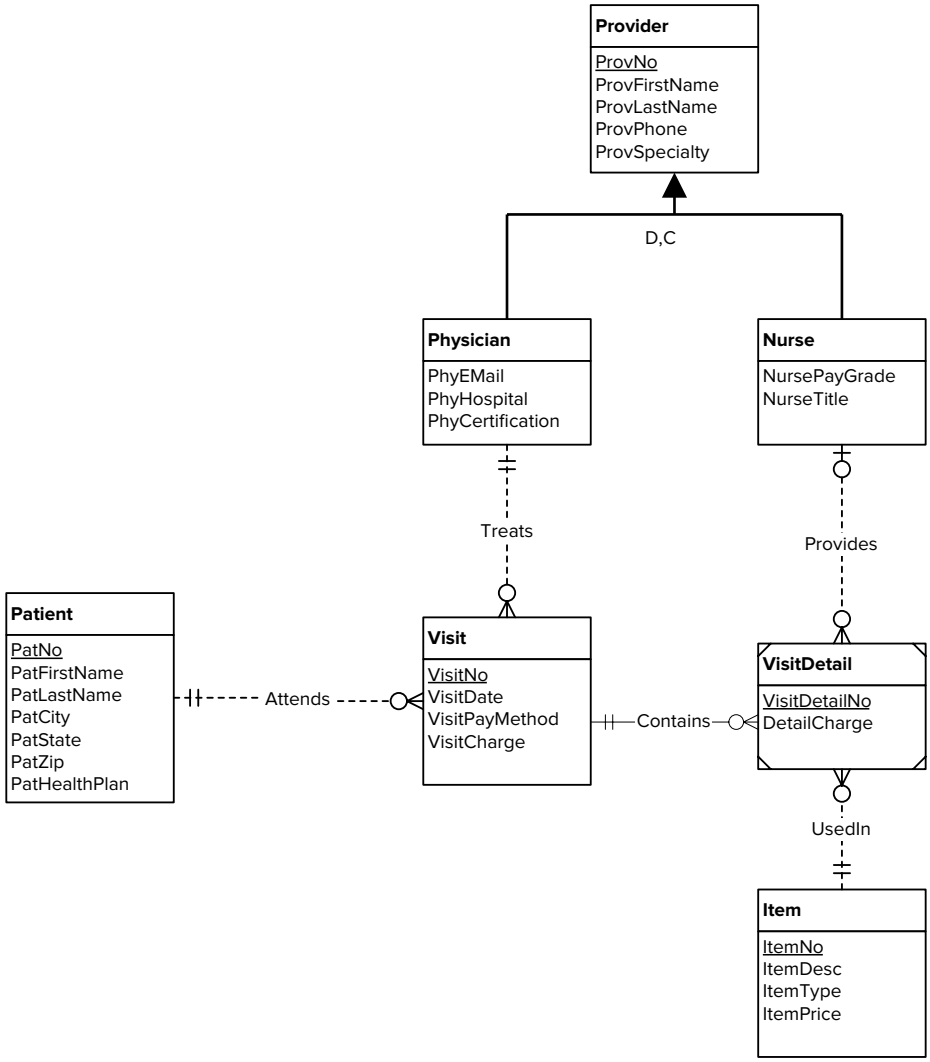


FIGURE 6.C10
ERD for Conversion Problem 10

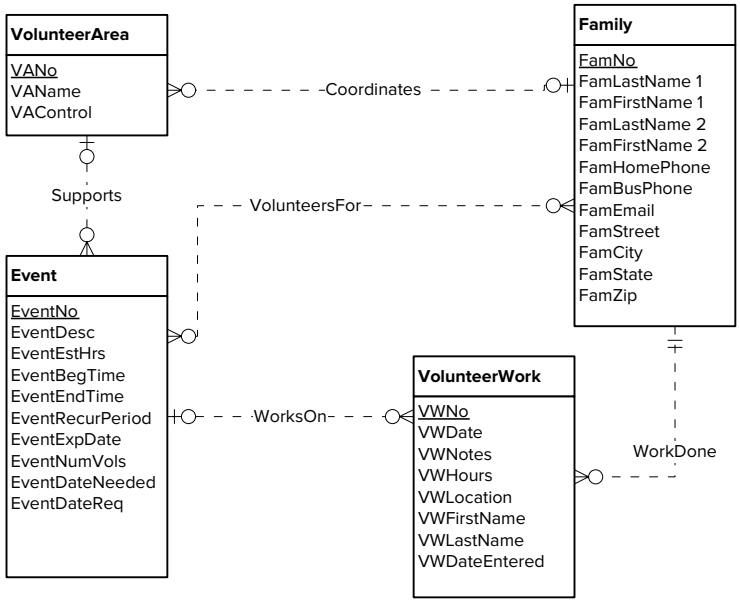


FIGURE 6.CP11
ERD for Conversion Problem 11

REFERENCES FOR FURTHER STUDY

Chapter 3 of Batini, Ceri, and Navathe (1992) and Chapter 10 of Nijssen and Halpin (1989) provide more details on transformations to refine an ERD. For more details about conversion of generalization hierarchies, consult Chapter 11 of Batini, Ceri, and Navathe (1992). The DevX Database Zone (www.devx.com) has practical advice about database development and data modeling.

part four

Relational Database Design

The chapters in Part 4 stress practical skills and design processes for relational databases to enable you to implement a conceptual design using a relational DBMS. Chapter 7 covers the motivation for data normalization and provides detailed coverage of functional dependencies, normal forms, and practical considerations to apply data normalization. Chapter 8 contains broad coverage of physical database design including objectives, inputs, and file structure and query optimization background, along with detailed guidelines for important design choices.

chapter

7

Normalization Concepts and Processes

Learning Objectives

This chapter describes normalization, a technique to eliminate unwanted redundancy in a table design. After this chapter, the student should have acquired the following knowledge and skills:

- Identify modification anomalies in tables with excessive redundancies
- Define functional dependencies among columns of a table
- Normalize tables by detecting violations of normal forms and applying normalization rules
- Analyze M-way relationships using the concept of independence
- Appreciate the usefulness and limitations of normalization

OVERVIEW

Chapters 5 and 6 presented tools for data modeling, a fundamental skill for database development. You learned about the notation used in entity relationship diagrams, important data modeling patterns and transformations, guidelines to avoid common modeling errors, and conversion of entity relationship diagrams (ERDs) into table designs. You applied this knowledge to construct ERDs for small, narrative problems. This chapter extends your database design skills by presenting normalization techniques to remove redundancy in a table design.

Redundancies can cause insert, update, and delete operations to produce unexpected side effects known as modification anomalies. This chapter prescribes normalization techniques to remove modification anomalies caused by redundancies. You will learn about functional dependencies, several normal forms, and a procedure to generate tables without redundancies. In addition, you will learn how to analyze M-way relationships for redundancies. This chapter concludes by briefly presenting additional normal forms and discussing the usefulness and limitations of normalization techniques in the database development process.

7.1 OVERVIEW OF RELATIONAL DATABASE DESIGN

After converting an ERD to a table design, your work is not yet finished. You need to analyze the tables for redundancies that can make the tables difficult to use. This section describes negative impacts of redundancies on using tables and presents an important kind of constraint to analyze redundancies.

7.1.1 Avoidance of Modification Anomalies

A good database design ensures that users can change the contents of a database without unexpected side effects. For example in a university database, a user should be able to insert a new course without having to simultaneously insert a new offering of the course and a new student enrolled in the course. Likewise, when a student is deleted from the database due to graduation, course data should not be inadvertently lost. These problems are examples of **modification anomalies**, unexpected side effects that occur when changing the contents of a table with excessive redundancies. A good database design avoids modification anomalies by eliminating excessive redundancies.

Modification Anomaly
an unexpected side effect that occurs when changing data in a table with excessive redundancies.

To understand more precisely the impact of modification anomalies, let us consider a poorly designed database. Imagine that a university database consists of the single table shown in Table 7-1. Such a poor design[1] makes it easy to identify anomalies. The following list describes some of the problems with this design.

- This table has insertion anomalies. An insertion anomaly occurs when extra column values beyond the target values must be added to a table. For example, to insert a course, it is necessary to know a student and an offering because the combination of *StdNo* and *OfferNo* is the primary key. Remember that a row cannot exist with null values for any part of its primary key.
- This table has update anomalies. An update anomaly occurs when it is necessary to change multiple rows to modify only a single fact. For example, if we change the *StdClass* of student S1, two rows must be changed. If S1 was enrolled in 10 classes, 10 rows must be changed.
- This table has deletion anomalies. A deletion anomaly occurs whenever deleting a row inadvertently causes other data to be deleted. For example, if we delete the enrollment of S2 in O3 (third row), we lose the information about offering O3 and course C3.

To deal with these anomalies, users may circumvent them (such as using a default primary key to insert a new course) or database programmers may write code to prevent inadvertent loss of data. A better solution is to modify the table design to remove redundancies that cause anomalies.

7.1.2 Functional Dependencies

Functional dependencies are important tools when analyzing a table for excessive redundancies. A functional dependency is a constraint about columns in a table.

TABLE 7-1
Sample Data for the Big University Database Table

StdNo	StdCity	StdClass	OfferNo	OffTerm	OffYear	EnrGrade	CourseNo	CrsDesc
S1	SEATTLE	JUN	O1	FALL	2017	3.5	C1	DB
S1	SEATTLE	JUN	O2	FALL	2017	3.3	C2	VB
S2	BOTHELL	JUN	O3	SPRING	2018	3.1	C3	OO
S2	BOTHELL	JUN	O2	FALL	2017	3.4	C2	VB

[1] This single-table design is not as extreme as it may seem. Users without proper database training often design a database using a single table.

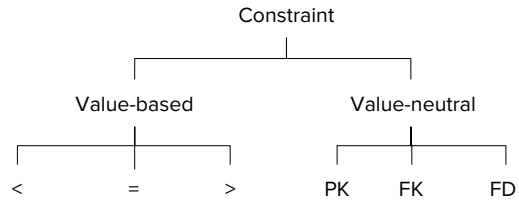


FIGURE 7.1
Classification of Database Constraints

Constraints can be characterized as value-based versus value-neutral (Figure 7.1). A value-based constraint involves a comparison of a column to a constant using a comparison operator such as <, =, or >. For example, age ≥ 21 is an important value-based constraint in a database used to restrict sales of alcohol to minors. A value-neutral constraint involves a comparison of columns. For example, a value-neutral constraint is that retirement age should be greater than current age in a database for retirement planning.

Primary key (PK) and foreign key (FK) constraints are important value-neutral constraints. A primary key can take any value as long as it does not match the primary key value in an existing row. A foreign key constraint requires that the value of a column in one table matches the value of a primary key in another table.

A functional dependency is another important value-neutral constraint. A **functional dependency (FD)** is a constraint about two or more columns of a table. X determines Y ($X \rightarrow Y$) if there exists at most one value of Y for every value of X. The word function comes from mathematics where a function gives one value. For example, student number determines city (*StdNo* → *StdCity*) in the university database table if there is at most one city value for every student number. The columns appearing on the left-hand side of an FD are called the determinant or, alternatively, an LHS for left-hand side. In this example, *StdNo* is a determinant.

Functional Dependency
a constraint about two or more columns of a table. X determines Y ($X \rightarrow Y$) if there exists at most one value of Y for every value of X.

You can also think about functional dependencies as identifying potential candidate keys. By stating that $X \rightarrow Y$, if X and Y are placed together in a table without other columns, X is a candidate key. Every determinant (LHS) is a candidate key if it is placed in a table with the other columns that it determines. For example, if *StdNo*, *StdCity*, and *StdClass* are placed in a table together and *StdNo* → *StdCity* and *StdNo* → *StdClass* then *StdNo* is a candidate key. If there are no other candidate keys, a determinant will become the primary key if it does not allow null values.

Functional Dependency Lists and Diagrams A simple organization of FDs is to list them, grouped by LHS as shown in Table 7-2. As you will see, this arrangement facilitates the normalization process.

As an alternative organization, a functional dependency diagram compactly displays the functional dependencies of a particular table. You should arrange FDs to visually group columns sharing the same determinant. In Figure 7.2, it is easy to spot the dependencies where *StdNo* is the determinant. By examining the position and height of lines, you can see that the combination of *StdNo* and *OfferNo* determines *EnrGrade* whereas *OfferNo* alone determines *OffTerm*, *OffYear*, and *CourseNo*. With a large number of FDs, functional dependency diagrams can be difficult to draw and understand. Thus, FD lists are preferred to FD diagrams even though FD lists can be long for a large collection of FDs.

TABLE 7-2
List of FDs for the Big University Database Table

StdNo → StdCity, StdClass
OfferNo → OffTerm, OffYear, CourseNo
CourseNo → CrsDesc
StdNo, OfferNo → EnrGrade

Identifying Functional Dependencies Besides understanding the functional dependency definition and notation, database designers must be able to identify functional dependencies when collecting database requirements. In problem narratives, some functional dependencies can be identified by statements about uniqueness. For example, a user may state that each course offering has a unique offering number along with the year and term of the offering. From this statement, the

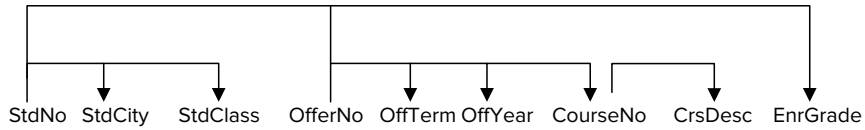


FIGURE 7.2
Dependency Diagram for the Big University Database Table

designer should assert that *OfferNo* → *OffYear* and *OfferNo* → *OffTerm*. You can also identify functional dependencies in a table design resulting from the conversion of an ERD. Functional dependencies would be asserted for each unique column (primary key or other candidate key) with the unique column as the LHS and other columns in the table on the right-hand side (RHS).

FDs for 1-M relationships assert an FD in the child-to-parent direction of a 1-M relationship. Do not assert an FD for the parent-to-child direction because each LHS value can be associated with at most one RHS value.

Although functional dependencies derived from statements about uniqueness are easy to identify, functional dependencies derived from statements about 1-M relationships can be confusing to identify. When you see a statement about a 1-M relationship, the functional dependency is derived from the child-to-parent direction, not the parent-to-child direction. For example, the statement "A faculty teaches many offerings but an offering is taught by one faculty," defines a functional dependency from a unique column of offering to a unique column of faculty such as *OfferNo* → *FacNo*. Novice designers sometimes incorrectly assert that *FacNo* determines a collection of *OfferNo* values. This statement is not correct because a functional dependency must allow at most one associated value, not a collection of values.

Functional dependencies in which the LHS is not a primary or candidate key can also be difficult to identify. These FDs are especially important to identify after converting an ERD to a table design. You should carefully look for FDs in which the LHS is not a candidate key or primary key. You should also consider FDs in tables with a combined primary or candidate key in which the LHS is part of a key, but not the entire key. The presentation of normal forms in Section 7.2 explains that these kinds of FDs can lead to modification anomalies.

Another important consideration in asserting functional dependencies is minimalism of the LHS. It is important to distinguish when one column alone is the determinant versus a combination of columns. An FD in which the LHS contains more than one column usually represents an M-N relationship. For example, the statement "The order quantity is collected for each product purchased in an order," translates to the FD *OrdNo*, *ProdNo* → *OrdQty*. Order quantity depends on the combination of order number and product number, not just one of these columns.

Minimal Determinant the determinant (column(s) appearing on the LHS of a functional dependency) must not contain extra columns. This minimalism requirement is similar to the minimalism requirement for candidate keys.

Part of the confusion about minimalism of the LHS is due to the meaning of columns in the left-hand versus right-hand side of a dependency. To record that student number determines city and class, you can write either *StdNo* → *StdCity*, *StdClass* (more compact) or *StdNo* → *StdCity* and *StdNo* → *StdClass* (less compact). If you assume that the e-mail address is also unique for each student, then you can write *Email* → *StdCity*, *StdClass*. You should not write *StdNo*, *Email* → *StdCity*, *StdClass* because these FDs imply that the combination of *StdNo* and *Email* is the determinant. Thus, you should write FDs so that the LHS does not contain unneeded columns.[2] The prohibition against unneeded columns for determinants is the same as the prohibition against unneeded columns in candidate keys. Both determinants and candidate keys must be minimal.

7.1.3 Falsification of FDs using Sample Data

A functional dependency cannot be proven to exist by examining the rows of a table. However, you can eliminate or falsify a functional dependency (i.e., prove that a

[2] This concept is more properly known as "full functional dependence." Full functional dependence means that the LHS is minimal.

StdNo	StdCity	StdClass	OfferNo	OffTerm	OffYear	EnrGrade	CourseNo	CrsDesc
S1	SEATTLE	JUN	O1	FALL	2017	3.5	C1	DB
S1	SEATTLE	JUN	O2	FALL	2017	3.3	C2	VB
S2	BOTHELL	JUN	O3	SPRING	2018	3.1	C3	OO
S2	BOTHELL	JUN	O2	FALL	2017	3.4	C2	VB
S3	DENVER	SEN	O4	FALL	2016	3.0	C3	OO

TABLE 7-3
Additional Row in the Sample Data for the Big University Database Table

functional dependency does not exist) by examining the rows of a table. For example, in the university database table (Table 7-1) you can conclude that *StdClass* does not determine *StdCity* because there are two rows with the same value for *StdClass* ("JUN") but different values for *StdCity* ("SEATTLE" and "BOTHELL"). Thus, it is sometimes helpful to examine sample rows in a table to eliminate potential functional dependencies. Ultimately, the database designer must make the final decision about the functional dependencies that exist in a table.

Falsifying Potential FDs
using sample data to eliminate potential FDs. If two rows have the same value for the LHS but different values for the RHS, an FD cannot exist. In subtle situations, a database designer can use sample rows with user feedback to determine FDs.

To demonstrate usage of sample data to falsify potential FDs, the following list explains falsification of FDs with *OffTerm* as the LHS using the sample data in Table 7-1. Note that falsification of an FD requires two rows with the same LHS value but different RHS value. For example, two pairs of rows (<1,4> and <2,4>) falsify *OffTerm* → *StdNo*. Although one would not normally consider FDs with *OffTerm* as a LHS, the elimination technique may be useful for plausible LHS columns such as *OfferNo* and *StdNo*.

- *OffTerm* → *StdNo* is falsified by two pairs of rows: <1,4> and <2,4>.
- *OffTerm* → *StdCity* is falsified by two pairs of rows: <1,4> and <2,4>.
- *OffTerm* → *StdClass* is not falsified by any pair of rows.
- *OffTerm* → *OfferNo* is falsified by two pairs of rows: <1,2> and <1,4>.
- *OffTerm* → *OffYear* is not falsified by any pair of rows.
- *OffTerm* → *EnrGrade* is falsified by the three pairs of rows: <1,2>, <1,4>, and <2,4>.
- *OffTerm* → *CourseNo* is falsified by two pairs of rows: <1,2> and <1,4>.
- *OffTerm* → *CrsDesc* is falsified by two pairs of rows: <1,2> and <1,4>.

Since *OffTerm* is not a determinant in any FD, you should add an additional row (row 5) in Table 7-3 to falsify FDs not eliminated by rows in Table 7-1. The following list shows FDs eliminated with the additional row in Table 7-3.

- *OffTerm* → *StdClass* is falsified by three pairs of rows: <1,5>, <2,5>, and <4,5>.
- *OffTerm* → *OffYear* is falsified by three pairs of rows: <1,5>, <2,5>, and <4,5>.

7.2 BASIC NORMAL FORMS

Normalization is the process of removing redundancy in a table so that the table does not have modification anomalies. A number of normal forms have been developed to remove redundancies. A normal form is a rule about allowable dependencies. Each normal form removes certain kinds of redundancies. As shown in Figure 7.3, first normal form (1NF) is the starting point. All tables without repeating groups are in 1NF. 2NF is stronger than 1NF. Only a subset of the 1NF tables is in 2NF. Each successive normal form refines the previous normal form to remove additional kinds of redundancies. Because BCNF (Boyce-Codd Normal Form) is a revised (and stronger) definition for 3NF, 3NF and BCNF are shown in the same part of Figure 7.3.

2NF and 3NF/BCNF are rules about functional dependencies. If the functional dependencies for a table match a specified pattern, the table is in the specified normal

form. 3NF/BCNF is the most important rule in practice because higher normal forms involve other kinds of dependencies that are less common and more difficult to apply. Because BCNF is a revised and simpler definition of 3NF, Section 7.2 presents BCNF without presentation of 2NF and 3NF.[3] You can understand BCNF without details about 2NF and 3NF.

Later sections present details of higher normal forms. Section 7.3 presents 4NF as a way to reason about M-way relationships. Section 7.4 presents 5NF and DKNF (domain key normal form) to show that higher normal forms have been proposed. DKNF is the ultimate normal form, but it remains an ideal rather than a practical normal form. Thus, your study should emphasize BCNF with some background on higher level normal forms.

7.2.1 First Normal Form

1NF prohibits nesting or repeating groups in tables. A table not in 1NF is unnormalized or nonnormalized. In Table 7-4, the university table is unnormalized because the two rows contain repeating groups or nested tables. To convert an unnormalized table into 1NF, you replace each value of a repeating group with a row. In a new row, you copy the nonrepeating columns. You can see the conversion by comparing Table 7-4 with Table 7-1 (two rows with repeating groups versus four rows without repeating groups).

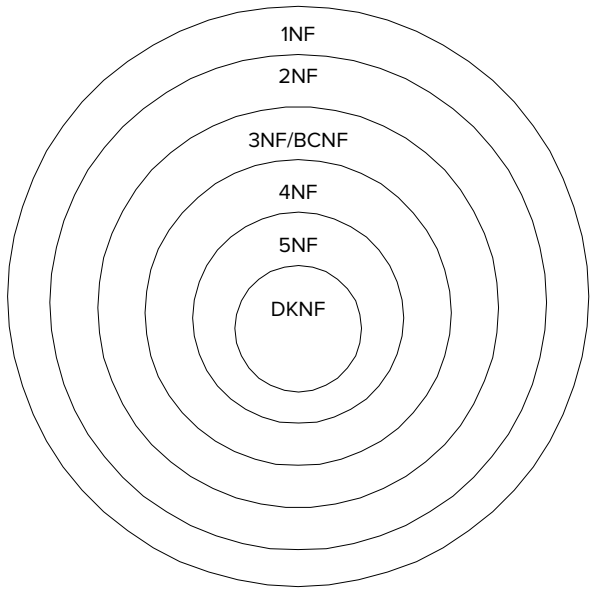
Because most commercial DBMSs require 1NF tables,[4] you normally do not need to convert tables into 1NF. However, you often need to perform the reverse process (1NF tables to unnormalized tables) for report generation and document representation. As discussed in Chapter 10, reports use nesting to show relationships. As discussed in Chapter 19, the SQL standard supports nested tables, but nested tables remain a niche practice for relational databases.

7.2.2 Boyce-Codd Normal Form

BCNF Definition
a table is in BCNF if every determinant is a candidate key.

The revised 3NF definition, known as Boyce-Codd normal form (BCNF), is a better definition because it is simpler and covers two special cases omitted by the original 3NF definition. The **BCNF definition** is simpler because it does not refer to 2NF. The special cases are uncommon so they are not covered in this chapter.

FIGURE 7.3
Relationship of Normal Forms



[3] Appendix 7.A presents 2NF and 3NF for additional background about normalization.

TABLE7-4
Unnormalized University Database Table

StdNo	StdCity	StdClass	OfferNo	OffTerm	OffYear	EnrGrade	CourseNo	CrsDesc
S1	SEATTLE	JUN	O1	FALL	2017	3.5	C1	DB
			O2	FALL	2017	3.3	C2	VB
S2	BOTHELL	JUN	O3	SPRING	2018	3.1	C3	OO
			O2	FALL	2017	3.4	C2	VB

BCNF involves two concepts, determinant and candidate key. Recall that a determinant is a LHS in an FD. A candidate key has the uniqueness property in a table. No two rows have the same value for a candidate key except when a candidate key allows null values. BCNF requires all determinants to be candidate keys.

Violations of BCNF involve FDs in which the determinant (LHS) is not a candidate key. In a poor table design such as the big university database table (sample data in Table 7-1 and FD list in Table 7-2), you can easily detect violations of BCNF. For ease of reference, the following list repeats the FDs in Table 7-2. The combination of (*StdNo*, *OfferNo*) is the only candidate key. The determinants are *StdNo*, *OfferNo*, *CourseNo*, and the combination of (*StdNo*, *OfferNo*). As a violation of BCNF, *StdNo* is a determinant but not a candidate key (it is part of a candidate key but not a candidate key by itself). The only FD that satisfies BCNF is *StdNo*, *OfferNo* → *EnrGrade*.

- *StdNo* → *StdCity*, *StdClass*
- *OfferNo* → *OffTerm*, *OffYear*, *CourseNo*
- *CourseNo* → *CrsDesc*
- *StdNo*, *OfferNo* → *EnrGrade*

For another example, let us apply the BCNF definition to the big patient table with sample rows in Table 7-5 and FDs in Table 7-6. The combination of (*VisitNo*, *ProvNo*) is the only candidate key. All of the FDs in Table 7-6 violate the BCNF definition except the last FD (*VisitNo*, *ProvNo* → *Diagnosis*). All of the other FDs have determinants that are not candidate keys (part of a candidate key in some cases but not an entire candidate key).

To resolve BCNF violations, you should split the big patient table into smaller tables. Each determinant should be placed into a separate table along with the columns that it determines. The result contains five tables, one table for each group of FDs with the same determinant.

PatDBTable1 (PatNo, PatAge, PatZip)
FOREIGN KEY (PatZip) REFERENCES PatientTable2
PatDBTable2 (PatZip, PatCity,)
PatDBTable3 (ProvNo, ProvSpecialty)
PatDBTable4 (VisitNo, VisitDate, PatNo)

TABLE 7-5
Sample Data for the Big Patient Table

VisitNo	VisitDate	PatNo	PatAge	PatCity	PatZip	ProvNo	ProvSpecialty	Diagnosis
V10020	1/13/2018	P1	35	DENVER	80217	D1	INTERNIST	EAR INFECTION
V10020	1/13/2018	P1	35	DENVER	80217	D2	NURSE PRACTITIONER	INFLUENZA
V93030	1/20/2018	P3	17	ENGLEWOOD	80113	D2	NURSE PRACTITIONER	PREGNANCY
V82110	1/18/2018	P2	60	BOULDER	85932	D3	CARDIOLOGIST	MURMUR

[4] Although nested tables have been supported since the SQL:1999 standard with commercial support in Oracle, this feature does not appear important in most business applications. Thus, this chapter does not consider the complications of nested tables on normalization.

FOREIGN KEY (PatNo) REFERENCES PatientTable1
PatDBTable5 (VisitNo, ProvNo, Diagnosis)
FOREIGN KEY (VisitNo) REFERENCES PatientTable4
FOREIGN KEY (ProvNo) REFERENCES PatientTable3

TABLE 7-6
List of FDs for the Big Patient Table

PatNo → PatAge, PatZip
PatZip → PatCity
ProvNo → ProvSpecialty
VisitNo → PatNo, VisitDate
VisitNo, ProvNo → Diagnosis

BCNF for Tables with Multiple, Composite Candidate Keys Tables with multiple composite candidate keys can be subtle to analyze especially if another column is also a determinant. *UnivTable2* (Figure 7.4) has two candidate keys: the combination of *StdNo* and *OfferNo* (the primary key) and the combination of *StdEmail* and *OfferNo*. In the FDs for *UnivTable2* (Figure 7.4), you should note that *StdNo* and *StdEmail* determine each other. Because of the FDs between *StdNo* and *StdEmail*, *UnivTable2* contains a redundancy as *Email* is repeated for each *StdNo*. For example, the first two rows contain the same e-mail address because the *StdNo* value is the same. The following points explain why *UnivTable2* is not in BCNF.

- The dependencies between *StdNo* and *StdEmail* violate BCNF. Both *StdNo* and *StdEmail* are determinants, but neither is an entire candidate key although each column is part of a candidate key.
- To eliminate the redundancy, you should split *UnivTable2* into two tables, as shown in Figure 7.4. The UNIQUE constraint supports the FD *StdNo* → *StdEmail*.

UnivTable3 (Figure 7.5) depicts another example of a table with multiple, composite candidate keys. *UnivTable3* has two candidate keys: the combination of *StdNo* and *AdvisorNo* (the primary key) and the combination of *StdNo* and *Major*. *UnivTable3* has a redundancy as *Major* is repeated for each row with the same *AdvisorNo* value. The following points explain why *UnivTable3* is not in BCNF.

- The dependency diagram (Figure 7.5) shows that *AdvisorNo* is a determinant but not a candidate key by itself. Thus, *UnivTable3* is not in BCNF.
- To eliminate the redundancy, you should split *UnivTable3* into two tables as shown in Figure 7.5.

These examples demonstrate two points about normalization. First, tables with multiple, composite candidate keys are difficult to analyze. You need to study the dependencies carefully in each example to understand the conclusions about BCNF

FIGURE 7.4
Sample Rows, FDs, and Normalized Tables for *UnivTable2*

UnivTable2

StdNo	*OfferNo*	*StdEmail*	*EnrGrade*
S1	O1	joe@bigu	3.5
S1	O2	joe@bigu	3.6
S2	O1	mary@bigu	3.8
S2	O3	mary@bigu	3.5

StdNo, OfferNo → EnrGrade
OfferNo, StdEmail → EnrGrade
StdNo → StdEmail
StdEmail → StdNo

UnivTable2-1 (OfferNo, StdNo, EnrGrade)
FOREIGN KEY (StdNo) REFERENCES UnivTable2-2
UnivTable2-2 (StdNo, StdEmail)
UNIQUE (StdEmail)

violations. Second, tables with multiple, composite candidate keys are not common. The examples in Figures 7.4 and 7.5 were purposely constructed to depict subtleties of multiple, composite candidate keys.

7.2.3 Simple Synthesis Procedure

Although BCNF has a simple definition, applying the definition is not always as easy as shown in the previous section. To apply BCNF, the list of FDs must be carefully analyzed. An FD list with derived FDs can lead to a poor table design.

The simple synthesis procedure can be used to generate tables satisfying BCNF starting with a list of FDs. The word synthesis means that the individual functional dependencies are combined to construct tables. This usage is similar to other disciplines such as music where synthesis involves combining individual sounds to construct larger units such as melodies, scores, and so on.

Figure 7.6 depicts the steps of the simple synthesis procedure. The first two steps eliminate redundancy by removing extraneous columns and derived FDs. The last three steps produce tables for collections of FDs. The tables produced in the last three steps may not be correct if redundant FDs are not eliminated in the first two steps.

Applying the Simple Synthesis Procedure To understand this procedure, you can apply it to the FDs of the big university database table shown in Table 7-7. Two FDs have been added to the FD list from Table 7-2 to depict the steps of the simple synthesis procedure. In the first step, the last FD group contains an extraneous column (*OfferNo*) in the LHS because *StdNo* → *StdCity* without *OfferNo* in the LHS. Since the

UnivTable3

StdNo	*AdvisorNo*	*Major*	*Status*
S1	A1	IS	COMPLETED
S1	A2	FIN	PENDING
S2	A1	IS	PENDING
S2	A3	FIN	COMPLETED

FIGURE 7.5
Sample Rows, FDs, and Normalized Tables for *UnivTable3*

StdNo, AdvisorNo → Status
StdNo, Major → Status
AdvisorNo → Major

UnivTable3-1 (AdvisorNo, StdNo, Status)
FOREIGN KEY (AdvisorNo) REFERENCES UnivTable3-2
UnivTable3-2 (AdvisorNo, Major)

1. Eliminate extraneous columns from the LHS of FDs.
2. Remove derived FDs from the FD list.
3. Arrange the FDs into groups with each group having the same determinant.
4. For each FD group, make a table with the determinant as the primary key. Add referential integrity constraints to connect the tables.
5. Merge tables in which one table contains all columns of the other table.
   5.1. Choose the primary key of one of the separate tables as the primary of the new, merged table.
   5.2. Define a unique constraint for each former primary key that was not designated as the primary key of the new table.

FIGURE 7.6
Steps of the Simple Synthesis Procedure

first group of FDs contains *StdNo* → *StdCity*, the revised list of FDs (Table 7-8) removes *StdCity* in the RHS of the last FD group.

To apply the second step, you need to know mathematical laws that derive FDs from other FDs. Although there are a number of laws to derive FDs,[5] the most prominent law is the law of transitivity. A **transitive dependency** is a functional dependency derived by the law of transitivity. The law of transitivity indicates that if an object A is related to B and B is related to C, then you can conclude that A is related to C. For example, the < operator obeys the transitive law for real numbers: A < B and B < C implies that A < C. Functional dependencies, like the < operator, obey the law of transitivity: A → B, B → C, then A → C. For example, *OrdNo* → *CustBal* is a transitive dependency derived from *OrdrNo* → *CustNo* and *CustNo* → *CustBal*.

Transitive Dependency
an FD derived by the law of transitivity. Transitive FDs should not be recorded as input to the normalization process.

In this chapter, the simple synthesis procedure eliminates only transitively derived FDs in step 2. For details about the other laws to derive FDs, you should consult references listed at the end of the chapter.

In the second step, the FD *OfferNo* → *CrsDesc* is a transitive dependency because *OfferNo* → *CourseNo* and *CourseNo* → *CrsDesc* implies *OfferNo* → *CrsDesc*. Therefore, you should delete this dependency (*OfferNo* → *CrsDesc*) from the list of FDs.

In the third step, you group FDs by determinant. From Table 7-8, you can make the following FD groups. Note that the FD *OfferNo* → *CrsDesc* has been removed from this list as a result of step 2.

- *StdNo* → *StdCity*, *StdClass*
- *OfferNo* → *OffTerm*, *OffYear*, *CourseNo*
- *CourseNo* → *CrsDesc*
- *StdNo*, *OfferNo* → *EnrGrade*

In the fourth step, you replace each FD group with a table having the determinant in a group as the primary key. Thus, you have four resulting BCNF tables as shown below. You should add table names for completeness.

Student(StdNo, StdCity, StdClass)
Offering(OfferNo, OffTerm, OffYear, CourseNo)
Course(CourseNo, CrsDesc)
Enrollment(StdNo, OfferNo, EnrGrade)

After defining the tables, you should add referential integrity constraints to connect the tables. To detect the need for a referential integrity constraint, you should look for a primary key in one table appearing in other tables. For example, *CourseNo* is the primary key of *Course* but it also appears in *Offering*. Therefore, you should define

TABLE 7-7
List of FDs for the Big University Database Table

StdNo → StdCity, StdClass
OfferNo → OffTerm, OffYear, CourseNo, CrsDesc
CourseNo → CrsDesc
StdNo, OfferNo → EnrGrade, StdCity

TABLE 7-8
Revised List of FDs after Step 1

StdNo → StdCity, StdClass
OfferNo → OffTerm, OffYear, CourseNo, CrsDesc
CourseNo → CrsDesc
StdNo, OfferNo → EnrGrade

[5] The laws to derive FDs are known as Armstrong's Axioms, published in a 1974 paper by William Armstrong.

a referential integrity constraint indicating that *Offering.CourseNo* refers to *Course.CourseNo*. The tables are repeated below with the addition of referential integrity constraints.

Student(StdNo, StdCity, StdClass)
Offering(OfferNo, OffTerm, OffYear, CourseNo)
    FOREIGN KEY (CourseNo) REFERENCES Course
Course(CourseNo, CrsDesc)
Enrollment(StdNo, OfferNo, EnrGrade)
    FOREIGN KEY (StdNo) REFERENCES Student
    FOREIGN KEY (OfferNo) REFERENCES Offering

The fifth step is not necessary because the FDs for this problem are simple. When there are multiple candidate keys for a table, the fifth step is necessary. For example, if *StdEmail* is added as a column, then the FDs *StdEmail* → *StdNo* and *StdNo* → *StdEmail* should be added to the list. Note that the FDs *StdEmail* → *StdCity*, *StdClass* should not be added to the list because these FDs can be transitively derived from other FDs. As a result of step 3, another group of FDs is added. In step 4, a new table (*Student2*) is added with *StdEmail* as the primary key. Because the *Student* table contains the columns of the *Student2* table, the tables (*Student* and *Student2*) are merged in step 5. One of the candidate keys (*StdNo* or *StdEmail*) is chosen as the primary key. Since *StdNo* is chosen as the primary key, a unique constraint is defined for *StdEmail*.

StdEmail → *StdNo*
StdNo → *StdEmail*
Student2(StdEmail, StdNo)
    UNIQUE(StdNo)

As this additional example demonstrates, multiple candidate keys do not violate BCNF. The fifth step of the simple synthesis procedure creates tables with multiple candidate keys because it merges tables. Multiple candidate keys do not violate 3NF either. There is no reason to split a table just because it has multiple candidate keys. Splitting a table with **multiple candidate keys** can slow query performance due to extra joins.

Multiple Candidate Keys a common misconception by novice database designers is that a table with multiple candidate keys violates BCNF. Multiple candidate keys do not violate BCNF. Thus, you should not split a table just because it has multiple candidate keys.

You can use the simple synthesis procedure to analyze simple dependency structures. Most tables resulting from a conversion of an ERD should have simple dependency structures because the data modeling process has already done much of the normalization process. Most tables should be nearly normalized after the conversion process.

To make the synthesis procedure easy to use, some of the details have been omitted. In particular, step 2 can be rather involved because there are more ways to derive dependencies than transitivity. Even checking for transitivity can be difficult with many columns. The full details of step 2 can be found in references cited at the end of the chapter. Even if you understand the complex details, step 2 cannot be done manually for complex dependency structures. Fortunately, complex dependency structures seem rare in practice. Commercial software for normalization does not exist because of the lack of demand due to mostly simple dependency structures encountered in practice.

Another Example Using the Simple Synthesis Procedure To gain more experience with the Simple Synthesis Procedure, you should understand another example. This example describes a database to track reviews of papers submitted to an academic conference. Prospective authors submit papers for review and possible acceptance in the published conference proceedings. Here are more details about authors, papers, reviews, and reviewers:

- Author information includes the unique author number, author name, mailing address, and the unique but optional electronic address.

- Paper information includes the primary author, the unique paper number, the title, the abstract, and the review status (pending, accepted, rejected).
- Reviewer information includes the unique reviewer number, the name, the mailing address, and the unique but optional electronic address.
- A completed review includes the reviewer number, the date, the paper number, comments to the authors, comments to the program chairperson, and ratings (overall, originality, correctness, style, and relevance). The combination of reviewer number and paper number identifies a review.

Before beginning the procedure, you must identify the FDs in the problem. The following is a list of FDs for the problem:

AuthNo → *AuthName*, *AuthEmail*, *AuthAddress*
AuthEmail → *AuthNo*
PaperNo → *Primary-AuthNo*, *Title*, *Abstract*, *Status*
RevNo → *RevName*, *RevEmail*, *RevAddress*
RevEmail → *RevNo*
RevNo, *PaperNo* → *Auth-Comm*, *Prog-Comm*, *Date*, *Rating1*, *Rating2*, *Rating3*, *Rating4*, *Rating5*

Because the LHS is minimal in each FD, the first step is finished. The second step is not necessary because there are no transitive dependencies. Note that the FDs *AuthEmail* → *AuthName*, *AuthAddress*, and *RevEmail* → *RevName*, *RevAddress* can be transitively derived. If any of these FDs was part of the original list, they should be removed. For each of the six FD groups, you should define a table. In the last step, you combine the FD groups with *AuthNo* and *AuthEmail* and *RevNo* and *RevEmail* as determinants. In addition, you should add unique constraints for *AuthEmail* and *RevEmail* because these columns were not selected as the primary keys of the new tables.

Author(AuthNo, AuthName, AuthEmail, AuthAddress)
    UNIQUE (AuthEmail)
Paper(PaperNo, Primary-AuthNo, Title, Abstract, Status)
    FOREIGN KEY (Primary-AuthNo) REFERENCES Author
Reviewer(RevNo, RevName, RevEmail, RevAddress)
    UNIQUE (RevEmail)
Review(PaperNo, RevNo, Auth-Comm, Prog-Comm, Date, Rating1, Rating2, Rating3, Rating4, Rating5)
    FOREIGN KEY (PaperNo) REFERENCES Paper
    FOREIGN KEY (RevNo) REFERENCES Reviewer

7.3 REFINING M-WAY RELATIONSHIPS

Beyond BCNF, a remaining concern is the analysis of M-way relationships. Recall that M-way relationships are represented by associative entity types in the Crow's Foot ERD notation. In the conversion process, an associative entity type converts into a table with a combined primary key consisting of three or more components. The concept of relationship independence, underlying 4NF, is an important tool used to analyze M-way relationships. Using the concept of relationship independence, you may find that an M-way relationship should be split into two or more binary relationships to avoid redundancy. The following sections describe the concept of relationship independence and 4NF.

7.3.1 Relationship Independence

Before you study relationship independence in database design, let us discuss the meaning of independence in statistics. Two variables are statistically independent if knowing something about one variable tells you nothing about another variable. More

precisely, two variables are independent if the probability of both variables (the joint probability) can be derived from the probability of each variable alone. For example, one variable may be the age of a rock and another variable may be the age of the person holding the rock. Because the age of a rock, and the age of a person holding the rock are unrelated, these variables are considered independent. However, the age of a person and a person's marital status are related. The value of a person's age influences the probability of being single, married, or divorced. If two variables are independent, it is redundant to store data about how they are related. You can use probabilities about individual variables to derive joint probabilities.

The concept of **relationship independence** is similar to statistical independence. If two relationships are independent (that is, not related), it is redundant to store data about a third relationship. You can derive the third relationship by combining the two essential relationships through a join operation. If you store a derived relationship, modification anomalies can result. Thus, the essential idea of relationship independence is not to store relationships that can be derived by joining other (independent) relationships.

Relationship Independence a relationship that can be derived from two independent relationships.

Relationship Independence Example To clarify relationship independence, consider the associative entity type *Enroll* (Figure 7.7) representing a three-way relationship among students, offerings, and textbooks. The *Enroll* entity type converts to the *Enroll* table (Table 7-9) that consists only of a combined primary key: *StdNo*, *OfferNo*, and *TextNo*.

The design question is whether the *Enroll* table has redundancies. If there is redundancy, modification anomalies may result. The *Enroll* table is in BCNF, so there are no anomalies due to functional dependencies. However, the concept of independence leads to the discovery of redundancies. The *Enroll* table can be divided into three combinations of columns representing three binary relationships: *StdNo-OfferNo* representing the relationship between students and offerings, *OfferNo-TextNo* representing the relationship between offerings and textbooks, and *StdNo-TextNo* representing the relationship between students and textbooks. If any of the binary relationships can be derived from the other two, there is a redundancy.

- The relationship between students and offerings (*StdNo-OfferNo*) cannot be derived from the other two relationships. For example, suppose that textbook T1 is used in two offerings, O1 and O2 and by two students, S1 and S2. Knowledge

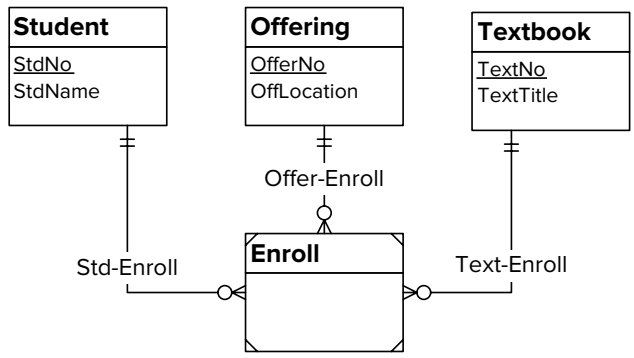


FIGURE 7.7
M-way Relationship Example

TABLE 7-9
Sample Rows of the *Enroll* Table

StdNo	OfferNo	TextNo
S1	O1	T1
S1	O2	T1
S1	O1	T2
S1	O2	T3

about these two facts does not determine the relationship between students and offerings. For example, S1 could be enrolled in O1 or perhaps O2.

- Likewise, the relationship between offerings and textbooks (*OfferNo-TextNo*) cannot be derived. A professor's choice for a collection of textbooks cannot be derived by knowing who enrolls in an offering and what textbooks a student uses.
- However, the relationship between students and textbooks (*StdNo-TextNo*) can be derived by the other two relationships. For example, if student S1 is enrolled in offering O1 and offering O1 uses textbook T1, then you can conclude that student S1 uses textbook T1 in offering O1. Because the *Student-Offering* and the *Offering-Textbook* relationships are independent, you know the textbooks used by a student without storing the relationship instances.

TABLE 7-10
Sample Rows of the Binary *Enroll* Table

StdNo	OfferNo
S1	O1
S1	O2

TABLE 7-11
Sample Rows of the Binary *Orders* Table

OfferNo	TextNo
O1	T1
O1	T2
O2	T1
O2	T3

Because of this independence, the *Enroll* table and the related associative entity type *Enroll* have redundancy. To remove the redundancy, replace the *Enroll* entity type with two binary relationships (Figure 7.8). Each binary relationship converts to a table as shown in Tables 7-10 and 7-11. The *Enroll* and *Orders* tables have no redundancies. For example, to delete a student's enrollment in an offering (say S1 in O1), only one row must be deleted from Table 7-10. In contrast, two rows must be deleted from Table 7-9.

If the assumptions change slightly, an argument can be made for an associative entity type representing a three-way relationship. Suppose that the bookstore wants to record textbook purchases by offering and student to estimate textbook demand. Then the relationship between students and textbooks is no longer independent of the other two relationships. Even though a student is enrolled in an offering and the offering uses a textbook, the student may not purchase the textbook (perhaps borrow it) for the offering. In this situation, there is no independence and a three-way relationship is needed. In addition to the M-N relationships in Figure 7.8, there should be a new associative entity type and three 1-M relationships, as shown in Figure 7.9. You need the *Enroll* relationship to record student selections of offerings and the *Orders* relationship to record professor selections of textbooks. The *Purchase* entity type records purchases of textbooks by students in a course offering. However, a purchase cannot be known from the other relationships.

FIGURE 7.8
Decomposed Relationships Example

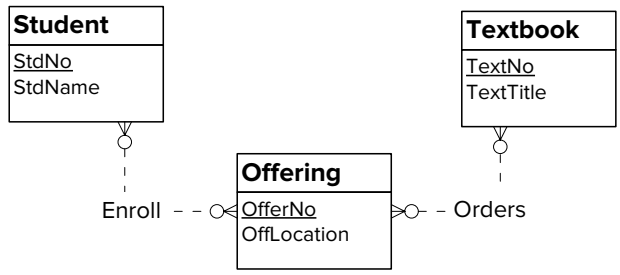
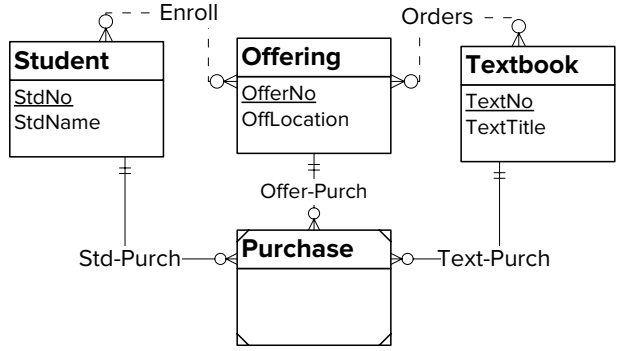


FIGURE 7.9
M-Way and Binary Relationships Example



7.3.2 Multivalued Dependencies and Fourth Normal Form

In relational database terminology, a relationship that can be derived from other relationships is known as a multivalued dependency (MVD). An MVD involves three columns as described in the following definition. Like in the discussion of relationship independence, the three columns comprise a combined primary key of an associative table. The nonessential or derived relationship involves the columns *B* and *C*. The definition states that the nonessential relationship (involving the columns *B* and *C*) can be derived from the relationships *A*-*B* and *A*-*C*. The word multivalued means that *A* can be associated with a collection of *B* and *C* values, not just single values as in a functional dependency.

MVD Definition: The multivalued dependency (MVD) $A \rightarrow\rightarrow B \mid C$ (read *A* multidetermines *B* or *C*) means that

- A given *A* value is associated with a collection of *B* and *C* values, and
- *B* and *C* are independent given the relationships between *A* and *B* and *A* and *C*.

MVDs can lead to redundancies because of independence among columns. You can see the redundancy by using a table to depict an MVD as shown in Figure 7.10. If the two rows above the line exist and the MVD $A \rightarrow\rightarrow B \mid C$ is true, then the two rows below the line will exist. The two rows below the line will exist because the relationship between *B* and *C* can be derived from the relationships *A*-*B* and *A*-*C*. In Figure 7.10, value A1 is associated with two *B* values (B1 and B2) and two *C* values (C1 and C2). Because of independence, value A1 will be associated with every combination of its related *B* and *C* values. The two rows below the line are redundant because they can be derived.

FIGURE 7.10
Table Representation of an MVD

A	B	C
A1	B1	C1
A1	B2	C2
A1	B2	C1
A1	B1	C2

To apply this concept to the *Enroll* table, consider the possible MVD *OfferNo* $\rightarrow\rightarrow$ *StdNo* | *TextNo*. In the first two rows of Figure 7.11, offering O1 is associated with students S1 and S2 and textbooks T1 and T2. If the MVD is true, then the two rows below the line will exist. The last two rows do not need to be stored if you know the first two rows and the MVD exists.

FIGURE 7.11
Representation of the MVD in the *Enroll* Table

OfferNo	StdNo	TextNo
O1	S1	T1
O1	S2	T2
O1	S2	T1
O1	S1	T2

MVDs are generalizations of functional dependencies (FDs). Every FD is an MVD but not every MVD is an FD. An MVD in which a value of *A* is associated with only one value of *B* and one value of *C* is also an FD. In this section, we are interested only in MVDs that are not also FDs. An MVD that is not an FD is known as a nontrivial MVD.

Fourth Normal Form (4NF) Fourth normal form (4NF) prohibits redundancies caused by multivalued dependencies. As an example, the table *Enroll*(*StdNo, OfferNo, TextNo*) (Table 7-8) is not in **4NF** if the MVD *OfferNo* $\rightarrow\rightarrow$ *StdNo* | *TextNo* exists. To eliminate the MVD, split the M-way table *Enroll* into the binary tables *Enroll* (Table 7-9) and *Orders* (Table 7-10).

4NF Definition
a table is in 4NF if it does not contain any nontrivial MVDs (MVDs that are not also FDs).

The ideas of MVDs and 4NF are somewhat difficult to understand. The ideas are somewhat easier to understand if you think of an MVD as a relationship that can be derived by other relationships because of independence.

7.4 HIGHER LEVEL NORMAL FORMS

The normalization story does not end with 4NF. Other normal forms have been proposed, but their practicality has not been demonstrated. This section briefly describes two higher normal forms to complete your normalization background.

7.4.1 Fifth Normal Form

Fifth normal form (5NF) applies to M-way relationships like 4NF. Unlike 4NF, 5NF involves situations in which a three-way relationship should be replaced with three

binary relationships, not two binary relationships as for 4NF. Because situations in which 5NF applies (as opposed to 4NF) are rare, 5NF is generally not considered a practical normal form. Understanding the details of 5NF requires a lot of intellectual investment, but the return on your study time is rarely applicable.

The example in Figure 7.12 demonstrates a situation in which 5NF could apply. The *Authorization* entity type represents authorized combinations of employees, workstations, and software. This associative entity type has redundancy because it can be divided into three binary relationships as shown in Figure 7.13. If you know employees authorized to use workstations, software licensed for workstations, and employees trained to use software, then you know the valid combinations of employees, workstations, and software. Thus, it is necessary to record the three binary combinations (employee-workstation, software-workstation, and employee-software), not the three-way combination of employee, workstation, and software.

Whether the situation depicted in Figure 7.13 is realistic is debatable. For example, if software is licensed for servers rather than workstations, the *Software-Auth* relationship may not be necessary. Even though it is possible to depict situations in which 5NF applies, these situations may not exist in practice.

7.4.2 Domain Key Normal Form

After reading about so many normal forms, you may be asking questions such as "Where does it stop?" and "Is there an ultimate normal form?" Fortunately, the answer to the last question is yes. In a 1981 paper, Dr. Ronald Fagin proposed domain key normal form (DKNF) as the ultimate normal form. In DKNF, domain refers to a data type: a set of values with allowable operations. A set of values is defined by the kind of values (e.g., whole numbers versus floating-point numbers) and the integrity rules about the values (e.g., values greater than 21). Key refers to the uniqueness property of candidate keys. A table is in DKNF if every constraint on a table can be derived from keys and domains. A table in DKNF cannot have modification anomalies.

FIGURE 7.12
Associative Entity Type

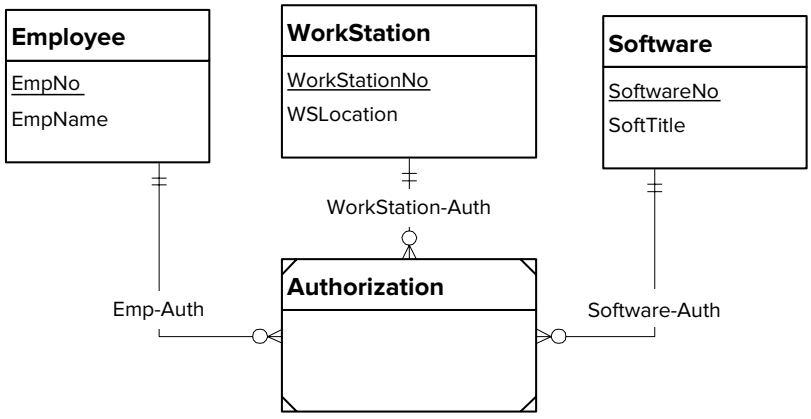
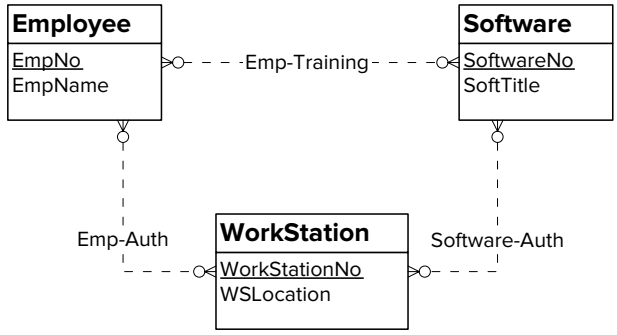


FIGURE 7.13
Replacement of Associative Entity Type with Three Binary Relationships



Unfortunately, DKNF remains an ideal rather than a practical normal form. There is no known procedure that converts a table into DKNF. In addition, it is not even known what tables can be converted to DKNF. As an ideal, you should try to define tables in which most constraints result from keys and domains. These kinds of constraints are easy to test and understand.

7.5 PRACTICAL CONCERNS ABOUT NORMALIZATION

After reading this far, you should be well acquainted with the tools of relational database design. Before you are ready to use these tools, some practical advice is useful. This section discusses the role of normalization in the database development process and the importance of thinking carefully about the objective of eliminating modification anomalis.

7.5.1 Role of Normalization in the Database Development Process

Normalization can be used as either a refinement tool or initial design tool in the database development process. In the refinement approach, you perform conceptual data modeling using the Entity Relationship Model and transform the ERD into tables using the conversion rules. Then, you apply normalization techniques to analyze each table: identify FDs, use the simple synthesis procedure to remove redundancies, and analyze a table for independence if the table represents an M-way relationship. Since the primary key determines the other columns in a table, you only need identify FDs in which the primary key is not the LHS.

In the initial design approach, you use normalization techniques in conceptual data modeling. Instead of drawing an ERD, you identify functional dependencies and apply a normalization procedure like the simple synthesis procedure. After defining the tables, you identify the referential integrity constraints and construct a relational model diagram such as that available in Microsoft Access. If needed, an ERD can be generated from the relational database diagram.

This book clearly favors using normalization as a refinement tool, not as an initial design tool. Through development of an ERD, you intuitively group related attributes. Much normalization is accomplished in an informal manner without the tedious process of recording functional dependencies. As a refinement tool, there are fewer FDs to specify and less normalization to perform. Applying normalization ensures that candidate keys and redundancies have not been overlooked.

Another reason for favoring the refinement approach is that relationships can be overlooked when using normalization as the initial design approach. 1-M relationships must be identified in the child-to-parent direction. For novice data modelers, identifying relationships is easier when considering both sides of a relationship. For an M-N relationship without attributes, there will not be any functional dependencies that show the need for a table. For example, in a design about textbooks and course offerings, if the relationship between them has no attributes, there are no functional

Advantages of Normalization as a Refinement Tool: use normalization to remove redundancies after conversion from an ERD to a table design rather than as an initial design tool.

- Easier to translate requirements into an ERD than into lists of FDs.
- Fewer FDs to specify because most FDs are derived from primary keys.
- Fewer tables to split because normalization performed intuitively during ERD development.
- Easier to identify relationships especially M-N relationships without attributes.

dependencies that relate textbooks and coure offerings.[6] In drawing an ERD, however, the need for an M-N relationship becomes clear.

Refinement Example To demonstrate the refinement process, this subsection uses the university database ERD in Figure 7.14. To make the normalization process more interesting, the university database ERD from Chapter 5 has been embellished with the *DeptNo* and *DeptName* attributes in the *Faculty* entity type and the *StdEmail* attribute in the *Student* entity type.

The conversion process generates five tables as shown in the following table design. The conversion process involves five applications of the entity type rule to add tables, five applications of the 1-M relationship rule to add foreign keys, and two applications of the identifying relationship rule to add primary key components for the *Enrollment* table. In the table design, primary keys are underlined and foreign keys are italicized.

Student(StdNo, StdFirstName, StdLastName, StdCity, StdState, StdZip, StdClass, StdMajor, StdGPA, StdEmail)
Faculty(FacNo, FacFirstName, FacLastName, FacSalary, FacHireDate, FacRank, *FacSupNo*, DeptNo, DeptName)
FOREIGN KEY (FacSupNo) REFERENCES Faculty
Offering(OfferNo, OffTerm, OffYear, *CourseNo*, *FacNo*)
FOREIGN KEY (CourseNo) REFERENCES Course
FOREIGN KEY (FacNo) REFERENCES Faculty
Course(CourseNo, CrsUnits, CrsDesc)
Enrollment(*StdNo, OfferNo*, EnrGrade)
FOREIGN KEY (StdNo) REFERENCES Student
FOREIGN KEY (OfferNo) REFERENCES Offering

After converting an ERD to a table design, you should record FDs for each table and analyze FDs for compliance with BCNF. For each table, the primary key determines other columns. You do not need to explicitly record these FDs as they directly follow from the primary keys. With a good ERD design, you should have relatively few FDs in which the LHS is not a primary key. You should focus on two types of FDs in which the LHS is not the primary key of a table.

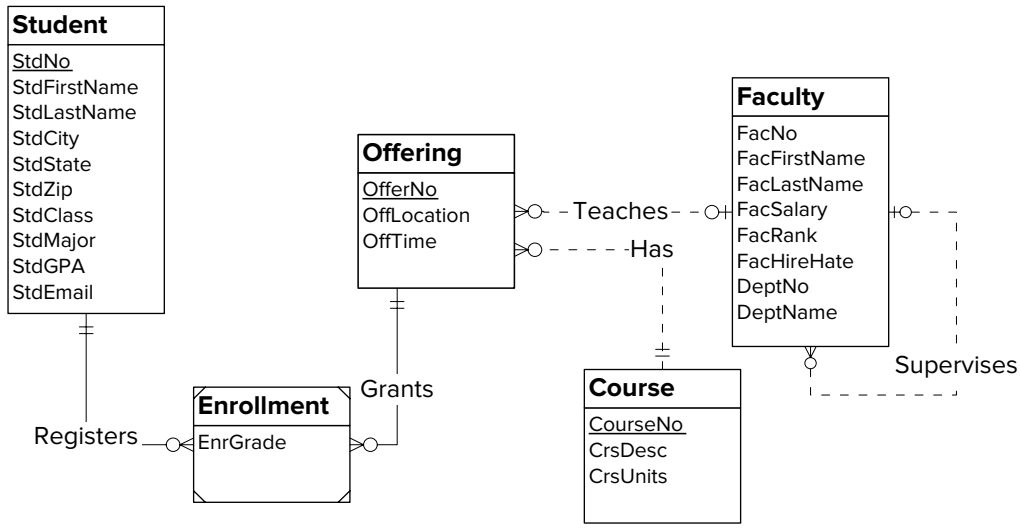


FIGURE 7.14
ERD for the University Database

[6] An FD can be written with a null right-hand side to represent M-N relationships. The FD for the offering-textbook relationship can be expressed as *TextId, OfferNo* → ∅. However, this kind of FD is awkward to state. It is much easier to define an M-N relationship.

- Candidate Key FD: If a column determines the primary key of a table, this column is a candidate key in the table. A candidate key FD does not violate BCNF. It involves a UNIQUE constraint.
- Nonkey FD: If a nonkey column determines another nonkey column, the table violates BCNF.

The table design for the university database contains FDs of each type. In the *Student* table, *StdEmail* → *StdNo* based on assignment of a unique email address for each student. In the revised table design, a UNIQUE constraint is added for *StdEmail*. In the *Faculty* table, *DeptNo* → *DeptName* violates BCNF as *DeptNo* is not a candidate key. To resolve the BCNF violation, you should decompose the *Faculty* table by adding a *Department* table with *DeptNo* as the primary key. If *DeptName* is unique in *Department*, you should record another FD (*DeptName* → *DeptNo*) and add a UNIQUE constraint. The revised table design shows a foreign key constraint for *DeptNo*, a new *Department* table, and a UNIQUE constraint for *DeptName*.

Student(StdNo, StdFirstName, StdLastName, StdCity, StdState, StdZip, StdClass, StdMajor, StdGPA, StdEmail)
UNIQUE (StdEmail)
Faculty(FacNo, FacFirstName, FacLastName, FacSalary, FacHireDate, FacRank, *FacSupNo*, *DeptNo*)
FOREIGN KEY (FacSupNo) REFERENCES Faculty
FOREIGN KEY (DeptNo) REFERENCES Department
Department(DeptNo, DeptName)
UNIQUE (DeptName)
Offering(OfferNo, OffTerm, OffYear, *CourseNo*, *FacNo*)
FOREIGN KEY (CourseNo) REFERENCES Course
FOREIGN KEY (FacNo) REFERENCES Faculty
Course(CourseNo, CrsUnits, CrsDesc)
Enrollment(*StdNo*, *OfferNo*, EnrGrade)
FOREIGN KEY (StdNo) REFERENCES Student
FOREIGN KEY (OfferNo) REFERENCES Offering

Because of the small number of FDs added, you may not need to use the entire simple synthesis procedure. In the full simple synthesis procedure, you would add another table for *StdEmail* and *DeptName* in step 4 and then merge tables in step 5. Because *StdEmail* and *DeptName* are already part of other tables, you can simply add UNIQUE constraints.

7.5.2 Analyzing the Normalization Objective

As a design criterion, avoidance of modification anomalies is biased toward database changes. As you have seen, removing anomalies usually results in a database with many tables. A design with many tables makes a database easier to change but more difficult to query. If a database is used predominantly for queries, avoiding modification anomalies may not be an appropriate design goal. Chapters 12 to 15 describe databases for business intelligence in which the primary use is query rather than modification. In this situation, a design that is not fully normalized may be appropriate. Denormalization is the process of combining tables so that they are easier to query. In addition, physical design goals may conflict with logical design goals. Chapter 8 describes physical database design goals and the use of denormalization as a technique to improve query performance.

Another time to consider denormalization is when an FD is not important. The classic example contains the FDs *Zip* → *POCity*, *POState* in a customer table where *POState* and *POCity* refer to the location of the post office for the zip code. Note that in the US postal system, a postal (zip) code can cross city and state boundaries so that zip code does not determine city of residence. In some databases, the dependencies on post office city and state may not be important to maintain. If there is not a need

to manipulate zip codes independent of customers, the FDs can be safely ignored. However, there are databases in which it is important to maintain a table of zip codes independent of customer information. For example, if a retailer does business in many states and countries, a zip code table is useful to record sales tax rates.[7] If you ignore an FD in the normalization process, you should note that it exists but will not lead to any significant anomalies. Proceed with caution: most FDs will lead to anomalies if ignored.

Usage of Denormalization: consider violating BCNF as a design objective for a table when:

- An FD is not important to enforce as a candidate key constraint.
- A database is used predominantly for queries.
- Query performance requires fewer tables to reduce the number of join operations.

CLOSING THOUGHTS

This chapter described the impact of redundancy on the ability to change rows in a table. Redundancy in a table design causes modification anomalies leading to difficulties to insert, delete, and update rows of a table. Avoiding modification anomalies is the goal of normalization techniques. As a prerequisite to normalizing a table, you should list functional dependencies (FDs) for the table. This chapter provided guidelines about asserting FDs and using sample data to falsify FDs. After generating FDs for a table design, you should ensure that a table design satisfies rules about allowable FDs. This chapter described Boyce-Codd Normal Form (BCNF), a revised definition of third normal form, as a fundamental rule about FDs to ensure that a table is free of modification anomalies. The simple synthesis procedure was presented to analyze FDs and generate tables in BCNF. Providing a complete list of FDs is the most important part of the normalization process.

This chapter also described an approach to analyze M-way relationships (represented by associative entity types) using the concept of independence. If two relationships are independent, a third relationship can be derived obviating the need to store the third relationship. The independence concept is equivalent to multivalued dependency. 4NF prohibits redundancy caused by multivalued dependencies.

This chapter and the data modeling chapters (Chapters 5 and 6) emphasized fundamental skills for database development. After data modeling and normalization are complete, you are ready to implement the design, usually with a relational DBMS. Chapter 8 describes physical database design concepts and practices to facilitate your implementation work on relational DBMSs.

REVIEW CONCEPTS

- Redundancies in a table cause modification anomalies.
- Modification anomalies: unexpected side effects when inserting, updating, or deleting
- Functional dependency: a value neutral constraint similar to a candidate key
- Usage of sample data to eliminate (falsify) possible functional dependencies
- 1NF: no repeating columns in a table

[7] A former student made this comment about the database of a large electronics retailer.

- BCNF: revised and simplified definition combining older definitions for 2NF and 3NF
- BCNF definition: every determinant is a candidate key.
- Simple synthesis procedure: analyze FDs and produce tables in BCNF
- Use the simple synthesis procedure to analyze simple dependency structures
- Use relationship independence as a criterion to split M-way relationships into smaller relationships
- MVD: association with collections of values and independence among columns
- MVDs cause redundancy because rows can be derived using independence
- 4NF: no redundancies due to MVDs
- Use normalization techniques as a refinement tool rather than as an initial design tool
- Refinement involving conversion of ERD into a table design and identifying FDs and applying BCNF for each table
- Denormalize a table if FDs do not cause modification anomalies

QUESTIONS

1. What is an insertion anomaly?
2. What is an update anomaly?
3. What is a deletion anomaly?
4. What is the cause of modification anomalies?
5. What is a functional dependency?
6. How is a functional dependency like a candidate key?
7. Can a software design tool identify functional dependencies? Briefly explain your answer.
8. What is the meaning of an FD with multiple columns on the right-hand side?
9. Why should you be careful when writing FDs with multiple columns on the left-hand side?
10. What is a normal form?
11. What does 1NF prohibit?
12. What is a key column? This question pertains to material in Appendix 7.A.
13. What is a nonkey column? This question pertains to material in Appendix 7.A.
14. What kinds of FDs are not allowed in 2NF? This question pertains to material in Appendix 7.A.
15. What kinds of FDs are not allowed in 3NF? This question pertains to material in Appendix 7.A.
16. What is the combined definition of 2NF and 3NF? This question pertains to material in Appendix 7.A.
17. What kinds of FDs are not allowed in BCNF?
18. What two concepts are in the BCNF definition?
19. Why is the BCNF definition preferred to the original 3NF definition?
20. When are situations with multiple composite keys difficult to analyze?
21. Are situations with multiple composite candidate keys common?
22. What is the goal of the simple synthesis procedure?
23. What is a limitation of the simple synthesis procedure?

24. What is a transitive dependency?
25. Are transitive dependencies permitted in BCNF tables? Explain why or why not.
26. Why eliminate transitive dependencies in the FDs used as input to the simple synthesis procedure?
27. When is it necessary to perform the fifth step of the simple synthesis procedure?
28. How is relationship independence similar to statistical independence?
29. What kind of redundancy is caused by relationship independence?
30. How many columns does an MVD involve?
31. What is a multivalued dependency (MVD)?
32. What is the relationship between MVDs and FDs?
33. What is a nontrivial MVD?
34. What is the goal of 4NF?
35. What are the advantages of using normalization as a refinement tool rather than as an initial design tool?
36. Why is 5NF not considered a practical normal form?
37. Why is DKNF not considered a practical normal form?
38. When is denormalization useful? Provide an example to depict when it may be beneficial for a table to violate 3NF.
39. What are the two ways to use normalization in the database development process?
40. Why does this book recommend using normalization as a refinement tool, not as an initial design tool?
41. How many sample rows are necessary to falsify a possible FD?
42. Explain the pattern in sample data to falsify the FD $X \rightarrow Y$.
43. For a large collection of functional dependencies, should you list the functional dependencies or draw a functional dependency diagram?
44. What is a value neutral constraint? Is a functional dependency a value neutral constraint? Explain your answer.
45. Why are functional dependencies about 1-M relationships confusing to identify?
46. Can you define a functional dependency for an M-N relationship without attributes?
47. What kind of FDs should you focus when using normalization as a refinement tool?
48. Do you need to perform the all steps of the simple synthesis procedure when using normalization as a refinement tool?

PROBLEMS

Besides the problems presented here, the textbook's website contains a case study for additional practice. To supplement the examples in this chapter, the case study provides a complete database design case including conceptual data modeling, schema conversion, and normalization.

1. For the big university database table, list FDs with the column *StdCity* as the determinant that are falsified by rows in Table 7-P1. For each FD, identify the sample rows that falsify it. Remember that it takes two rows to falsify an FD. The sample rows are repeated in Table 7-P1 for your reference.
2. Following on problem 1, list FDs with the column *StdCity* as the determinant that the sample rows do not falsify. For each FD, add one or more sample rows and then identify the sample rows that falsify the FD. Remember that it takes two rows to falsify an FD.

StdNo	StdCity	StdClass	OfferNo	OffTerm	OffYear	EnrGrade	CourseNo	CrsDesc
S1	SEATTLE	JUN	O1	FALL	2017	3.5	C1	DB
S1	SEATTLE	JUN	O2	FALL	2017	3.3	C2	VB
S2	BOTHELL	JUN	O3	SPRING	2018	3.1	C3	OO
S2	BOTHELL	JUN	O2	FALL	2017	3.4	C2	VB

TABLE 7-P1
Sample Data for the Big University Database Table

3. For the big patient table, list FDs with the column *PatZip* as the determinant that are falsified by sample rows in Table 7-P2. For the other FDs, identify sample rows that falsify it. Remember that it takes two rows to falsify an FD. The sample rows are repeated in Table 7-P2 for your reference.
4. Following on problem 3, list FDs with the column *PatZip* as the determinant that are not falsified by sample rows. Exclude the FD *PatZip* → *PatCity* because it is a valid FD. For each FD, add one or more sample rows and then identify the sample rows that falsify the FD. Remember that it takes two rows to falsify an FD.
5. Add sample rows to Table 7-P2 to falsify the following FDs. Remember that it takes two rows to falsify an FD.
   - PatNo → VisitNo
   - PatNo → ProvNo
   - PatAge → PatZip
   - PatAge → PatCity
   - PatAge → PatNo
6. Apply the simple synthesis procedure to the FDs of the big patient table. The FDs are repeated in Table 7-P3 for your reference. Show the result of each step in the procedure. Include the primary keys, foreign keys, and other candidate keys in the final list of tables.
7. The FD diagram in Figure 7.P1 depicts FDs among columns in an order entry database. Figure 7.P1 shows FDs with determinants *CustNo*, *OrderNo*, *ItemNo*, the combination of *OrderNo* and *ItemNo*, the combination of *ItemNo* and *PlantNo*, and the combination of *OrderNo* and *LineNo*. The combination of *OrderNo* and

VisitNo	VisitDate	PatNo	PatAge	PatCity	PatZip	ProvNo	ProvSpecialty	Diagnosis
V10020	1/13/2018	P1	35	DENVER	80217	D1	INTERNIST	EAR INFECTION
V10020	1/13/2018	P1	35	DENVER	80217	D2	NURSE PRACTITIONER	INFLUENZA
V93030	1/20/2018	P3	17	ENGLEWOOD	80113	D2	NURSE PRACTITIONER	PREGNANCY
V82110	1/18/2018	P2	60	BOULDER	85932	D3	CARDIOLOGIST	MURMUR

TABLE 7-P2
Sample Data for the Big Patient Table

PatNo → PatAge, PatCity, PatZip
PatZip → PatCity
ProvNo → ProvSpecialty
VisitNo → PatNo, VisitDate, PatAge, PatCity, PatZip
VisitNo, ProvNo → Diagnosis

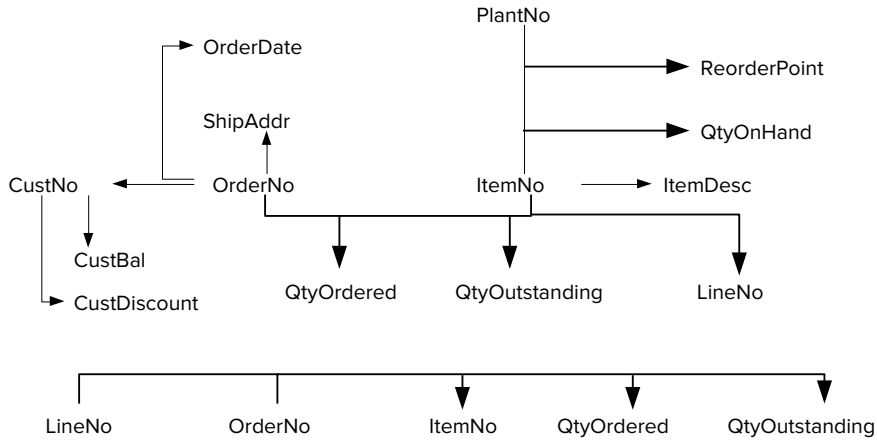
TABLE 7-P3
List of FDs for the Big Patient Table

ItemNo determines *LineNo*, *QtyOrdered*, and *QtyOutstanding*. In the bottom FDs, the combination of *LineNo* and *OrderNo* determines *ItemNo*, *QtyOrdered*, and *QtyOutstanding*. To test your understanding of dependency diagrams, convert the dependency diagram into a list of dependencies organized by the LHSs.

8. Using the FD diagram (Figure 7.P1) and the FD list (solution to problem 7) as guidelines, make a table with sample data. There are two candidate keys for the underlying table: the combination of *OrderNo*, *ItemNo*, and *PlantNo* and the combination of *OrderNo*, *LineNo*, and *PlantNo*. Using the sample data, identify insertion, update, and deletion anomalies in the table.
9. Derive 2NF tables starting with the FD list from problem 7 and the table from problem 8. This problem applies to Appendix 7.A.
10. Derive 3NF tables starting with the FD list from problem 7 and the 2NF tables from problem 9. This problem applies to Appendix 7.A.
11. Following on problems 7 and 8, apply the simple synthesis procedure to produce BCNF tables.
12. Modify your table design in problem 11 if the shipping address (*ShipAddr*) column determines customer number (*CustNo*). Do you think that this additional FD is reasonable? Briefly explain your answer.
13. Go back to the original FD diagram in which *ShipAddr* does not determine *CustNo*. How does your table design change if you want to keep track of a master list of shipping addresses for each customer? Assume that you do not want to lose a shipping address when an order is deleted.
14. Using the following FD list for a simplified expense report database, identify insertion, update, and deletion anomalies if all columns are in one table (big expense report table). There are two candidate keys for the big expense report table: *ExpItemNo* (expense item number) and the combination of *CatNo* (category number) and *ERNo* (expense report number). *ExpItemNo* is the primary key of the table.

    - ERNo → UserNo, ERSubmitDate, ERStatusDate
    - ExpItemNo → ExpItemDesc, ExpItemDate, ExpItemAmt, CatNo, ERNo
    - UserNo → UserFirstName, UserLastName, UserPhone, UserEmail
    - CatNo → CatName, CatLimit
    - ERNo, CatNo → ExpItemNo
    - UserEmail → UserNo
    - CatName → CatNo

FIGURE 7.P1
Dependency Diagram for the Big Order Entry Table



15. Using the FD list in problem 14, identify the FDs that violate 2NF. Using knowledge of the FDs that violate 2NF, design a collection of tables that satisfies 2NF but not 3NF. This problem applies to Appendix 7.A.
16. Using the FD list in problem 14, identify the FDs that violate 3NF. Using knowledge of the FDs that violate 2NF, design a collection of tables that satisfies 3NF. This problem applies to Appendix 7.A.
17. Apply the simple synthesis procedure to produce BCNF tables using the FD list given in problem 14. Show the results of each step in your analysis.
18. Using the following FD list for a simplified graduate student advising database, identify insertion, update, and deletion anomalies if all columns are in one table (big graduate student advising table). There is only one candidate key for the big graduate student advising table: the combination of *PlanNo*, *CourseNo*, and *PaperNo*.
    - StdNo → StdName, StdAdmitSems, StdAdmitYear, StdStatus, StdEmail
    - StdEmail → StdNo, StdStatus
    - CourseNo → CrsDesc, CrsUnits, CrsDeptName, CrsCollName
    - PlanNo → PlanDate, PlanAdvName, StdNo, PlanApproval, StdName
    - PlanNo, CourseNo → Semester, Year, CreditType, Grade
    - PlanNo, PaperNo → DateSubmit, DateDecided, Decision, PaperTitle, StdNo
19. Using the FD list in problem 18, identify the FDs that violate 2NF. Using knowledge of the FDs that violate 2NF, design a collection of tables that satisfies 2NF but not 3NF. This problem applies to Appendix 7.A.
20. Using the FD list in problem 18, identify the FDs that violate 3NF. Using knowledge of the FDs that violate 2NF, design a collection of tables that satisfies 3NF. This problem applies to Appendix 7.A.
21. Apply the simple synthesis procedure to produce BCNF tables using the FD list given in problem 18. Show the results of each step in your analysis.
22. Convert the ERD in Figure 7.P2 into tables and perform further normalization as needed. After converting the ERD to tables, specify FDs for each table. Since the primary key of each table determines the other columns, you should only identify FDs in which the LHS is not the primary key. If a table is not in BCNF, explain why and split it into two or more tables that are in BCNF.

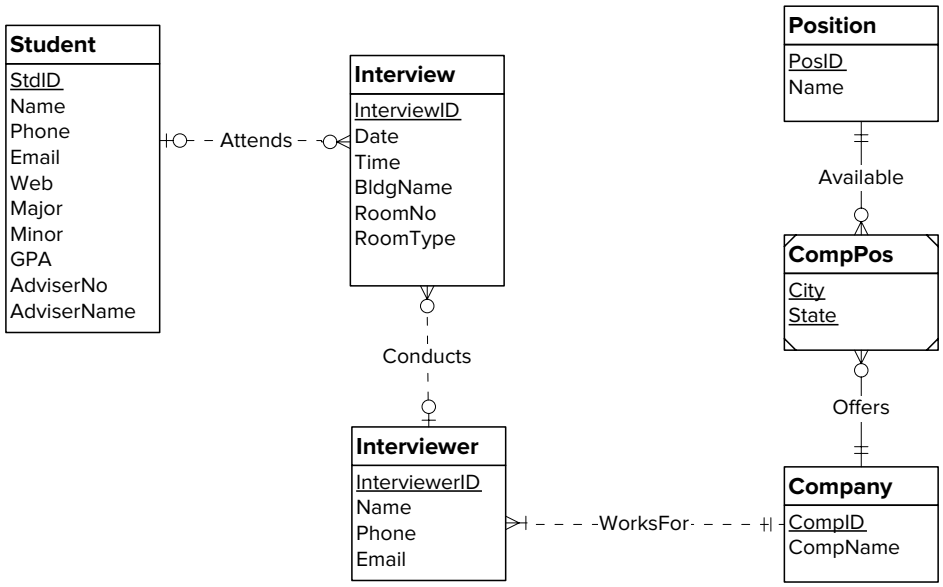
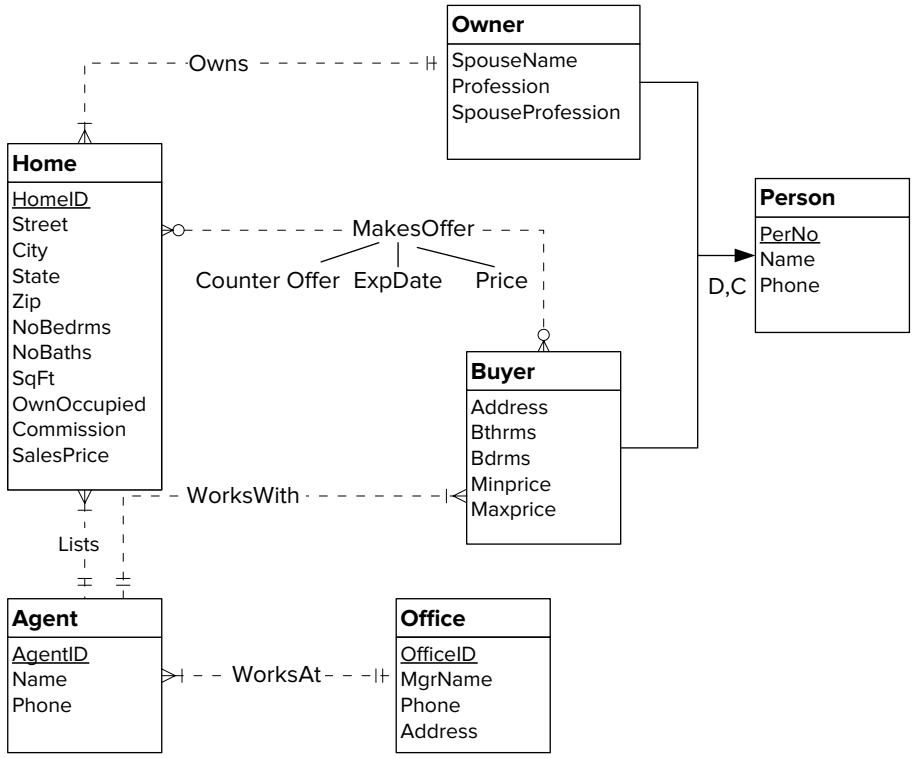
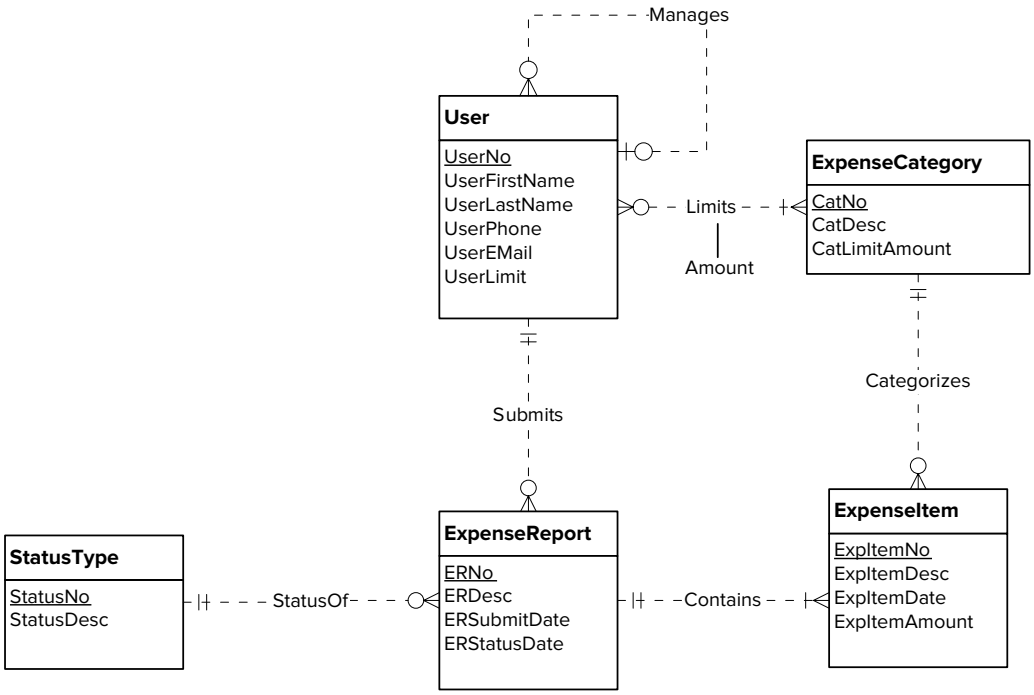


FIGURE 7.P2
ERD for Problem 22

23. Convert the ERD in Figure 7.P3 into tables and perform further normalization as needed. After the conversion, specify FDs for each table. Since the primary key of each table determines the other columns, you should only identify FDs in which the LHS is not the primary key. If a table is not in BCNF, explain why and split it into two or more tables that are in BCNF.
24. Convert the ERD in Figure 7.P4 into tables and perform further normalization as needed. After the conversion, write down FDs for each table. Since the primary key of each table determines the other columns, you should only identify FDs in which the LHS is not the primary key. If a table is not in BCNF, explain why and split it into two or more tables that are in BCNF. In the *User* entity type, *UserEmail* is unique. In the *ExpenseCategory* entity type, *CatDesc* is unique. In the *StatusType* entity type, *StatusDesc* is unique. For the *ExpenseItem* entity type, the combination of the *Categorizes* and *Contains* relationships are unique.
25. Convert the ERD in Figure 7.P5 into tables and perform further normalization as needed. After the conversion, write down FDs for each table. Since the primary key of each table determines the other columns, you should only identify FDs in which the LHS is not the primary key. If a table is not in BCNF, explain why and split it into two or more tables that are in BCNF. In the *Employee* entity type, each department has one manager. All employees in a department are supervised by the same manager. For the other entity types, *FacName* is unique in *Facility*, *ResName* is unique in *Resource*, and *CustName* and *CustEmail* are unique in *Customer*.
26. Extend the solution to the problem described in Section 7.2.4 about a database to track submitted conference papers. In the description, underlined parts are new. Write down the new FDs. Using the simple synthesis procedure, design a collection of tables in BCNF. Note dependencies that are not important to the problem and relax your design from BCNF as appropriate. Justify your reasoning.

FIGURE 7.P3
ERD for Problem 23





FIGUE 7.P4
ERD for Problem 24

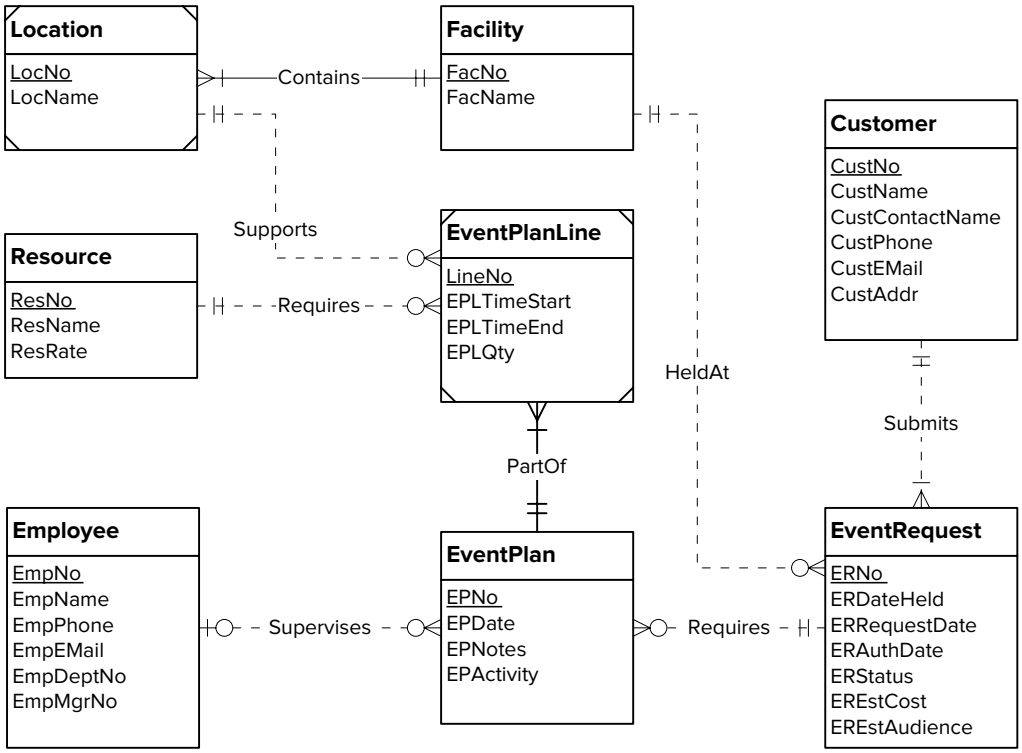


FIGURE 7.P5
ERD for Problem 25

- Author information includes a unique author number, a name, a mailing address, and a unique but optional electronic address.
- Paper information includes the list of authors, the primary author, the paper number, the title, the abstract, the review status (pending, accepted, rejected), and a list of subject categories.

- Reviewer information includes the reviewer number, the name, the mailing address, a unique but optional electronic address, and a list of expertise categories.
- A completed review includes the reviewer number, the date, the paper number, comments to the authors, comments to the program chairperson, and ratings (overall, originality, correctness, style, and relevance).
- The conference organizer should maintain master lists of expertise categories and subject categories. Each category includes a category number and name.
- Accepted papers are assigned to sessions. Each session has a unique session identifier, a list of papers, a presentation order for each paper, a session title, a session chairperson, a room, a date, a start time, and a duration. Note that each accepted paper can be assigned to only one session.

27. For the following description of an airline reservation database, identify functional dependencies and construct normalized tables. Using the simple synthesis procedure, design a collection of tables in BCNF. Note dependencies that are not important to the problem and relax your design from BCNF as appropriate. Justify your reasoning.

    The Fly by Night Operation is a newly formed airline aimed at the burgeoning market of clandestine travelers (fugitives, spies, con artists, scoundrels, deadbeats, cheating spouses, politicians, etc.). The Fly by Night Operation needs a database to track flights, customers, fares, airplane performance, and personnel assignment. Since the Fly by Night Operation is touted as a "fast way out of town," individual seats are not assigned, and flights of other carriers are not tracked. More specific notes about different parts of the database are listed below:

    - Information about a flight includes its unique flight number, its origin, its (supposed) destination, and (roughly) estimated departure and arrival times. To reduce costs, the Fly by Night Operation only has nonstop flights with a single origin and destination.
    - Flights are scheduled for one or more dates with an airplane and a crew assigned to each scheduled flight, and the remaining capacity (seats remaining) noted. In a crew assignment, the employee number and the role (e.g., captain, flight attendant) are noted.
    - Airplanes have a unique serial number, a model, a capacity, and a next scheduled maintenance date.
    - The maintenance record of an airplane includes a unique maintenance number, a date, a description, the serial number of the plane, and the employee responsible for the repairs.
    - Employees have a unique employee number, a name, a phone, and a job title.
    - Customers have a unique customer number, a phone number, and a name (typically an alias).
    - Records are maintained for reservations of scheduled flights including a unique reservation number, a flight number, a flight date, a customer number, a reservation date, a fare, and the payment method (usually cash but occasionally someone else's check or credit card). If the payment is by credit card, a credit card number and an expiration date are part of the reservation record.

28. For the following description of an accounting database, identify functional dependencies and construct normalized tables. Using the simple synthesis procedure, design a collection of tables in BCNF. Note dependencies that are

not important to the problem and relax your design from BCNF as appropriate. Justify your reasoning.

- The primary function of the database is to record entries into a register. A user can have multiple accounts and there is a register for each account.
- Information about users includes a unique user number, a name, a street address, a city, a state, a zip, and a unique but optional e-mail address.
- Accounts have attributes including a unique number, a unique name, a start date, a last check number, a type (checking, investment, etc.), a user number, and a current balance (computed). For checking accounts, the bank number (unique), the bank name, and the bank address are also recorded.
- An entry contains a unique number, a type, an optional check number, a payee, a date, an amount, a description, an account number, and a list of entry lines. The type can have various values including ATM, next check number, deposit, and debit card.
- In the list of entry lines, the user allocates the total amount of the entry to categories. An entry line includes a category name, a description of the entry line, and an amount.
- Categories have other attributes not shown in an entry line: a unique category number (name is also unique), a description, a type (asset, expense, revenue, or liability), and a tax-related status (yes or no).
- Categories are organized in hierarchies. For example, there is a category Auto with subcategorizes Auto:fuel and Auto:repair. Categories can have multiple levels of subcategories.

29. For the ERDs in Figure 7.P6, describe assumptions under which the ERDs correctly depict the relationships among operators, machines, and tasks. In each case, choose appropriate names for the relationships and describe the meaning of the relationships. In part (b) you should also choose the name for the new entity type.

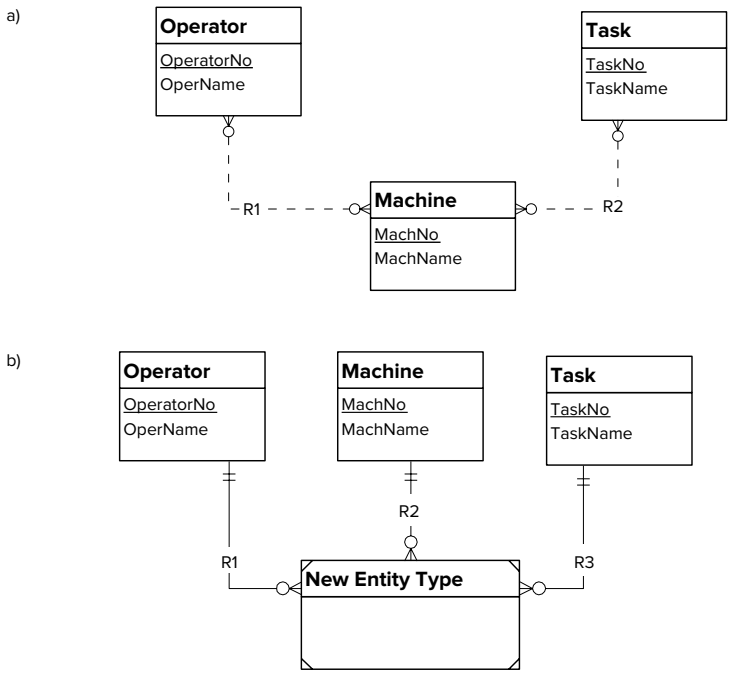


FIGURE 7.P6
ERDs for Problem 29

30. For the following description of a database to support physical plant operations, identify functional dependencies and construct normalized tables. Using the simple synthesis procedure, design a collection of tables in BCNF. Note dependencies that are not important to the problem and relax your design from BCNF as appropriate. Justify your reasoning.

    Design a database to assist physical plant personnel in managing key cards for access to buildings and rooms. The primary purpose of the database is to ensure proper accounting for all key cards.

    - A building has a unique building number, a unique name, and a location within the campus.
    - A room has a unique room number, a size (physical dimensions), a capacity, a number of entrances, and a description of equipment in the room. Each room is located in exactly one building. The room number includes a building identification and followed by an integer number. For example, room number KC100 identifies room 100 in the King Center (KC) building.
    - An employee has a unique employee number, a name, a position, a unique e-mail address, a phone, and an optional room number in which the employee works.
    - Magnetically encoded key cards are designed to open one or more rooms. A key card has a unique card number, a date encoded, a list of room numbers that the key card opens, and the number of the employee authorizing the key card. A room may have one or more key cards that open it. A key type must be authorized before it is created.

31. For the ERDs in Figure 7.P7, describe assumptions under which the ERDs correctly depict the relationships among work assignments, tasks, and materials. A work assignment contains the scheduled work for a construction job at a specific location. Scheduled work includes the tasks and materials needed for the construction job. In each case, choose appropriate names for the relationships and describe the meaning of the relationships. In part (b) you should also choose the name for the new entity type.
32. For the following description of a database to support volunteer tracking, identify functional dependencies and construct normalized tables. Using the simple synthesis procedure, design a collection of tables in BCNF. Note dependencies that are not important to the problem and relax your design from BCNF as appropriate. Justify your reasoning.

    Design a database to support organizations that need to track volunteers, volunteer areas, events, and hours worked at events. The system will be initially deployed for charter schools that have mandatory parent participation as volunteers. Volunteers register as a dual- or single-parent family. Volunteer coordinators recruit volunteers for volunteer areas. Event organizers recruit volunteers to work at events. Some events require a schedule of volunteers while other events do not use a schedule. Volunteers work at events and record the time worked.

    - For each family, the database records the unique family number, the first and last name of each parent, the home and business phones, the mailing address (street, city, state, and zip), and an optional e-mail address. For single-parent households, information about only one parent is recorded.
    - For each volunteer area, the database records the unique volunteer area, the volunteer area name, the group (faculty senate or parent teacher association) controlling the volunteer area, and the family coordinating the volunteer area. In some cases, a family coordinates more than one volunteer area.

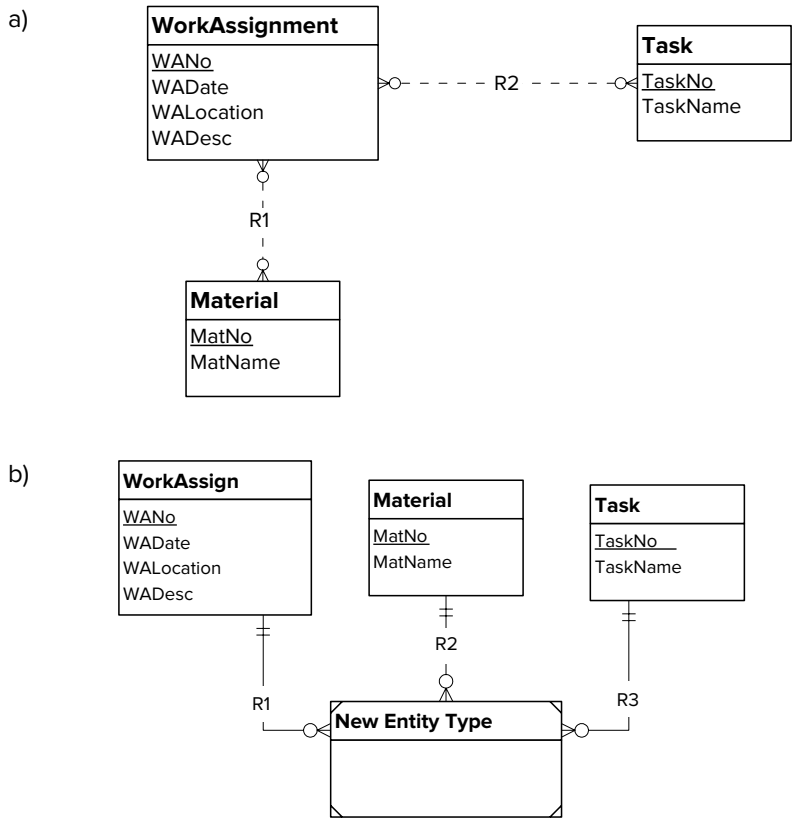


FIGURE 7.P7
ERDs for Problem 31

- For events, the database records the unique event number, the event description, the event date, the beginning and ending time of the event, the number of required volunteers, the event period and expiration date if the event is a recurring event, the volunteer area, and the list of family volunteers for the event. Families can volunteer in advance for a collection of events.
- After completing a work assignment, hours worked are recorded. The database contains the first and last name of the volunteer, the family the volunteer represents, the number of hours worked, the optional event, the date worked, the location of the work, and optional comments. Usually the volunteer is one of the parents of the family, but occasionally the volunteer is a friend or relative of the family. The event is optional to allow volunteer hours for activities not considered as events.

33. For the big order database table, list FDs with the column *CustNo* as the determinant that are falsified by sample rows. For each FD, identify the sample rows that falsify it. Remember that it takes two rows to falsify an FD. The sample rows are repeated in Table 7-P4 for your reference.

TABLE 7-P4
Sample Data for the Big Order Database Table

OrdNo	ItemNo	QtyOrd	PlantNo	CustNo	CustBal	CustDisc	ReordPt	OrdDate
O1	I1	10	P1	C1	100	0.10	10	1/15/2018
O1	I2	10	P1	C1	100	0.10	10	1/15/2018
O2	I3	5	P1	C2	200	0.05	20	1/16/2018
O2	I4	5	P1	C2	200	0.05	10	1/16/2018
O3	I1	10	P2	C1	100	0.10	10	1/17/2018

TABLE 7-P5
Sample Rows for the Big Customer Portfolio Table

CustId	CName	CZip	Stock1	Price1	Shares1	Stock2	Price2	Shares2
3	Sanchez	80217	IBM	100	20			
4	Smith	80113	ATT	40	25	IBM	100	10
4	Smith	80113	GM	50	20			

34. Following on problem 33, list FDs with the column *CustNo* as the determinant that the sample rows do not falsify. Do not show falsifications for *CustNo* → *CustBal*, *CustDisc* because these FDs are true. For each FD, add one or more sample rows and then identify sample rows that falsify the FD. Remember that it takes two rows to falsify an FD.
35. Add sample rows to Table 7-P4 to demonstrate falsifications of the following FDs. Remember that it takes two rows to falsify an FD.
    - ItemNo, PlantNo → CustNo
    - ItemNo, PlantNo → CustBal
    - ItemNo, PlantNo → CustDisc
    - ItemNo, PlantNo → OrdNo
    - ItemNo, PlantNo → OrdDate
36. Table 7-P5 shows sample rows with basic customer details along with a listing of a customer's stock portfolio. For brevity, some basic columns have been omitted. Each row of the table allows two stocks to be listed. For customers with more than two stocks in their portfolio, additional rows are created. For example, customer with *CustId* 4 has two rows with three stocks. Redesign the table so that it has only one stock per row.
37. In the revised table with one stock per row, identify the functional dependencies. Each stock appears at most one time in a customer's portfolio. A customer may have many stocks in a portfolio. What is the primary key of the table?
38. Revise the table design so that it is in BCNF using the FDs identified in problem 37. Identify the primary key, candidate keys, and foreign keys in your design.
39. The *EmpSkill* table has the following FDs. The *EmpSkill* table has two candidate keys, <*EmpNo*, *SkillNo*> and <*EmpEmail*, *SkillNo*>. Is *EmpSkill* table in BCNF? Explain your reasoning.

    EmpSkill(EmpNo, SkillNo, EmpEmail, PayRate)
    EmpNo, SkillNo → PayRate
    EmpEmail, SkillNo → PayRate
    EmpNo → EmpEmail
    EmpEmail → EmpNo

40. If the *EmpSkill* table in problem 39 is not in BCNF, apply the simple synthesis procedure to achieve a table design satisfying BCNF.

REFERENCES FOR FURTHER STUDY

The subject of normalization can be much more detailed than described in this chapter. For a more mathematical presentation of normalization, consult computer science books such as Elmasri and Navathe (2017). The simple synthesis procedure was adapted from Hawryszkiewycz (1984). For a classic tutorial on normalization, consult Kent (1983). Fagin (1981) describes domain key normal form, the ultimate normal form. Armstrong (1974) presented axioms for deriving functional dependencies of which transitivity is the most prominent. The DevX Database Zone (www.devx.com) has practical advice about database development and data modeling.

chapter

8

Physical Database Design

Learning Objectives

This chapter describes physical database design, the final phase of the database development process. Physical database design transforms a table design from the logical design phase into an efficient implementation that supports all applications using the database. After this chapter, the student should have acquired the following knowledge and skills:

- Describe the inputs, outputs, and objectives of physical database design
- Appreciate difficulties of performing physical database design and the need for periodic review of physical database design choices
- Explain characteristics of sequential, Btree, hash, bitmap, and columnstore file structures
- Understand choices made by a query optimizer and areas in which optimization decisions can be improved
- Understand trade-offs in index selection and denormalization decisions
- Understand the need for computer-aided tools to assist with physical database design decisions

OVERVIEW

Chapters 5 to 7 covered the conceptual and logical design phases of database development. You learned about entity relationship diagrams, data modeling practice, schema conversion, and normalization. This chapter extends your database design skills by explaining the process to achieve an efficient implementation of a table design.

To become proficient in physical database design, you need to understand the process and environment. This chapter describes the process of physical database design including the inputs, outputs, and objectives along with two critical parts of the environment, file structures and query optimization. Most choices in physical database design relate to characteristics of file structures and query optimization decisions.

After understanding the process and environment, you are ready to perform physical database design. In performing physical database design, you should provide detailed inputs and make choices to balance needs of retrieval and update applications. This chapter describes the complexity of table profiles and application profiles and their importance for physical design decisions. Index selection is the most important choice of physical database design. This chapter describes trade-offs in index selection and provides index selection rules that you can apply to moderate-size databases. In addition to index selection, this chapter presents denormalization, record formatting, and parallel processing as techniques to improve database performance.

8.1 OVERVIEW OF PHYSICAL DATABASE DESIGN

Decisions in the physical database design phase involve the storage level of a database. Collectively, the storage level decisions are known as the internal schema. This section describes the storage level as well as the objectives, inputs, and outputs of physical database design.

8.1.1 Storage Level of Databases

Physical Record
collection of bytes that are transferred between volatile storage in main memory and stable storage on a disk. The number of physical record accesses is an important measure of database performance.

The storage level is closest to the hardware and operating system. At the storage level, a database consists of physical records (also known as blocks or pages) organized into files. A **physical record** is a collection of bytes that are transferred between volatile storage in main memory and stable storage on a disk (magnetic hard drive or solid state device). Main memory is considered volatile storage because the contents of main memory may be lost if a failure occurs. A file is a collection of physical records organized for efficient access. Figure 8.1 depicts relationships between logical records (rows of a table) and physical records stored in a file. Typically, a physical record contains multiple logical records. The size of a physical record is a power of two such as 1,024 (2^{10}) or 4,096 (2^{12}) bytes. A large logical record may be split over multiple physical records. Another possibility is that logical records from more than one table are stored in the same physical record.

The DBMS and the operating system work together to satisfy requests for logical records made by applications. Figure 8.2 depicts the process of transferring physical and logical records between a disk, DBMS buffers, and application buffers. Normally, the DBMS and the application have separate memory areas known as buffers. When an application makes a request for a logical record, the DBMS locates the physical record containing it. In the case of a read operation, the operating system transfers the physical record from disk to the memory area of the DBMS. The DBMS then transfers

FIGURE 8.1
Relationships between Logical Records (LR) and Physical Records (PR)

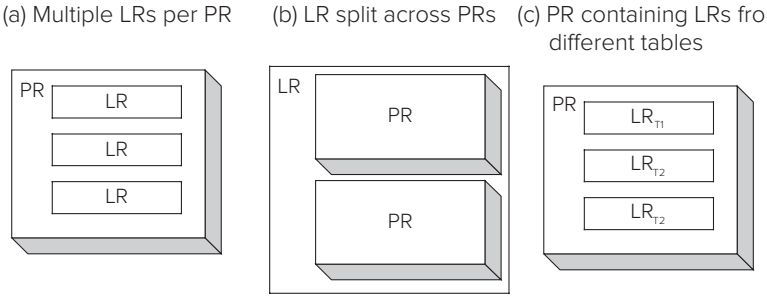
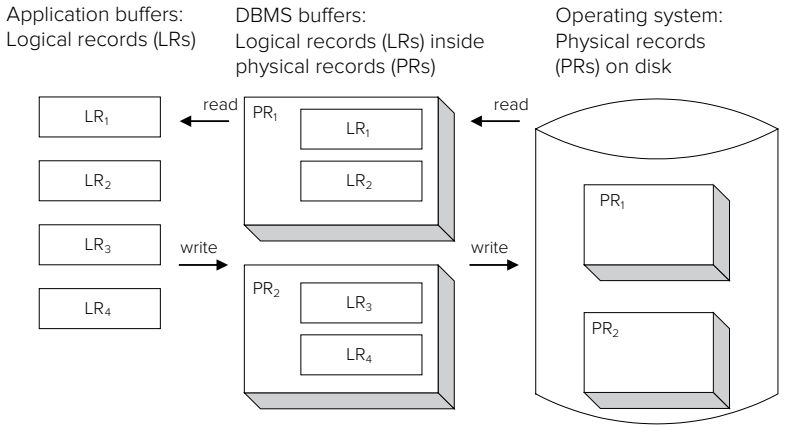


FIGURE 8.2
Transferring Physical Records



the logical record to the application's buffer. In the case of a write operation, the transfer process is reversed.

A logical record request may not result in a physical record transfer because of buffering. The DBMS tries to anticipate the needs of applications so that corresponding physical records already reside in the DBMS buffers. A significant difficulty about predicting database performance is the uncertainty about the contents of DBMS buffers. If a DBMS buffer contains a requested logical record, a physical record transfer is not necessary. For example, if multiple applications are accessing the same logical records, the corresponding physical records may reside in the DBMS buffers. Consequently, the uncertainty about the contents of DBMS buffers can make physical database design difficult.

8.1.2 Objectives and Constraints

The goal of physical database design is to minimize response time to access and change a database. Because response time is difficult to estimate directly, minimizing computing resources is used as a substitute measure. The resources that are consumed by database processing are physical record transfers, central processing unit (CPU) operations, main memory, and disk space. The latter two resources (main memory and disk space) are considered as constraints rather than resources to minimize. Minimizing main memory and disk space can lead to high response times.

The number of physical record accesses limits the performance of most database applications. A physical record access may involve mechanical movement of a disk including rotation and magnetic head movement. Mechanical movement is generally much slower than electronic switching of main memory. The speed of a disk access is measured in milliseconds (thousandths of a second) whereas a memory access is measured in nanoseconds (billionths of a second). Thus, a physical record access may be several orders of magnitude slower than a main memory access. Reducing the number of physical record accesses will usually improve response time.

The recent movement to incorporate solid state devices changes limitations on database performance in some applications. Solid state devices use electronic switching to sharply reduce latency or delay involved with electromechanical disks. Solid state storage can be used in place of hard drives for moderate-size databases. For larger databases, solid state drives complement hard drives providing a faster intermediate level of storage for frequently accessed data.

CPU usage also can be a factor in some database applications. For example, sorting requires a large number of comparisons and assignments. These operations, performed by the CPU, are many times faster than a physical record access, however. To accommodate both physical record accesses and CPU usage, a weight can be used to combine them into one measure. The weight is usually close to 0 to reflect that many CPU operations can be performed in the time to perform one physical record transfer with a hard disk. For solid state devices, the weight is larger to reflect much faster electronic switching to read data. Random access times for solid state devices are about 50 times faster than hard drives (0.1 milliseconds versus 5 milliseconds). Transfer rates can be as much as 5 times faster for solid state storage than hard drives.

Combined Measure of Database Performance: *PRA* + *W* * *CPU-OP* where

PRA is the number of physical record accesses,
CPU-OP is the number of CPU operations such as comparisons and assignments, and
W is a weight, a real number between 0 and 1.

The objective of physical database design is to minimize the combined measure for all applications using the database. Generally, improving performance on retrieval applications comes at the expense of update applications and vice versa. Therefore, an

important theme of physical database design is to balance the needs of retrieval and update applications.

The measures of performance are too detailed to estimate manually except for simple situations. Complex optimization software calculates estimates using detailed cost formulas. The optimization software is usually part of an SQL compiler. Understanding the nature of the performance measure helps one to interpret choices made by the optimization software.

For most choices in physical database design, the amounts of main memory and disk space are usually fixed. In other words, main memory and disk space are constraints of the physical database design process. As with constraints in other optimization problems, you should consider the effects of changing the given amounts of main memory and disk space. Increasing the amounts of these resources can improve performance. The amount of performance improvement may depend on many factors such as the DBMS, table design, and applications using the database.

8.1.3 Inputs, Outputs, and Environment

Physical database design consists of a number of different inputs and outputs as depicted in Figure 8.3 and summarized in Table 8-1. The starting point is the table design from the logical database design phase. The table and application profiles are used specifically for physical database design. Because these inputs are so critical to the physical database design process, they are discussed in more detail in Section 8.2. The most important outputs are decisions about file structures and data placement. Section 8.5 discusses these decisions in more detail. For simplicity, decisions about other outputs are made separately even though the outputs can be related. For example, file structures are usually selected separately from denormalization decisions even though denormalization decisions can affect file structure decisions. Thus, physical database design is better characterized as a sequence of decision-making processes rather than one large process.

Knowledge about file structures and query optimization are in the environment of physical database design rather than being inputs. The knowledge can be embedded in database design tools. If database design tools are not available, a designer informally uses knowledge about the environment to make physical database decisions. Acquiring the knowledge can be difficult because much of it is specific to each DBMS. Because knowledge of the environment is so crucial in the physical database design process, Sections 8.3 and 8.4 discuss it in more detail.

8.1.4 Difficulties

Before learning more details about physical database design, you should understand difficulties of physical database design. Difficulties involve the number of decisions,

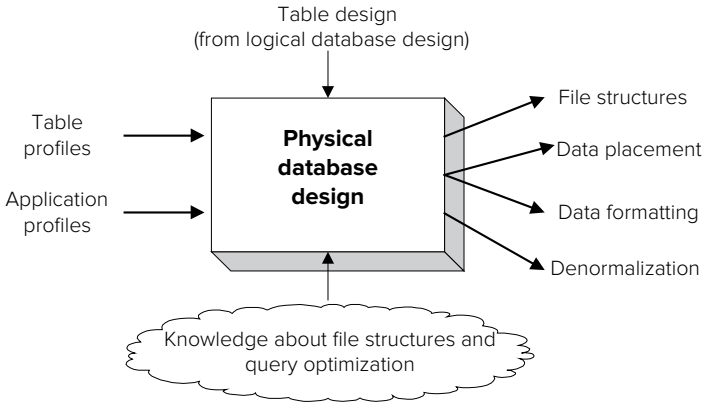


FIGURE 8.3
Inputs, Outputs, and Environment of Physical Database Design

TABLE 8-1
Summary of Inputs, Outputs, and Environment of Physical Database Design

Item	Description
Inputs	
Table profiles	Statistics about each table such as the number of physical records, number of rows, unique column values, and distribution of column values
Application profiles	Statistics for each form, report, and query such as the tables accessed/updated, the frequency of access/update, and values used in search requests
Outputs	
File structures	Method of organizing physical records for each table
Data placement	Criteria for arranging physical records in close proximity
Data formatting	Usage of compression and derived data
Denormalization	Combining separate tables into a single table
Environment knowledge	
File structures	Characteristics such as operations supported and cost formulas
Query optimization	Access decisions made by the optimization component for each query

relationships among decisions, detailed inputs, complex environment, and uncertainty in predicting physical record accesses. These difficulties are briefly discussed below. In the remainder of this chapter, you should remember these difficulties.

- The number of possible choices available to a designer can be large. For databases with many columns, the number of possible choices can be too large to evaluate even on large computers.
- Some decisions cannot be made in isolation. For example, file structure decisions for one table can influence the decisions for other tables.
- The quality of decisions is limited by the precision of table and application profiles. However, these inputs can be large and difficult to collect. In addition, the inputs change over time so that periodic revision is necessary.
- The environment knowledge is specific to each DBMS. Much of the knowledge is either a trade secret or too complex to apply without software assistance.
- The number of physical record accesses is difficult to predict because of uncertainty about the contents of DBMS buffers. The uncertainty arises because the mix of applications accessing the database is constantly changing.
- The usage of two types of permanent storage (magnetic and solid state) complicates physical record access cost estimation. Physical record accesses should be divided between hard disk and solid state accesses due to their speed differences. However, predicting the number of physical records on the two types of storage can be as difficult as predicting the contents of DBMS buffers.

8.2 INPUTS OF PHYSICAL DATABASE DESIGN

Physical database design requires inputs specified in sufficient detail. Inputs specified without enough detail can lead to poor decisions in physical database design and query optimization. This section describes the level of detail recommended for both table profiles and application profiles.

8.2.1 Table Profiles

A table profile summarizes a table as a whole, the columns within a table, and relationships between tables as shown in Table 8-2. Because table profiles are tedious to construct manually, most DBMSs provide tools to construct them automatically. The designer may need to periodically execute the tools so that profiles do not become

TABLE 8-2
Typical Components of a Table Profile

Component	Statistics
Table	Number of rows and physical records
Column	Number of unique values, distribution of values, correlation among columns
Relationship	Distribution of the number of related rows

obsolete. For large databases, table profiles may be estimated on samples of a database. Using the entire database can be too time-consuming and disruptive.

For column and relationship summaries, the distribution conveys the number of rows and related rows for column values. The distribution of values can be specified in a number of ways. A simple way is to assume that the column values are uniformly distributed. Uniform distribution means that each value has an equal number of rows. If the uniform value assumption is made, only the minimum and maximum values must be stored.

A more detailed way to specify a distribution is to use a histogram. A histogram is a two-dimensional graph in which the *x*-axis represents column ranges and the *y*-axis represents the number of rows. For example, the first bar in Figure 8.4 means that 9,000 rows have a salary between $10,000 and $50,000. Traditional equal-width histograms do not work well with skewed data because a large number of ranges are necessary to control estimation errors. In Figure 8.4, estimating the number of employee rows using the first two ranges leads to large estimation errors because more than 97% of employees have salaries less than $80,000. For example, you would calculate about 1,125 rows (12.5% of 9,000) to estimate the number of employees earning between $10,000 and $15,000 using Figure 8.4. However, the actual number of rows is much smaller because few employees earn less than $15,000.

Because skewed data can lead to poor estimates using traditional (equal-width) histograms, most DBMSs use equal-height histograms as shown in Figure 8.5. In an equal-height histogram, each range contains about the same number of rows. Thus the width of ranges varies, but the height remains about the same. Most DBMSs use equal-height histograms because the maximum and expected estimation errors can be controlled with a small number of ranges, typically 20 to 50.

Oracle 12c provides hybrid histograms, a variation of equal-height histograms to improve row count estimates involving popular values. A hybrid histogram stores row frequencies for all popular column values in addition to the normal information for equal-height histograms. A column value is popular if its row frequency is larger than the number of rows divided by the number of buckets in the histogram. A hybrid histogram provides improved row count estimates for conditions involving popular column values.

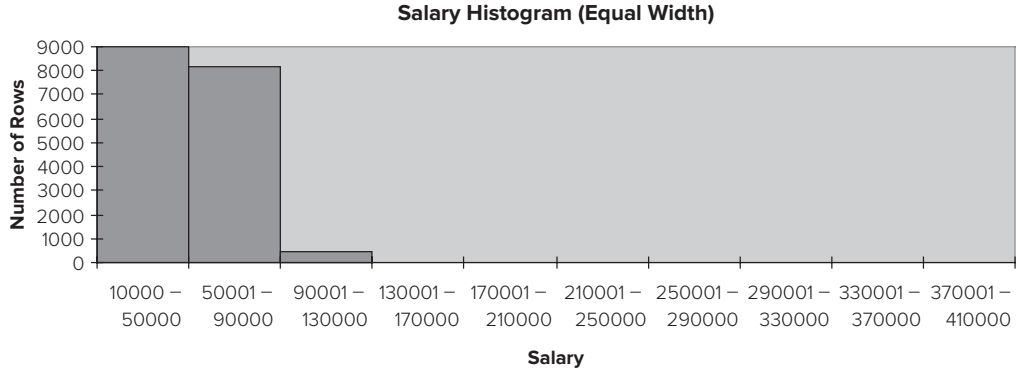


FIGURE 8.4
Example Equal-Width Histogram for the Salary Column

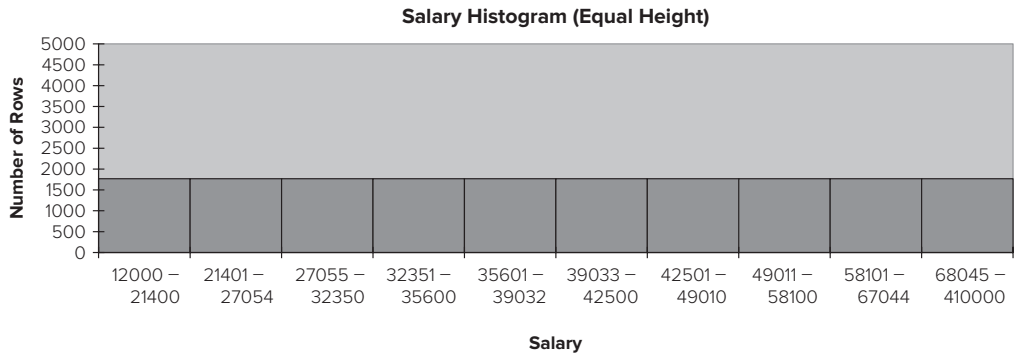


FIGURE 8.5
Example Equal-Height Histogram for the Salary Column

An optimizing SQL compiler uses table profiles to estimate the combined measure of performance presented in Section 8.1.2. For example, the number of physical records is used to calculate the physical record accesses to retrieve all rows of a table. The distribution of column values is needed to estimate the fraction of rows that satisfy a condition in a query. For example, to estimate the fraction of rows that satisfy the condition, `Salary > 45000`, you would sum the number of rows in the last three bars of Figure 8.5 and use linear interpolation in the seventh bar.

It is sometimes useful to store more detailed data about columns. If columns are related, errors can be made when estimating the fraction of rows that satisfy conditions connected by logical operators. For example, if the salary and age columns are related, the fraction of rows satisfying the logical expression, `Salary > 45000 AND Age < 25,` cannot be accurately estimated by knowing the distribution of salary and age alone. Data about the statistical relationship between salary and age are also necessary. Because summaries about column relationships are important for determination of efficient retrieval plans, some enterprise DBMSs provide tools to determine important groups of related columns and efficiently store relationships among them.

Another Histogram Example The histograms shown in Figures 8.4 and 8.5 demonstrate skewed salary data for a typical moderate-sized business or government agency. The existence of a few large salaries (such as CEO and agency director) makes traditional histograms a poor fit. Most enterprise DBMSs do not provide traditional (equal-width) histograms because of estimation errors caused by extreme values. Instead, most enterprise DBMSs provide a choice between an equal-height histogram and the uniform value assumption. In most cases, the uniform value assumption using minimum and maximum values provides even worse estimates than an equal-width histogram as the uniform value assumption is a traditional histogram with only one range.

Equal-height histograms work well on symmetric data as shown in Figures 8.6 and 8.7. The histograms were created using the Microsoft Excel histogram tool using a data set of the highest average salaries of retirees in the Denver (Colorado) Public Schools from 2001 to 2006. Highest average salaries (over the last three years of employment in the Denver Public Schools) are important determinants for pension benefits. The equal-width histogram in Figure 8.6 is reasonably symmetric because very high paid employees do not participate in the pension plan. Because of the lack of skew, the equal-width histogram would provide better row fraction estimates for some queries than the equal-height histogram shown in Figure 8.7. The ranges in the equal-height histogram were determined by the GATHER_TABLE_STATS procedure in the Oracle DBMS_STATS package. However, the performance of the equal-height histogram can be easily improved by doubling the ranges with no noticeable performance overhead.

FIGURE 8.6
Equal-Width Histogram for the Highest Average Salary Column

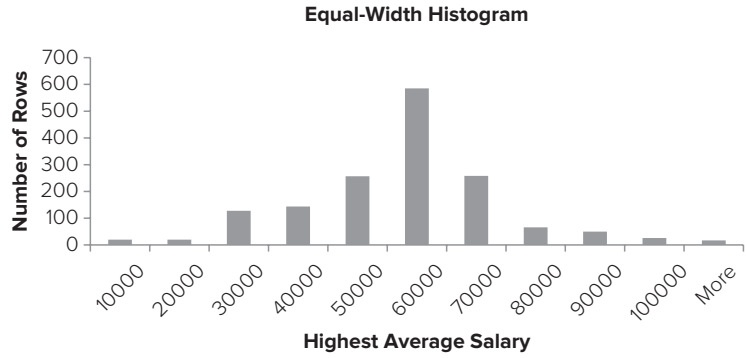
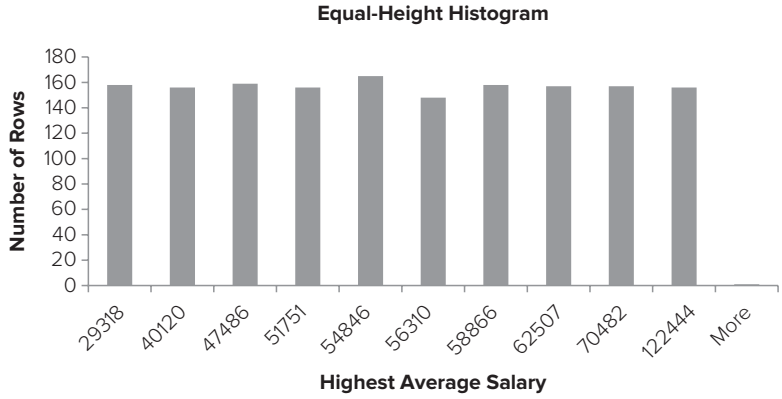


FIGURE 8.7
Equal-Height Histogram for the Highest Average Salary Column



8.2.2 Application Profiles

Application profiles summarize the queries, forms, reports, and web pages that access a database as shown in Table 8-3. For forms, the frequency of using the form for each kind of operation (insert, update, delete, and retrieval) should be specified. For queries, reports and web pages, the distribution of parameter values encodes the number of times the query/report is executed with various parameter values. For all application objects, execution statistics indicate the computing resources consumed by executing SQL statements associated with applications. Execution statistics cover disk access, CPU usage, communication system usage, rows impacted, and elapsed time. Enterprise DBMSs typically provide tools to collect data about SQL statements executed by applications. For example, Oracle provides the Automatic Workload Repository to store data about SQL statements. The major limitation of application profile tools provided by enterprise DBMSs is the lack of connection between applications and associated SQL statements.

Table 8-4 depicts profiles for several applications of the university database. The frequency data are specified as an average per unit time period such as per day. Sometimes it is useful to summarize frequencies in more detail. Specifying peak frequencies and variance in frequencies can help avoid problems with peak usage. In addition, importance of applications can be specified as response time limits so that physical designs are biased towards critical applications.

TABLE 8-3
Typical Components of an Application Profile

Application Type	Statistics
Query	Frequency, distribution of parameter values, execution statistics
Form	Frequency of insert, update, delete, and retrieval operations, execution statistics
Report	Frequency, distribution of parameter values, execution statistics
Web page	Frequency, distribution of parameter values, execution statistics

TABLE 8-4
Example Application Profiles

Application Name	Tables	Operation	Frequency
Enrollment Query	*Course, Offering, Enrollment*	Retrieval	100 per day during the registration period; 50 per day during the drop/add period
Registration Form	*Registration*	Insert	1,000 per day during the registration period
Registration Form	*Enrollment*	Insert	5,000 per day during the registration period; 1,000 per day during drop/add period
Registration Form	*Registration*	Delete	100 per day during the registration period; 10 per day during the drop/add period
Registration Form	*Enrollment*	Delete	1,000 per day during the registration period; 500 per day during the drop/add period
Registration Form	*Registration, Student*	Retrieval	6,000 per day during the registration period; 1,500 per day during the drop/add period
Registration Form	*Enrollment, Course, Offering, Faculty*	Retrieval	6,000 per day during the registration period; 1,500 per day during the drop/add period
Faculty Workload Report	*Faculty, Course, Offering, Enrollment*	Retrieval	50 per day during the last week of the academic period; 10 per day otherwise; typical parameters: current year and academic period

8.3 FILE STRUCTURES

As mentioned in Section 8.1, selecting among alternative file structures is one of the most important choices in physical database design. In order to choose intelligently, you must understand characteristics of available file structures. This section describes the characteristics of common file structures available in most DBMSs.

8.3.1 Sequential Files

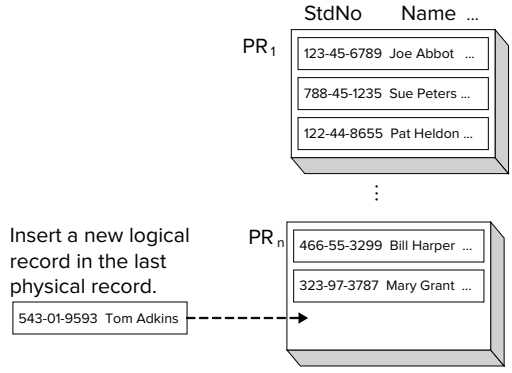
Sequential File
a simple file organization in which records are stored in insertion order or by key value. Sequential files are simple to maintain and provide good performance for processing large numbers of records.

The simplest kind of file structure stores logical records in insertion order. New logical records are appended to the last physical record in the file, as shown in Figure 8.8. Unless logical records are inserted in a particular order and no deletions are made, the file becomes unordered. Unordered files are sometimes known as heap files because of the lack of order.

The primary advantage of unordered sequential files is fast insertion. However, when logical records are deleted, insertion becomes more complicated. For example, if the second logical record in PR_1 is deleted, space is available in PR_1. A list of free space must be maintained to tell if a new record can be inserted into the empty space instead of into the last physical record. Alternately, new logical records can always be inserted in the last physical record. However, periodic reorganization to reclaim lost space due to deletions is necessary.

Because ordered retrieval is sometimes needed, ordered sequential files can be preferable to unordered sequential files. Logical records are arranged in key order where the key can be any column, although it is often the primary key. Ordered sequential files are faster when retrieving in key order, either the entire file or a subset of records. The primary disadvantage to ordered sequential files is slow insertion speed. Figure 8.9 demonstrates that records must sometimes be rearranged during the insertion process. The rearrangement process can involve movement of logical records between blocks and maintenance of an ordered list of physical records.

FIGURE 8.8
Inserting a New Logical Record into an Unordered Sequential File



8.3.2 Hash Files

Hash File
a specialized file structure that supports search by key. Hash files transform a key value into an address to provide fast access.

Hash files, in contrast to sequential files, support fast access of records by primary key value. The basic idea behind hash files is a function that converts a key value into a physical record address. The mod function (remainder division) is a simple hash function. Table 8-5 applies the mod function to the *StdNo* column values in Figure 8.8. For simplicity, assume that the file capacity is 100 physical records. The divisor for the mod function is 97, a large prime number close to the file capacity. The physical record number is the result of the hash function result plus the starting physical record number, assumed to be 150. Figure 8.10 shows selected physical records of the hash file.

Hash functions may assign more than one key to the same physical record address. A collision occurs when two keys hash to the same physical record address. As long as the physical record has free space, a collision is no problem. However, if the original or home physical record is full, a collision-handling procedure locates a physical record with free space. Figure 8.11 demonstrates the linear probe procedure for collision

FIGURE 8.9
Inserting a New Logical Record into an Ordered Sequential File

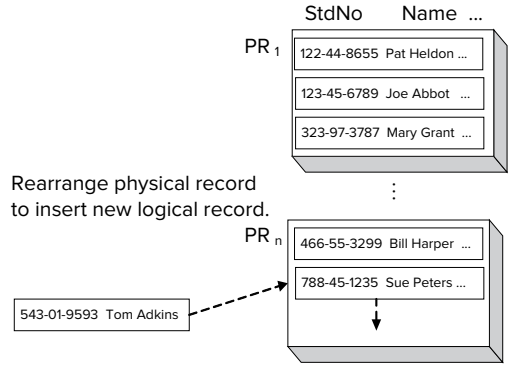


TABLE 8-5
Hash Function Calculations for *StdNo* Values

StdNo	StdNo Mod 97	PR Number
122448655	26	176
123456789	39	189
323973787	92	242
466553299	80	230
788451235	24	174
543019593	13	163

FIGURE 8.10
Hash File after Insertions

handling. In the linear probe procedure, a logical record is placed in the next available physical record if its home address is occupied. To retrieve a record by its key, the home address is initially searched. If the record is not found in its home address, a linear probe is initiated.

The existence of collisions highlights a potential problem with hash files. If collisions do not occur often, insertions and retrievals are very fast. If collisions occur often, insertions and retrievals can be slow. The likelihood of a collision depends on the remaining capacity in the hash file. Generally, if a file is less than 70 percent full, collisions do not occur often. Using a hash file that grows beyond 70 percent full can seriously degrade performance for both retrievals and insertions. If the hash file becomes too full, reorganization is necessary. Reorganization can be time-consuming and disruptive because a larger hash file is allocated and all logical records are inserted into the new file.

To eliminate periodic reorganizations, dynamic hash files have been proposed. In a dynamic hash file, periodic reorganization is never necessary and search performance does not degrade after many insert operations. However, the average number of physical record accesses to retrieve a record may be slightly higher as compared to a static hash file that is not too full. The basic idea in dynamic hashing is that the size

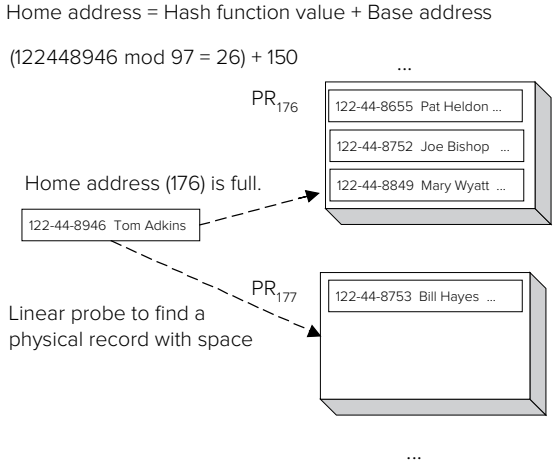


FIGURE 8.11
Linear Probe Collision Handling during an Insert Operation

of the hash file grows as records are inserted. For details of the various approaches, consult the references at the end of this chapter.

Another problem with hash files is sequential search. Good hash functions tend to spread logical records uniformly among physical records. Because of gaps between physical records, sequential search may examine empty physical records. For example, to search the hash file depicted in Figure 8.10, 100 physical records must be examined even though only six contain data. Even if a hash file is reasonably full, logical records are spread among more physical records than in a sequential file. Thus, when performing a sequential search, the number of physical record accesses may be higher in a hash file than in a sequential file.

8.3.3 Multiway Tree (Btrees) Files

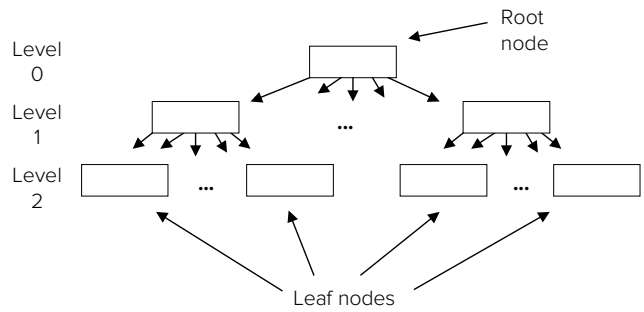
Sequential files and hash files provide good performance on some operations but poor performance on other operations. Sequential files perform well on sequential search but poorly on key search. Hash files perform well on key search but poorly on sequential search. The multiway tree or **Btree** as it is popularly known, is a compromise and widely used file structure. The Btree provides good performance on both sequential search and key search. This section describes characteristics of the Btree, shows examples of Btree operations, and discusses the cost of operations.

Btree File
a popular file structure supported by most DBMSs because it performs well on key search as well as sequential search. A Btree file is a balanced, multiway tree.

Btree Characteristics: What's in a Name? A Btree is a special kind of tree as depicted in Figure 8.12. A tree is a structure in which each node has at most one parent except for the root or top node. The Btree structure possesses a number of characteristics, discussed in the following list, that make it a useful file structure. Some of the characteristics are possible meanings for the letter *B*[1] in the name.

- Balanced: all leaf nodes (nodes without children) reside on the same level of the tree. In Figure 8.12, all leaf nodes are two levels beneath the root. A balanced tree ensures that all leaf nodes can be retrieved with the same access cost.
- Bushy: the number of branches from a node is large, perhaps 50 to 200 branches. Multiway, meaning more than two, is a synonym for bushy. The width (number of arrows from a node) and height (number of nodes between root and leaf nodes) are inversely related: increase width, decrease height. The ideal Btree is wide (bushy) but short (few levels).
- Block-Oriented: each node in a Btree is a block. To search a Btree, you start in the root node and follow a path to a leaf node containing data of interest. The height of a Btree is important because it determines the number of physical record accesses for searching.
- Dynamic: the shape of a Btree changes as logical records are inserted and deleted. Periodic reorganization is never necessary for a Btree. The next

FIGURE 8.12
Structure of a Btree of Height 3



[1] Another possible meaning for the letter *B* is Bayer, for the inventor of the Btree, Professor Rudolph Bayer. In a private conversation, Professor Bayer denied naming the Btree after himself or for his employer at the time, Boeing. When pressed, Professor Bayer only said that the *B* represents the *B*.

subsection describes node splitting and concatenation, changes to a Btree as records are inserted and deleted.

- Ubiquitous: the Btree is a widely implemented and used file structure.

Before studying the dynamic nature, let us look more carefully at the contents of a node as depicted in Figure 8.13. Each node consists of pairs with a key value and a pointer (physical record address), sorted by key value. The pointer identifies the physical record that contains the logical record with the key value. Other data in a logical record, besides the key, do not usually reside in the nodes. The other data may be stored in separate physical records or in the leaf nodes.

An important property of a Btree is that each node, except the root, must be at least half full. The physical record size, the key size, and the pointer size determine node capacity. For example, if the physical record size is 4,096 bytes, the key size is 8 bytes, and the pointer size is 8 bytes, the maximum capacity of a node is 256 <key, pointer> pairs. Thus, each node must contain at least 128 pairs. Because the designer usually does not have control over the physical record size and the pointer size, the key size determines the number of branches. Btrees are usually not good for large key sizes due to less branching per node and, hence, taller and less-efficient Btrees.

Node Splitting and Concatenation Insertions are handled by placing a new key in a nonfull node or by splitting nodes, as depicted in Figure 8.14. In the partial Btree in Figure 8.14(a), each node contains a maximum of four keys. Inserting the key value 55 in Figure 8.14(b) requires rearrangement in the right-most leaf node. Inserting the key value 58 in Figure 8.14(c) requires more work because the right-most leaf node is full. To accommodate the new value, the node is split into two nodes and a key value is moved to the root node. In Figure 8.14(d), a split occurs at two levels because both nodes are full. When a split occurs at the root, the tree grows another level.

Deletions are handled by removing the deleted key from a node and repairing the structure if needed, as demonstrated in Figure 8.15. If the node is still at least half full, no additional action is necessary as shown in Figure 8.15(b). However, if the node is less than half full, the structure must be changed. If a neighboring node contains more than half capacity, a key can be borrowed, as shown in Figure 8.15(c). If a key cannot be borrowed, nodes must be concatenated, as shown in Figure 8.15(d).

Cost of Operations The height of a Btree is small even for a large table when the branching factor is large. An upper bound or limit on the height (h) of a Btree is

$h \leq ceil(log_d\ (n+1)/2)$ where
ceil is the ceiling function ($ceil(x)$ is the smallest integer $\geq x$).
d is the minimum number of keys in a node.
n is the number of keys to store in the index.
log_d is the *log* function with base d. The *log* function returns the exponent, $log_d(x) = y$ meaning that $d^y = x$. For example, $log_{10}\ (100) = 2$ meaning that $10^2 = 100$.
Example: $h \leq 4$ for $n = 1{,}000{,}000$ and $d = 42$.

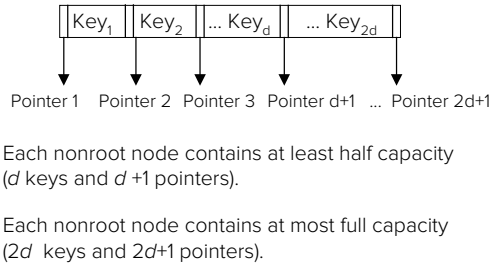
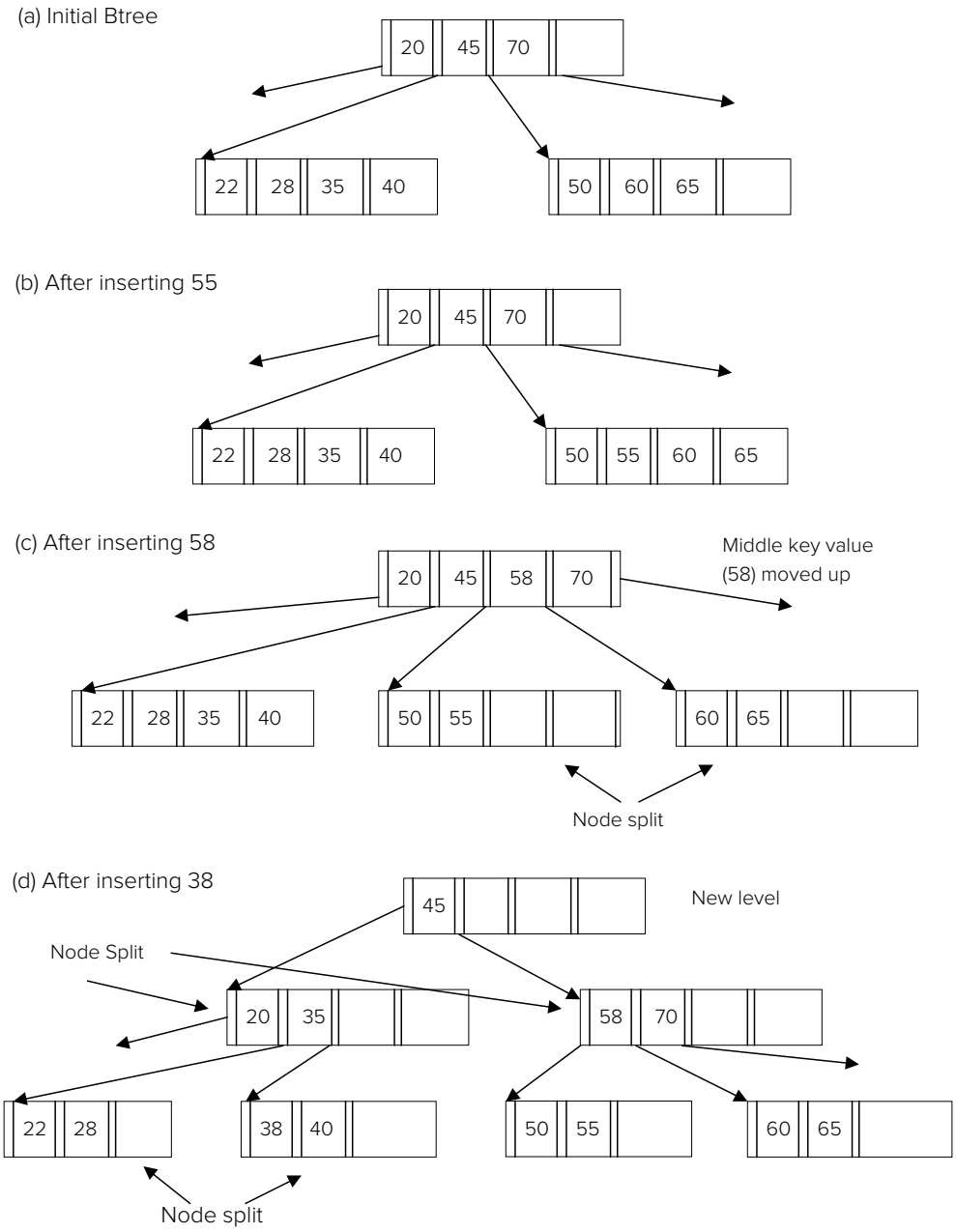


FIGURE 8.13
Btree Node Containing Keys and Pointers

FIGURE 8.14
Btree Insertion Examples



The height dominates the number of physical record accesses in Btree operations. The cost in terms of physical record accesses to find a key is less than or equal to the height. If the row data are not stored in the tree, another physical record access is necessary to retrieve the row data after finding the key. Btrees have logarithmic search cost because the log function dominates the formula for height. Bushy Btrees with a large number of keys in a node can be efficiently searched.

The cost to insert a key includes the cost to locate the nearest key plus the cost to change nodes. In the best case as demonstrated in Figure 8.14(b), the additional cost is one physical record access to change the index record and one physical record access to write the row data. The worst case occurs when a new level is added to the tree, as depicted in Figure 8.14(d). Even in the worst case, the height of the tree still dominates. Another $2h$ write operations are necessary to split a tree at each level.

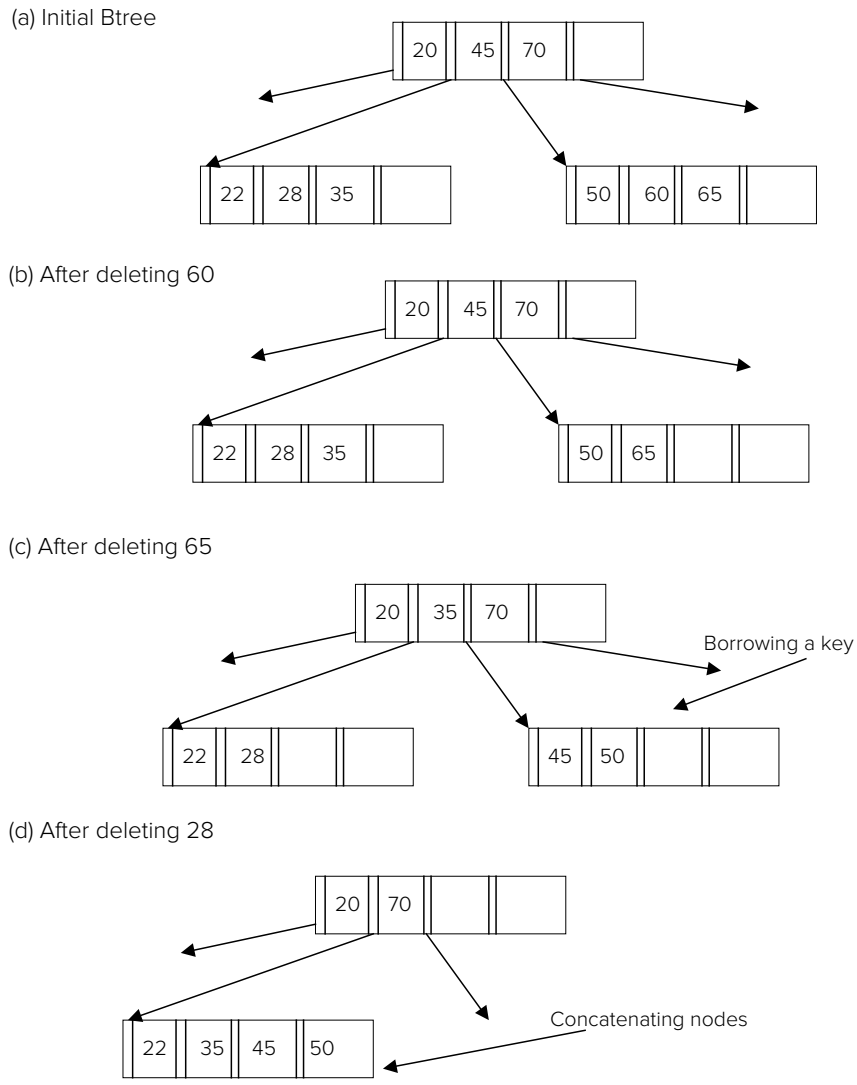


FIGURE 8.15
Btree Deletion Examples

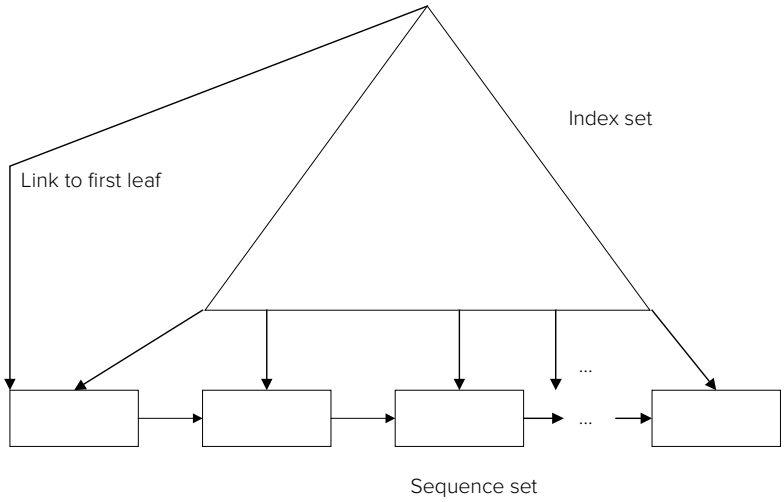
B+tree Sequential range searches can be a problem with Btrees. To perform a range search, the search procedure must travel up and down a tree. For example, to retrieve keys in the range 28 to 60 in Figure 8.15(a), the search process starts in the root, descends to the left leaf node, returns to the root, and then descends to the right leaf node. This procedure has problems with retention of physical records in memory. Operating systems may replace physical records if there have not been recent accesses. Because some time may elapse before a parent node is accessed again, the operating system may replace it with another physical record if main memory becomes full. Thus, another physical record access may be necessary when the parent node is accessed again.

To ensure that physical records are not replaced, the B+tree variation is usually implemented. Figure 8.16 shows the two parts of a **B+tree**. The triangle (index set) represents a normal Btree index. The lower part (sequence set) contains the leaf nodes. All keys reside in the leaf nodes even if a key appears in the index set. The leaf nodes are connected together so that sequential searches do not need to move up the tree. After the initial key is found, the search process accesses only nodes in the sequence set.

B+tree File
the most popular variation of the Btree. In a B+tree, all keys are redundantly stored in the leaf nodes. The B+tree provides improved performance on sequential and range searches.

Index Matching A Btree can be used to store all data in the nodes (primary file structure) or just pointers to the data records (secondary file structure or index).

FIGURE 8.16
B+tree Structure



Index Matching
determining if an index can be used on a search condition in a query. The determination uses matching rules that depend on the comparison operation and column(s) in an index. Composite indexes with multiple columns are more restrictive for index usage.

A Btree is especially versatile as an index because it can be used for a variety of queries. Determining whether an index can be used in a query is known as **index matching**. When a condition in a WHERE clause references an indexed column, the DBMS must determine if the index can be used. The complexity of a condition determines whether an index can be used. For single-column indexes, an index matches a condition if the column appears alone without functions or operators and the comparison operator matches one of the following items:

- =, >, <, >=, <= (but not <>)
- BETWEEN
- IS NULL
- IN <list of constant values>
- LIKE 'Pattern' in which pattern does not contain a meta character (%, _) as the first part of the pattern

For composite indexes involving more than one column, the matching rules are more complex and restrictive. Composite indexes are ordered by the most significant (first column in the index) to the least significant (last column in the index) column. A composite index matches conditions according to the following rules:

- The first column of the index must have a matching condition.
- Columns match from left (most significant) to right (least significant). Matching stops when the next column in the index is not matched.
- Only the first BETWEEN condition matches.
- At most, one IN condition matches an index column. Matching stops after the next matching condition. The second matching condition cannot be IN or BETWEEN.

To depict index matching, Table 8-6 shows examples of matching between indexes and conditions. When matching a composite index, the conditions can be in any order. Because of the restrictive matching rules, composite indexes should be used with caution. It is usually a better idea to create indexes on the individual columns as most DBMSs can combine the results of multiple indexes when answering a query.

8.3.4 Bitmap Indexes

Btree and hash files work best for columns with unique values. For nonunique columns, Btrees index nodes can store a list of row identifiers instead of an individual row identifier for unique columns. However, if a column has few values, the list of row identifiers can be very long.

TABLE 8-6
Index Matching Examples

Condition	Index	Matching Notes
C1 = 10	C1	Matches index on C1
C2 BETWEEN 10 AND 20	C2	Matches index on C2
C3 IN (10, 20)	C3	Matches index on C3
C1 <> 10	C1	Does not match index on C1
C4 LIKE 'A%'	C4	Matches index on C4
C4 LIKE '%A'	C4	Does not match index on C4
C1 = 10 AND C2 = 5 AND C3 = 20 AND C4 = 25	(C1,C2,C3,C4)	Matches all columns of the index
C2 = 5 AND C3 = 20 AND C1 = 10	(C1,C2,C3,C4)	Matches the first three columns of the index
C2 = 5 AND C4 = 22 AND C1 = 10 AND C6 = 35	(C1,C2,C3,C4)	Matches the first two columns of the index
C2 = 5 AND C3 = 20 AND C4 = 25	(C1,C2,C3,C4)	Does not match any columns of the index: missing condition on C1
C1 IN (6, 8, 10) AND C2 = 5 AND C3 IN (20, 30, 40)	(C1,C2,C3,C4)	Matches the first two columns of the index: at most one matching IN condition
C2 = 5 AND C1 BETWEEN 6 AND 10	(C1,C2,C3,C4)	Matches the first column of the index: matching stops after the BETWEEN condition

As an alternative structure for columns with few values, many DBMSs support bitmap indexes. Figure 8.17 depicts a bitmap column index for a sample *Faculty* table. A bitmap contains a string of bits (0 or 1 values) with one bit for each row of a table. In Figure 8.17, the length of the bitmap is 12 positions because there are 12 rows in the

FIGURE 8.17
Sample Faculty Table and Bitmap Column Index on FacRank

Faculty Table

RowId	FacNo	...	FacRank
1	098-55-1234		Asst
2	123-45-6789		Asst
3	456-89-1243		Assc
4	111-09-0245		Prof
5	931-99-2034		Asst
6	998-00-1245		Prof
7	287-44-3341		Assc
8	230-21-9432		Asst
9	321-44-5588		Prof
10	443-22-3356		Assc
11	559-87-3211		Prof
12	220-44-5688		Asst

Bitmap Column Index on FacRank

FacRank	Bitmap
Asst	110010010001
Assc	001000100100
Prof	000101001010

sample *Faculty* table. A record of a bitmap column index contains a column value and a bitmap. A 0 value in a bitmap indicates that the associated row does not have the column value. A 1 value indicates that the associated row has the column value. The DBMS provides an efficient way to convert a position in a bitmap to a row identifier.

A variation of the bitmap column index is the bitmap join index. In a bitmap join index, the bitmap identifies rows of a related table, not the table containing the indexed column. Thus, a bitmap join index represents a precomputed join from a column in a parent table to the rows of a child table that join with rows of the parent table.

A join bitmap index can be defined for a join column such as *FacNo* or a nonjoin column such as *FacRank*. Figure 8.18 depicts a bitmap join index for the *FacRank* column in the *Faculty* table to the rows in the sample *Offering* table. The length of the bitmap is 24 bits because there are 24 rows in the sample *Offering* table. A 1 value in a bitmap indicates that a parent row containing the column value joins with the child table in the specified bit position. For example, a 1 in the first bit position of the "Asst"

FIGURE 8.18
Sample Offering Table and Bitmap Join Index on *FacRank*

Offering Table

RowId	OfferNo	...	FacNo
1	1111		098-55-1234
2	1234		123-45-6789
3	1345		456-89-1243
4	1599		111-09-0245
5	1807		931-99-2034
6	1944		998-00-1245
7	2100		287-44-3341
8	2200		230-21-9432
9	2301		321-44-5588
10	2487		443-22-3356
11	2500		559-87-3211
12	2600		220-44-5688
13	2703		098-55-1234
14	2801		123-45-6789
15	2944		456-89-1243
16	3100		111-09-0245
17	3200		931-99-2034
18	3258		998-00-1245
19	3302		287-44-3341
20	3901		230-21-9432
21	4001		321-44-5588
22	4205		443-22-3356
23	4301		559-87-3211
24	4455		220-44-5688

Bitmap Join Index on FacRank

FacRank	Bitmap
Asst	110010010001110010010001
Assc	001000100100001000100100
Prof	000101001010000101001010

row of the join index means that a *Faculty* row with the "Asst" value joins with the first row of the *Offering* table.

Bitmap indexes work well for stable columns with few values. The *FacRank* column would be attractive for a bitmap column index because it contains few values and faculty members do not change rank often. The size of a bitmap is not an important issue because compression techniques can reduce the size significantly. Due to the requirement for stable columns, bitmap indexes are most common for data warehouse tables especially as join indexes. A data warehouse is a business intelligence database that is mostly used for retrievals and periodic insertion of new data. Chapter 15 discusses the use of bitmap indexes for data warehouse tables.

Bitmap Index
a secondary file structure consisting of a column value and a bitmap. A bitmap contains one bit position for each row of a referenced table. A bitmap column index references the rows containing the column value. A bitmap join index references the rows of a child table that join with rows of the parent table containing the column value. Bitmap indexes work well for stable columns with few values, typical of tables in a data warehouse.

8.3.5 Columnstore Indexes

Storage structures presented in section 8.3 (hash, btree, and bitmap) focus on specific types of queries. A hash structure supports queries by key value. A btree supports queries by key value and range of key values. A bitmap index supports queries on a column with relatively few values as well as star join queries combining a child table with multiple parent tables. Queries that involve grouping and aggregate calculations do not have a specialized storage structure. These type of queries support business intelligence applications for medium and long-term decision making in an organization. Part 6 describes data warehouse processing to support business intelligence requirements with Chapter 15 focusing on extended query support.

Columnstore indexes support queries for business intelligence often involving tables with millions of rows. Without columnstore capabilities, full table scans using sequential file structures are often necessary. The columnstore index provides a specialized storage structure with sharply reduced computing resources, leading to much faster response times.

Columnstore
a file organization reversing the traditional row store approach. A columnstore places column values in physical records. The values of each column for a large group of rows are stored together with each column in a separate file. A columnstore index stores all or a subset of columns of a table in a compressed manner, optionally with sorting of column values.

The columnstore approach reverses the basic method of storing data. The traditional storage approach, known as a row store, places entire rows in physical records as described in Section 8.1. In contrast, a columnstore places columns in physical records as depicted in Figure 8.19. The values of each column for a group of rows (typically more than a hundred thousand rows) are stored together with each column in a separate file. Figure 8.19 shows *k* column files for row group *i*. Each column file contains physical records (separated by dashed lines) storing column values. Each column may have a different number of values with varying space requirements.

To reduce space requirements, a columnstore index stores all or a subset of columns of a table in a compressed manner. To support grouping queries, column values can be optionally sorted. Figure 8.20 depicts compression and sorting of column values in a columnstore index for a sample row group of the *Faculty* table. Figure 8.20a shows a traditional row store for a subset of *Faculty* rows. Figures 8.20b to 8.20d depict a columnstore index for the *Faculty* row group. Compression occurs for duplicate column values, indicated with the multiplication symbol. Values in each columnstore file are sorted in ascending order.

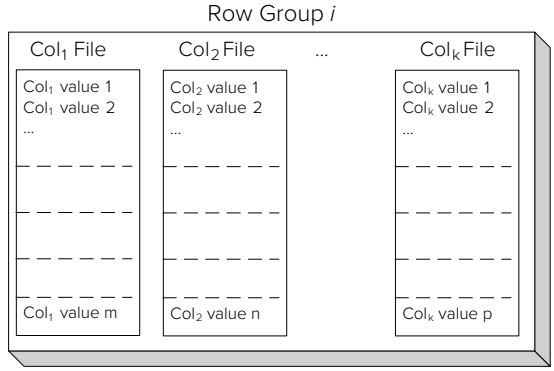


FIGURE 8.19
Columnstore Files for Columns in a Row Group of a Table

FIGURE 8.20
Columnstore Index for Sample Row Group

a) Sample Row Group of the Faculty Table

FacNo	FacRank	FacSalary
1111	Assc	120000
1122	Asst	90000
1133	Asst	85000
1144	Prof	140000
1155	Assc	115000
1166	Prof	140000

b) FacNo Columnstore for Sample Group of the Faculty Table

FacNo
1111
1122
1133
1144
1155
1166

c) FacRank Columnstore for Sample Group of the Faculty Table

FacRank
Assc × 2
Asst × 2
Prof × 2

d) FacSalary Columnstore for Sample Group of the Faculty Table

FacSalary
85000
90000
85000
115000
140000 × 2

Columnstore indexes provide substantial performance improvements over traditional row storage for grouping queries performing calculations on relatively few columns on large tables. A columnstore index maximizes main memory usage with retrieval of only necessary columns in a highly compressed format. Performance experiments have demonstrated 10 times performance improvements for columnstore indexes over sequential files. Due to high potential performance improvements, columnstore indexes have been widely implemented in relational DBMSs (both commercial and open source) as well as NoSQL DBMSs. However, columnstore indexes are highly specialized, appropriate only for grouping queries accessing large tables.

8.3.6 Summary of File Structures

To help recall file structures, Table 8-7 summarizes major characteristics of each structure. In the first row, hash files can be used for sequential access, but there may be extra physical records because keys are evenly spread among physical records. In the second row, unordered and ordered sequential files must examine on average half the physical records (linear). Hash files examine a constant number (usually close to 1) of physical records, assuming that the file is not too full. Btrees have logarithmic search costs because of the relationship between the height, the *log* function, and search cost formulas. File structures can store all the data of a table (primary file structure) or store only key data along with pointers to the data records (secondary file structure). A secondary file structure or index provides an alternative path to the data. A bitmap index supports range searches by performing union operations on the bitmaps for each column value in a range. Although a columnstore supports sequential search, its only practical usage is grouping queries on large tables.

8.3.7 Oracle Storage Concepts and File Structures

To supplement conceptual presentation of storage concepts, the last part of this section presents selected details about Oracle storage concepts and file structures. Oracle supports a rich collection of file structures to achieve performance objectives.

TABLE 8-7
Summary of File Structures

	Unordered	Ordered	Hash	B+tree	Bitmap	Columnstore
Sequential search	Y	Y	Extra PRs	Y	N	Y
Key search	Linear	Linear	Constant time	Logarithmic	Linear	N
Range search	N	Y	N	Y	Y	N
Grouping queries	Y	Y	N	Y	N	Y (best)
Usage	Primary only	Primary only	Primary or secondary	Primary or secondary	Secondary only	Primary or secondary

In Oracle, a database consists of a collection of tablespaces. In the simplest arrangement, a database contains a single tablespace stored in one file. For large databases, a database may contain multiple tablespaces with each tablespace stored on multiple files. A file is a collection of physical records managed by the operating system. Objects such as tables and indexes are stored in files as shown in Figure 8.21. A file is allocated in extents that are collections of contiguous disk blocks. Oracle recommends using uniform extent sizes in a file such as 1 MB.

Oracle has two important parameters to control free space in a data block. As depicted in Figure 8.22, the PCTFREE parameter indicates the minimum percentage of a data block preserved as free space for updates to rows. The PCTFREE parameter provides a high water mark as a block is marked as full for insertions when the PCTFREE limit is reached. After reaching the PCTFREE limit, the PCTUSED parameter indicates the minimum percentage of a block that must be available before new rows are added to the block. Thus, the PCTUSED parameter acts as a low water mark in that a block is marked full until it falls to its PCTUSED limit. The PCTFREE and PCTUSED parameters must sum to less than 100%. For example, PCTFREE of 20% indicates that new rows can be added until a block reaches 80% full. After this point, new rows cannot be added again until the remaining space falls to the PCTUSED value (40%) through deletions.

Oracle provides a variety of file structures to organize disk blocks. For primary storage, Oracle provides unordered, index-organized, and clustered files. An index-organized file is a Btree in which index nodes contain complete rows (key values and nonkey values). To ensure that an index-organized file is sufficiently bushy, Oracle uses overflow areas to store long rows separate from associated

FIGURE 8.21
Oracle Tablespace, Files, and Database Objects

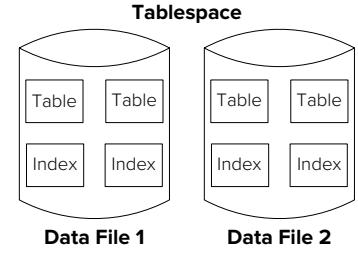
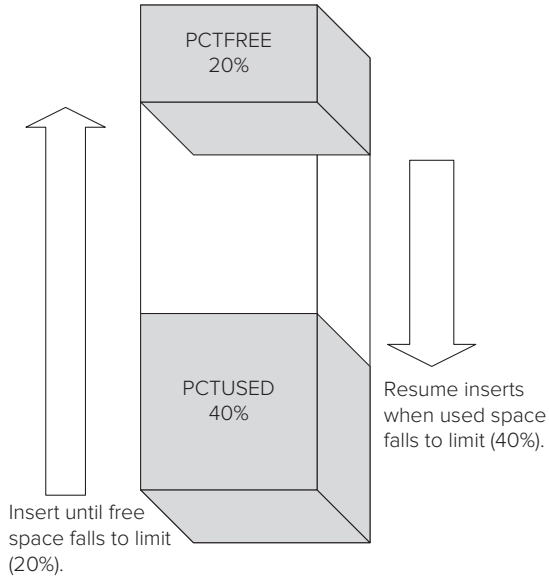


FIGURE 8.22
Relationship between PCTFREE and PCTUSED Parameters



index nodes. A clustered file contains rows from two or more tables. Clustering is a specialized structure for closely related tables such as order and order line in which parent and child rows are almost always accessed together. For secondary storage, Oracle provides B+tree, hash, bitmap, and function indexes. A function index uses a function to precompute a column value and store it in the index. A function index is useful for conditions involving a function on a column rather than the column alone.

To augment traditional row storage, Oracle supports an in-memory column store architecture. The architecture supports columnstore indexes along with dual-format memory management for both row and column data.

For large databases, Oracle provides a variety of partitioning options to improve performance and reliability. Partitioning can improve query performance by accessing a subset of partitions, rather than an entire table. Partitioning increases database availability if critical tables and indexes are divided into partitions to reduce maintenance windows, recovery times, and impact of failures. Partitioning should be done in conjunction with parallel database processing as discussed in Chapter 18. As listed below, Oracle supports various combinations of partitioning for tables and indexes.

- A non partitioned table can have partitioned or non-partitioned indexes.
- A partitioned table can have partitioned or non-partitioned indexes.
- A table can be partitioned by lists of values, range of values, or hash functions.

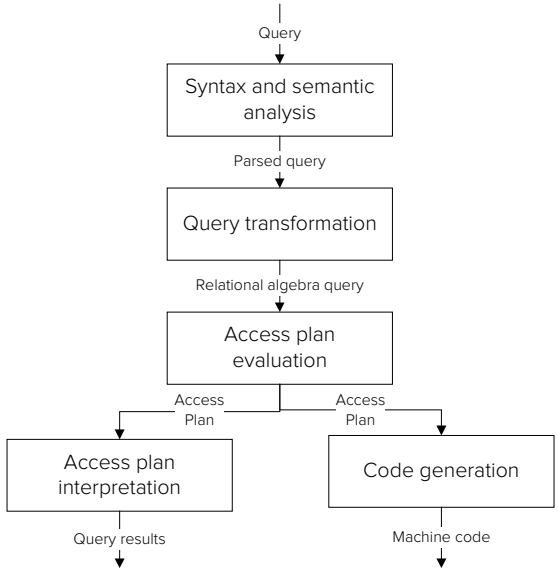
8.4 QUERY OPTIMIZATION

In most relational DBMSs, you do not have the ability to choose the implementation of queries on a physical database. The query optimization component assumes this responsibility. Your productivity increases because you do not need to make these tedious decisions. However, you can sometimes improve optimization decisions if you understand principles of the optimization process. To provide you with an understanding of the optimization process, this section describes the tasks performed and discusses tips to improve optimization decisions.

8.4.1 Translation Tasks

When you submit an SQL statement for execution, the query optimization component translates your query in four phases as shown in Figure 8.23. The first and fourth phases are common to any computer language translation process. The second

FIGURE 8.23
Tasks in Database Language Translation



phase has some unique aspects. The third phase is unique to translation of database languages.

Syntax and Semantic Analysis The first phase analyzes a query for syntax and simple semantic errors. Syntax errors involve misuse of keywords such as if the FROM keyword was misspelled in Example 8.1. Semantic errors involve misuse of columns and tables. A data language compiler can detect only simple semantic errors involving incompatible data types. For example, a WHERE condition that compares the *CourseNo* column with the *FacSalary* column results in a semantic error because these columns have incompatible data types. To find semantic errors, the database language translator uses table, column, and relationship definitions as stored in the data dictionary.

Example 8.1

Joining three tables (Oracle)

```
SELECT FacName, CourseNo, Enrollment.OfferNo, EnrGrade
  FROM Enrollment, Offering, Faculty
  WHERE CourseNo LIKE 'IS%' AND OffYear = 2018
    AND OffTerm = 'FALL'
    AND Enrollment.OfferNo = Offering.OfferNo
    AND Faculty.FacNo = Offering.FacNo
```

Query Transformation The second phase transforms a query into a simplified and standardized format. As with optimizing programming language compilers, database language translators can eliminate redundant parts of a logical expression. For example, the logical expression `(OffYear = 2017 AND OffTerm = 'WINTER') OR (OffYear = 2017 AND OffTerm = 'SPRING')` can be simplified to `OffYear = 2017 AND (OffTerm = 'WINTER' OR OffTerm = 'SPRING')`. Join simplification is unique to database languages. For example, if Example 8.1 contained a join with the *Student* table, this table could be eliminated if no columns or conditions involving the *Student* table are used in the query.

The standardized format is usually based on relational algebra. The relational algebra operations are rearranged so that the query can be executed faster. Typical rearrangement operations are described below. Because the query optimization component performs this rearrangement, you do not have to be careful about writing your query in an efficient way.

- Restriction operations are combined so that they can be tested togethr.
- Projection and restriction operations are moved before join operations to eliminate unneeded columns and rows before resource-intensive join operations.
- Cross product operations are transformed into join operations if a join condition exists in the WHERE clause.

Access Plan Evaluation The third phase generates an access plan to implement the rearranged relational algebra query. An **access plan** indicates the implementation of a query as operations on files, as depicted in Figures 8.24 and 8.25. In an access plan, the leaf nodes are individual tables in the query, and the arrows point upward to indicate the flow of data. The nodes above the leaf nodes indicate decisions about accessing individual tables. In Figure 8.24, Btree indexes are used to access individual tables. The first join combines the *Enrollment* and the *Offering* tables. The Btree file structures provide the sorting needed for the merge join algorithm. The second join combines the result of the first join with the *Faculty* table. The intermediate result must be sorted on

Access Plan
a tree that encodes decisions about file structures to access individual tables, the order of joining tables, and the algorithm to join tables.

FIGURE 8.24
Access Plan for Example 8.1

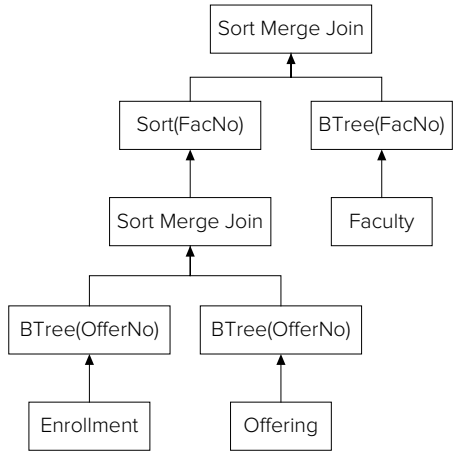
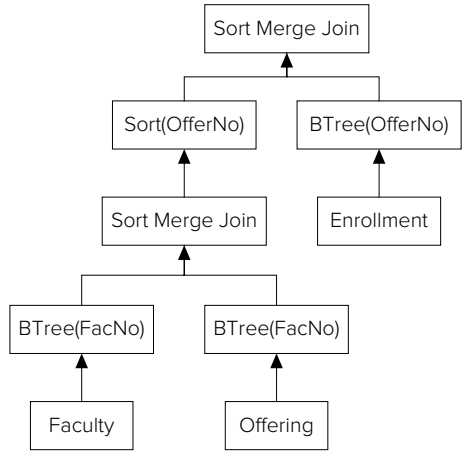


FIGURE 8.25
Alternative Access Plan for Example 8.1



FacNo before the merge join algorithm can be used. Figure 8.25 shows a variation of the access plan in Figure 8.24 in which the join order is changed.

Access plans vary by join orders, file structures, and join algorithms. For join order, the optimization component only considers feasible join orders in which the next join shares at least one table with the previous joins in a join order. For example, if a query involves four tables (T1 to T4) with three joins (T1-T2, T2-T3, and T3-T4), some feasible join orders are <T1-T2, T2-T3, T3-T4> and <T3-T4, T2-T3, T1-T2>. The join order < T1-T2, T3-T4, T2-T3> is not feasible because the first two joins do not share a common table[2]. For file structures, the optimization component considers primary and secondary file structures. For secondary file structures, a WHERE condition must match an index. Some optimization components can consider set operations (intersection for conditions connected by AND and union for conditions connected by OR) to combine the results of multiple indexes on the same table.

Most optimization components use a small set of join algorithms. Table 8-8 summarizes common join algorithms employed by optimization components. For each join operation in a query, the optimization component considers each supported join algorithm. For the nested loops and the hybrid algorithms, the optimization component must also choose the outer table and the inner table. All algorithms except the star join involve two tables at a time. The star join can combine any number of tables matching the star pattern (a child table surrounded by parent tables in 1-M relationships).

[2] This join order is feasible if joins are performed in parallel rather than pipelining. If an optimizer performs parallel joins at the table level, this join order is feasible. Parallelism is usually done at the partition level, not table level.

TABLE 8-8
Summary of Common Join Algorithms

Algorithm	Requirements	When to Use
Nested loops	Choose outer table and inner table; can be used for all joins	Appropriate when restrictive conditions on the outer table with an index on the join column of the inner table or when all pages of the inner table fit into memory
Sort merge	Both tables must be sorted (or use an index) on the join columns; only used for equi-joins	Appropriate if sort cost is small or if a clustered join index exists on the join columns of both tables
Hybrid join	Combination of sort merge and nested loops; outer table must be sorted (or use a join column index); inner table must have an index on the join column; only used for equi-joins	Performs better than sort merge when there is a nonclustering index (see the next section) on the join column of the inner table
Hash join	Internal hash file built for both tables; only used for equi-joins	Hash join performs better than sort merge when the tables are not sorted or clustered indexes do not exist
Star join	Join multiple tables in which there is one child table related to multiple parent tables in 1-M relationships; bitmap join index required on each parent table; only used for equi-joins	Useful for tables matching the star pattern with bitmap join indexes especially when the query has highly selective conditions on the parent tables; widely used to optimize data warehouse queries (see Chapter 15)

The nested loops algorithm can be used with any join operation, not just an equi-join operation.

The query optimization component uses cost formulas to evaluate access plans. Each operation in an access plan has a corresponding cost formula that estimates the physical record accesses and CPU operations. The cost formulas use table profiles to estimate the number of rows in a result. For example, the number of rows resulting from a WHERE condition can be estimated using distribution data such as a histogram. The query optimization component chooses the access plan with the lowest cost.

The query optimization component evaluates a large number of access plans. The number of join orders is the dominant element in determining the number of access plans. In most queries, the number of joins is one less than the number of tables in the query. In this situation, an upper bound on the number of join orders with N tables is $(N - 1)!$ where ! indicates the factorial function. The factorial function has explosive growth as it grows faster than exponential functions. The factorial function overstates the number of join orders because it includes infeasible join orders. As the number of joins increases, the number of feasible join orders approaches the total number of join orders. Figure 8.26 shows the number of access plans as a function of the number of tables using the factorial function as an upper limit for the number of join orders. Note that the scale of the y-axis is a power of 10. So for a query with 10 tables, the number of access plans is more than 10^6.

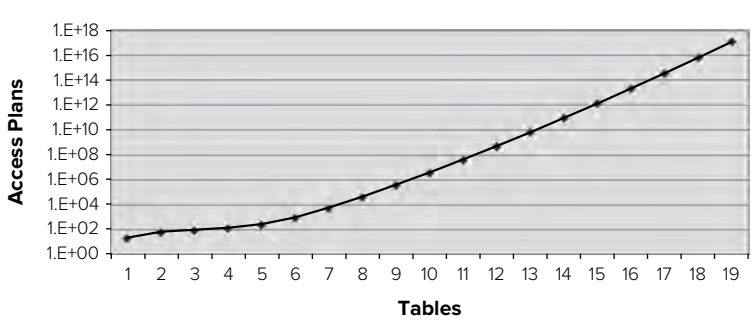


FIGURE 8.26
Access Plan Growth by Number of Tables

The number of access plans indicates that query optimization can consume considerable resources. Query optimization components use many heuristics to reduce the number of plans evaluated. Still, evaluating access plans can involve substantial computing resources and elapsed time when a query contains more than eight tables.

Access Plan Execution The last phase executes the selected access plan. The query optimization component either generates machine code or interprets the access plan. Execution of machine code results in faster response than interpreting an access plan. However, most DBMSs interpret access plans because of the variety of hardware supported. The performance difference between interpretation and machine code execution is usually not significant in most situations.

8.4.2 Improving Optimization Decisions

Even though the query optimization component performs its role automatically, the database administrator also has a role to play. The database administrator should review access plans of poorly performing queries and updates. Enterprise DBMSs typically provide graphical displays of access plans to facilitate review. Graphical displays are essential because text displays of hierarchical relationships are difficult to read.

To improve poor decisions in access plan selection, some enterprise DBMSs allow hints that influence the choice of access plans. For example, Oracle provides hints to choose the optimization goal, the file structures to access individual tables, the join algorithm, and the join order. Hints should be used with caution because they override the judgment of the optimizer. Hints with join algorithms and join orders are especially problematic because of the subtlety of these decisions. Overriding the judgment of the optimizer should only be done as a last resort after determining the cause of poor performance. In many cases, a database administrator can fix problems with table profile deficiencies and query coding style to improve performance rather than override the judgment of the optimizer.

Table Profile Deficiencies The query optimization component needs detailed and current statistics to evaluate access plans. Statistics that are not detailed enough or outdated can lead to the choice of poor access plans. Most DBMSs provide control over the level of detail of statistics and the currency of the statistics. Some DBMSs even allow dynamic database sampling at optimization time, but normally this level of data currency is not needed.

If statistics are not collected for a column, most DBMSs use the uniform value assumption to estimate the number of rows. Using the uniform value assumption often leads to sequential file access rather than Btree access if the column has significant skew in its values. For example, consider a query to list employees with salaries greater than $100,000. If the salary range is $10,000 to $2,000,000, about 95 percent of the employee table should satisfy this condition using the uniform value assumption. For most companies, however, few employees would have a salary greater than $100,000. Using the estimate from the uniform value assumption, the optimizer will choose a sequential file instead of a Btree to access the employee table. The estimate would not improve much using an equal-width histogram because of the extreme skew in salary values.

An equal-height histogram will provide much better estimates. To improve estimates using an equal-height histogram, the number of ranges should be increased. For example with 10 ranges, the maximum error is about 10% and the expected error is about 5%. To decrease the maximum and expected estimation errors by 50%, the number of ranges should be doubled. A database administrator should increase the number of ranges if estimation errors for the number of rows cause poor choices for accessing individual tables.

A hint can be useful for conditions involving parameter values. If a database administrator knows that the typical parameter values result in the selection of few rows, a hint can be used to force the optimization component to use an index.

In addition to detailed statistics about individual columns, an optimization component sometimes needs detailed statistics on combinations of columns. If a combination of columns appears in the WHERE clause of a query, statistics on the column combination are important if the columns are not independent. For example, employee salaries and positions are typically related. A WHERE clause with both columns such as `EmpPosition = 'Janitor' AND Salary > 50000` would likely have few rows that satisfy both conditions. An optimization component with no knowledge of the relationship among these columns would be likely to significantly overestimate the number of rows in the result. The poor estimate about the number of rows would likely lead to an erroneous decision by the optimizer causing slow query execution. The optimizer would probably choose a sequential table scan instead of combining indexes on each column.

Most optimization components assume that combinations of columns are statistically independent to simplify the estimation of the number of rows. Because optimizer errors involving conditions on combination of columns are reasonably common, DBMS vendors have begun providing tools to improve row estimates. For example Oracle provides a hint to force the optimizer to use an index join, an access method that combines indexes on individual columns.

Oracle Tools to Improve Optimization Choices Since hints override the judgment of the optimizer, Oracle provides several tools that extend the judgment of the optimizer. Extended statistics in Oracle support statistics collection on combinations of columns, not just individual columns. In the example from the previous paragraph, extended statistics could be collected on the column combination <*EmpPosition*, *Salary*>. The optimizer estimates the number of rows using the extended statistics instead of relying on individual column statistics and the independence assumption. To reduce the effort to calculate extended statistics, Oracle can use random sampling instead of scanning entire tables. The major drawback with extended statistics is identification of column combinations needing extended statistics. For a table with many columns such as a customer table, a DBA can be overwhelmed with the number of column combinations to consider. In response to this difficulty, Oracle provides a tool to suggest column combinations for extended statistics by analyzing a query workload.

Oracle provides adaptive access plans to deal with parameterized queries. The Oracle optimizer analyzes the values used as parameters to determine if the access plan for a query should be recompiled based on row estimates for common parameter values. The Oracle optimizer can choose the correct access plan based on a parameter value in a query.

Dynamic sampling is another tool to improve row estimates. With dynamic sampling, Oracle samples tables involved in a query during the compilation process instead of using prebuilt statistics. Dynamic sampling increases query compilation time but can improve query execution times with improved row estimates. In Oracle 11, the optimizer can decide to use dynamic sampling depending on the number of tables in an SQL statement, the complexity of the WHERE clause, and the collection of prebuilt statistics available. Alternatively, a DBA can limit the usage of dynamic sampling with the OPTIMIZER_DYNAMIC_SAMPLING parameter.

Oracle 12c extends the optimizer with more adaptive capabilities. Alternative access plans can be stored to substantially reduce the need to determine a new access plan for a query as a database changes. If statistics collected during compilation of a plan appear deficient during query execution, the plan might be changed dynamically such as switching join algorithms. Dynamic sampling has been extended to joins and group-by conditions, beyond just single table statistics in earlier Oracle versions. The Oracle 12c optimizer considers dynamic statistics based on the complexity of query conditions, the existing base statistics, and the expected execution time for the SQL statement.

Query Coding Practices Poorly written queries can lead to slow query execution. The database administrator should review poorly performing queries looking for

coding practices that lead to slow performance. The remainder of this subsection explains the coding practices that can lead to poorly performing queries. Table 8-9 provides a convenient summary of the coding practices.

- You should avoid functions on indexable columns as functions eliminate the opportunity to use an index unless a function index exists. You should be especially aware of implicit type conversions even if a function is not used. An implicit type conversion occurs if the data type of a column and the associated constant value do not match. For example the condition, `OffYear = '2018'` causes an implicit conversion of the *OffYear* column to a character data type. The conversion eliminates the possibility of using an index on *OffYear*.
- If a column expression is typically required in a query, some DBMSs such as Oracle support function indexes on column expressions. For example, Oracle supports an index on the column expression `upper(EmpFirstName)`. If an index is created for a column expression, extended statistics should be collected for the column expression.
- Queries with extra join operations will slow performance. The execution speed of a query is primarily determined by the number of join operations so eliminating unnecessary join operations may significantly decrease execution time.
- For queries involving 1-M relationships in which there is a condition on the join column, you should make the condition on the parent table rather than the child table. The condition on the parent table can significantly reduce the effort in joining the tables.
- For queries involving the HAVING clause, eliminate conditions that do not involve aggregate functions. Conditions involving simple comparisons of columns in the GROUP BY clause belong in the WHERE clause, not the HAVING clause. Moving these conditions to the WHERE clause will eliminate rows sooner, thus providing faster execution.
- You should avoid Type II nested queries (see Chapter 9), especially when the nested query performs grouping with aggregate calculations. Many DBMSs perform poorly as query optimization components often do not consider efficient

TABLE 8-9
Summary of Coding Practices

Coding Practice	Recommendation	Performance Issue
Functions on columns in conditions	Avoid functions on columns unless a function index exists	Eliminates possible usage of index
Implicit type conversions	Use constants with data types matching the corresponding columns	Eliminates possible usage of index
Extra join operations	Eliminate unnecessary join operations by looking for tables that do not involve conditions or result columns	Execution time is primarily determined by the number of join operations.
Conditions on join columns	Conditions on join columns should use the parent table not the child table.	Reducing the number of rows in the parent table will decrease execution time of join operations.
Row conditions in the HAVING clause	Move row conditions in the HAVING clause to the WHERE clause	Row conditions in the WHERE clause allow reduction in the intermediate result size.
Type II nested queries with grouping (Chapter 9)	Convert Type II nested queries into separate queries.	Query optimization components often do not consider efficient ways to implement Type II nested queries.
Queries using complex views (Chapter 10)	Rewrite queries using complex views to eliminate view references	An extra query may be executed.
Rebinding of queries (Chapter 11)	Ensure that queries in a stored procedure are bound once	Repetitive binding involves considerable overhead for complex queries.

ways to implement Type II nested queries. You can improve query execution speed by replacing a Type II nested query with a separate query.
- Queries with complex views can lead to poor performance because an extra query may be executed. Chapter 10 describes view processing with some guidelines for limiting the complexity of views.
- The optimization process can be time-consuming, especially for queries containing more than seven or eight tables. To reduce optimization time, most DBMSs save access plans to avoid the time-consuming phases of the translation process. **Query binding** is the process of associating a query with an access plan. Most DBMSs rebind automatically if a query changes or the database changes (file structures, table profiles, data types, etc.). Chapter 11 discusses query binding for dynamic SQL statements inside of a stored procedure.

Query Binding
associating an access plan with an SQL statement. Binding can reduce execution time for complex queries because the time-consuming phases of the translation process are not performed after the initial binding occurs.

Oracle SQL Tuning Advisor Oracle provides the SQL Tuning Advisor to help a database designer improve performance of high-load SQL statements. The SQL Tuning Advisor executes as an option of the Oracle query optimizer. In normal mode, the optimizer determines the best access plan for a SQL statement under time restrictions. In tuning mode, the optimizer makes recommendations to improve performance of a specified SQL statement. Since tuning mode can take extensive time, it should only be used periodically when problems occur. The SQL Tuning Advisor makes the following types of recommendations.

- Gathers statistics on objects with missing or stale statistics. Statistic recommendations require comparison of stored statistics for a query with statistics obtained from sampling the associated tables.
- Determines if new indexes can significantly enhance the performance of a query. If index changes are identified, the tuning advisor suggests the use of the SQL Access Advisor (see Section 8.5.2) to check the impact of these indexes on a representative SQL workload.
- Suggests alternative coding practices such as changing nested query coding style.

To provide support for SQL statement monitoring, the Automatic Database Diagnostic Monitor (ADDM) identifies high load SQL statements. By default, the ADDM executes every hour although a DBA can change this frequency. The ADDM recommends tuning analysis for high-load SQL statements.

8.5 INDEX SELECTION

Index selection is the most important decision available to the physical database designer. However, it also can be one of the most difficult decisions. As a designer, you need to understand the difficulty of index selection and the limitations of performing index selection without an automated tool. This section helps you gain this knowledge by defining the index selection problem, discussing trade-offs in selecting indexes, and presenting index selection rules for moderate-size databases.

Index
a secondary file structure that provides an alternative path to the data. In a clustering index, the order of the data records is close to the index order. In a nonclustering index, the order of the data records is unrelated to the index order.

8.5.1 Problem Definition

Index selection involves two kinds of indexes, clustering and nonclustering. In a clustering index, the order of the rows is close to the index order. Close means that physical records will not have to be accessed more than one time if the index is accessed sequentially. Figure 8.27 shows the sequence set of a B+tree index pointing to associated rows inside physical records. Note that for a given node in the sequence set, most associated rows are clustered inside the same physical record. Ordering the row data by the index column is a simple way to make a clustering index.

In contrast, a nonclustering index does not have this closeness property. In a nonclustering index, the order of the rows is not related to the index order.

FIGURE 8.27
Clustering Index Example

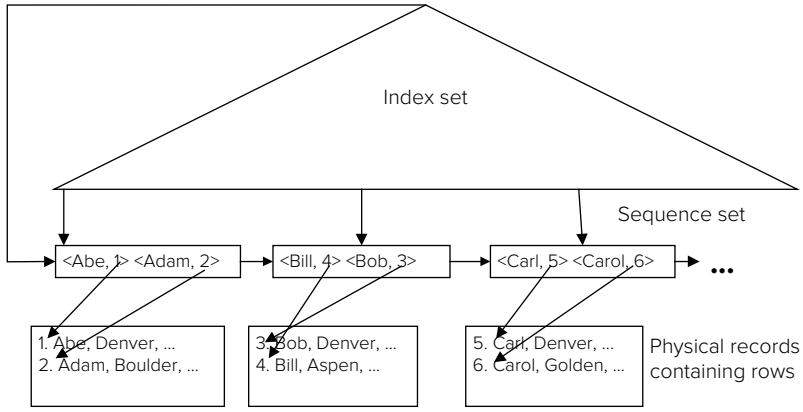


Figure 8.28 shows that the same physical record may be repeatedly accessed when using the sequence set. The pointers from the sequence set nodes to the rows cross many times, indicating that the index order is different from the row order.

Index Selection Problem
for each table, select at most one clustering index and zero or more nonclustering indexes.

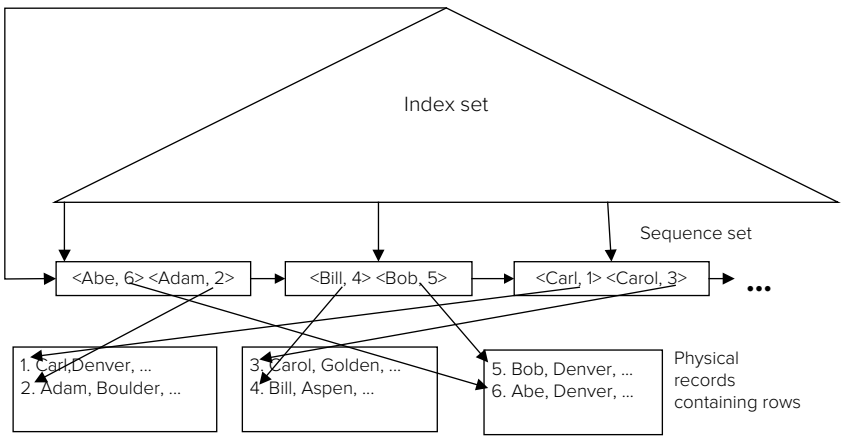
Index selection involves choices about clustering and nonclustering indexes, as shown in Figure 8.29. It is usually assumed that each table is stored in one file. The SQL statements indicate the database work to be performed by applications. The weights should combine the frequency of a statement with its importance. The table profiles must be specified in the same level of detail as required for query optimization.

Usually, the index selection problem is restricted to Btree indexes and separate files for each table. The references at the end of the chapter provide details about using other kinds of indexes (such as hash indexes) and placing data from multiple tables in the same file. However, these extensions make the problem more difficult without adding much performance improvement. The extensions are useful only in specialized situations.

8.5.2 Trade-offs and Difficulties

The best selection of indexes balances faster retrieval with slower updates. A nonclustering index can improve retrievals by providing fast access to selected records. In Example 8.2, a nonclustering index on the *OffYear*, *OffTerm*, or *CourseNo* columns may be useful if relatively few rows satisfy the associated condition in the query. Usually, less than 5 percent of the rows must satisfy a condition for a nonclustering index to be useful. It is unlikely that any of the conditions in Example 8.2 will yield such a small fraction of the rows.

FIGURE 8.28
Nonclustering Index Example



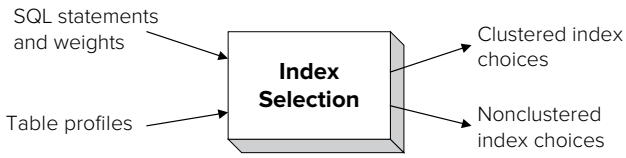


FIGURE 8.29
Inputs and Outputs of Index Selection

For optimizers that support multiple index access for the same table, nonclustering indexes can be useful even if a single index by itself does not provide high enough selectivity of rows. For example, the number of rows after applying the conditions on *CourseNo*, *OffYear*, and *OffTerm* should be small, perhaps 20 to 30 rows. If an optimizer can accurately estimate the number of rows, indexes on the three columns can be combined to access the *Offering* rows. Thus, the ability to use multiple indexes on the same table increases the usefulness of nonclustering indexes.

A nonclustering index can also be useful in a join if one table in the join has a small number of rows in the result. For example, if only a few *Offering* rows meet all three conditions in Example 8.2, a nonclustering index on the *Faculty.FacNo* column may be useful when joining the *Faculty* and *Offering* tables.

Example 8.2

Join of the Faculty and Offering Tables

```
SELECT FacName, CourseNo, OfferNo
 FROM Offering, Faculty
 WHERE CourseNo LIKE 'IS%' AND OffYear = 2018
   AND OffTerm = 'FALL'
   AND Faculty.FacNo = Offering.FacNo
```

A clustering index can improve retrievals under more situations than a nonclustering index. A clustering index is useful in the same situations as a nonclustering index except that the number of resulting rows can be larger. For example, a clustering index on the *CourseNo*, *OffYear*, or *OffTerm* columns may be useful if perhaps 20 percent of the rows satisfy the associated condition in the query.

A clustering index can also be useful on joins because it avoids the need to sort. For example, using clustering indexes on the *Offering.FacNo* and *Faculty.FacNo* columns, the *Offering* and *Faculty* tables can be joined by merging the rows from each table. Merging rows is often a fast way to join tables if the tables do not need to be sorted (clustering indexes exist).

The cost to maintain indexes as a result of INSERT, UPDATE, and DELETE statements balances retrieval improvements. INSERT and DELETE statements affect all indexes of a table. Thus, many indexes on a table are not preferred if the table has frequent insert and delete operations. UPDATE statements affect only the columns listed in the SET clause. If UPDATE statements on a column are frequent, the benefit of an index is usually los.

Clustering index choices are more sensitive to maintenance than nonclustering index choices. Clustering indexes are more expensive to maintain than nonclustering indexes because the data file must be changed similar to an ordered sequential file. For nonclustering indexes, the data file can be maintained as an unordered sequential file.

Difficulties of Index Selection Index selection is difficult to perform well for a variety of reasons as explained in this subsection. If you understand the reasons that index selection is difficult, you should gain insights into the computer-aided tools to help in the selection process for large databases. Enterprise DBMSs and some outside vendors provide computer-aided tools to assist with index selection.

- Application weights are difficult to specify. Judgments that combine frequency and importance can make the result subjective.
- Distribution of parameter values is sometimes needed. Many SQL statements in reports and forms use parameter values. If parameter values vary from being highly selective to not very selective, selecting indexes is difficult.
- The behavior of the query optimization component must be known. Even if an index appears useful for a query, the query optimization component must use it. There may be subtle reasons why the query optimization component does not use an index, especially a nonclustering index.
- The number of choices is large. Even if indexes on combinations of columns are ignored, the theoretical number of choices is exponential in the number of columns (2^{NC} where NC is the number of columns). Although many of these choices can be easily eliminated, the number of practical choices is still quite large.
- Index choices can be interrelated. The interrelationships can be subtle, especially when choosing indexes to improve join performance. The selection of a clustering index on a parent table can influence the selection on a related child table.

An index selection tool can help with the last three problems. A good tool should use the query optimization component to derive cost estimates for each query under a given choice of indexes. However, a good tool cannot help alleviate the difficulty of specifying application profiles and parameter value distributions. Other tools may be provided to specify and capture application profiles.

Oracle SQL Access Advisor The Oracle SQL Access Advisor performs a variety of physical design tasks including index selection. The SQL Access Advisor makes recommendations for indexes (B+tree, bitmap, hash, and function) using a workload specification. To ensure realistic recommendations, the SQL Access Advisor uses the Oracle optimization component to determine the impact of index choices on a workload. Workloads can be directly provided or collected from database operations. A workload specification involves a collection of SQL statements with properties for each SQL statement including the priority and execution statistics such as optimizer cost, rows retrieved, frequency, memory usage, CPU time, and disk reads. Each query in a workload must be ranked by one or more query properties such as priority and frequency.

To support workload collection, Oracle provides a workload database known as the Automatic Workload Repository (AWR). The AWR contains execution history of SQL statements including the statement text, CPU time, elapsed time, disk read, rows retrieved, and frequency. The AWR is populated by periodic snapshots of database operations. By default, Oracle automatically generates snapshots of database operations once every hour and retains the statistics in the workload repository for eight days. A database designer can modify the default snapshot interval as well as create snapshots manually.

The SQL Access Advisor is often used in tandem with the SQL Tuning Advisor. Recall from Section 8.4.2 that the SQL Tuning Advisor makes recommendations about improving the performance of high-load SQL statements. If the SQL Tuning Advisor recommends creation of additional indexes for a high-load SQL statement, the database designer should use the SQL Access Advisor to evaluate the impact of the additional indexes on other statements in the workload. If the additional indexes adversely impact important SQL manipulation statements, the database designer may want to ignore the recommendations of the SQL Tuning Advisor.

8.5.3 Selection Rules

Despite the difficulties previously discussed, you usually can avoid poor index choices by following some simple rules. You also can use the rules as a starting point for a more careful selection process.

Rule 1: A primary key is a good candidate for a clustering index.
Rule 2: To support joins, consider indexes on foreign keys. A nonclustering index on a foreign key is a good idea when there are important queries with highly selective conditions on the related primary key table. A clustering index is a good choice when most joins use a parent table with a clustering index on its primary key, and the queries do not have highly selective conditions on the parent table.
Rule 3: A column with almost unique values may be a good choice for a nonclustering index if it is used in equality conditions. Almost unique means that the number of column values is close to the number of rows containing non-null values for the column.
Rule 4: A column used in highly selective range conditions is a good candidate for a nonclustering index.
Rule 5: A combination of columns used together in query conditions may be good candidates for nonclustering indexes if the combined conditions return few rows, the DBMS optimizer supports multiple index access, and the columns are stable. Individual indexes should be created on each column.
Rule 6: A frequently updated column is not a good index candidate.
Rule 7: Volatile tables (lots of insertions and deletions) should not have many indexes.
Rule 8: Stable columns with few values are good candidates for bitmap indexes if the columns appear in WHERE conditions.
Rule 9: Avoid indexes on combinations of columns. Most optimization components can use multiple indexes on the same table. An index on a combination of columns is not as flexible as multiple indexes on individual columns of the table.

Applying the Selection Rules Let us apply these rules to the *Student*, *Enrollment*, and *Offering* tables of the university database. Table 8-10 lists summaries of the table profiles. More detail about column and relationship distributions can be encoded in histograms. Table 8-11 lists SQL statements and frequencies for these tables. The names beginning with $ represent parameters supplied by a user. The frequencies assume a student population of 30,000, in which students enroll in an average of four offerings per term. After a student graduates or leaves the university, the *Student* and *Enrollment* rows are archived.

Table 8-12 lists index choices according to the index selection rules. Only a few indexes are recommended because of the frequency of maintenance statements and the absence of highly selective conditions on columns other than the primary key. In queries 9 and 10, although the individual conditions on *OffTerm* and *OffYear* are not highly selective, the combined condition may be reasonably selective to recommend bitmap indexes, especially in query 9 with the additional condition on *CourseNo*. There is an index on *StdGPA* because parameter values should be very high or low, providing high selectivity with few rows in the result. A more detailed study of the *StdGPA* index may be necessary because it has a considerable amount of update activity. Even

TABLE 8-10
Table Profiles

Table	Number of Rows	Column (Number of Unique Values)
Student	30,000	StdNo (PK), StdLastName (29,000), StdAddress (20,000), StdCity (500), StdZip (1,000), StdState (50), StdMajor (100), StdGPA (400)
Enrollment	300,000	StdNo (30,000), OfferNo (2,000), EnrGrade (400)
Offering	10,000	OfferNo (PK), CourseNo (900), OffTime (20), OffLocation (500), FacNo (1,500), OffTerm (4), OffYear (10), OffDays (10)
Course	1,000	CourseNo (PK), CrsDesc (1,000), CrsUnits (6)
Faculty	2,000	FacNo (PK), FacLastName (1,900), FacAddress (1,950), FacCity (50), FacZip (200), FacState (3), FacHireDate (300), FacSalary (1,500), FacRank (10), FacDept (100)

TABLE 8-11
SQL Statements and Frequencies for Several University Database Tables

SQL Statement	Frequency	Comments
1. `INSERT INTO Student …`	7,500/year	Beginning of year
2. `INSERT INTO Enrollment …`	120,000/term	During registration
3. `INSERT INTO Offering …`	1,000/year	Before scheduling deadline
4. `DELETE Student WHERE StdNo = $X`	8,000/year	After separation
5. `DELETE Offering WHERE OfferNo = $X`	1,000/year	End of year
6. `DELETE Enrollment WHERE OfferNo = $X AND StdNo = $Y`	64,000/year	End of year
7. `SELECT * FROM Student WHERE StdGPA > $X AND StdMajor = $Y`	1,200/year	$X is usually very large or small
8. `SELECT * FROM Student WHERE StdNo = $X`	30,000/term	
9. `SELECT * FROM Offering WHERE OffTerm = $X AND OffYear = $Y AND CourseNo LIKE $Z`	60,000/term	Few rows in result
10. `SELECT * FROM Offering, Enrollment WHERE StdNo = $X AND OffTerm = $Y AND OffYear = $Z AND Offering.OfferNo = Enrollment.OfferNo`	30,000/term	Few rows in result
11. `UPDATE Student SET StdGPA = $X WHERE StdNo = $Y`	30,000/term	Updated at end of reporting form
12. `UPDATE Enrollment SET EnrGrade = $X WHERE StdNo = $Y AND OfferNo = $Z`	120,000/term	Part of grade reporting form
13. `UPDATE Offering SET FacNo = $X WHERE OfferNo = $Y`	500/year	
14. `SELECT FacNo, FacFirstName, FacLastName FROM Faculty WHERE FacRank = $X AND FacDept = $Y`	1,000/term	Most occurring during registration
15. `SELECT * FROM Student, Enrollment, Offering WHERE Offering.OfferNo = $X AND Student.StdNo = Enrollment.StdNo AND Offering.OfferNo = Enrollment.OfferNo`	4,000/year	Most occurring beginning of semester

TABLE 8-12
Index Selections for the University Database Tables

Column	Index Kind	Rule
Student.StdNo	Clustering	1
Faculty.FacNo	Clustering	1
Student.StdGPA	Nonclustering	4
Offering.OfferNo	Clustering	1
Enrollment.OfferNo	Clustering	2
Faculty.FacRank	Bitmap	8
Faculty.Dept	Bitmap	8
Offering.OffTerm	Bitmap	8
Offering.OffYear	Bitmap	8

though not suggested by the SQL statements, the *StdLastName* and *FacLastName* columns also may be good index choices because they are almost unique (a few duplicates) and reasonably stable. If there are additional SQL statements that use these columns in conditions, nonclustered indexes should be considered.

Although SQL:2016 does not support statements for indexes, most DBMSs support index statements. In Example 8.3, the word following the INDEX keyword is the

name of the index. The CREATE index statement also can be used to create an index on a combination of columns by listing multiple columns in the parentheses. The Oracle CREATE INDEX statement cannot be used to create a clustered index. To create a clustered index, Oracle provides the ORGANIZATION INDEX clause as part of the CREATE TABLE statement.

Example 8.3

Oracle CREATE INDEX statements

```
CREATE UNIQUE INDEX StdNoIndex ON Student (StdNo)
CREATE UNIQUE INDEX FacNoIndex ON Faculty (FacNo)
CREATE INDEX StdGPAIndex ON Student (StdGPA)
CREATE UNIQUE INDEX OfferNoIndex ON Offering (OfferNo)
CREATE INDEX EnrollOfferNoIndex ON Enrollment (OfferNo)
CREATE BITMAP INDEX OffYearIndex ON Offering (OffYear)
CREATE BITMAP INDEX OffTermIndex ON Offering (OffTerm)
CREATE BITMAP INDEX FacRankIndex ON Faculty (FacRank)
CREATE BITMAP INDEX FacDeptIndex ON Faculty (FacDept)
```

8.6 ADDITIONAL CHOICES IN PHYSICAL DATABASE DESIGN

Although index selection is the most important decision of physical database design, there are other decisions that can significantly improve performance. This section discusses two decisions, denormalization and record formatting, that can improve performance in selected situations. Next, this section presents parallel processing to improve database performance, an increasingly popular alternative. Finally, several ways to improve performance related to specific kinds of processing are briefly discussed.

8.6.1 Denormalization

Denormalization combines tables so that they are easier to query. After combining tables, the new table may violate a normal form such as BCNF. Although some of the denormalization techniques do not lead to violations in a normal form, they still make a design easier to query and more difficult to update. Denormalization should always be done with extreme care because a normalized design has important advantages. Chapter 7 described one situation for denormalization: ignoring a functional dependency if it does not lead to significant modification anomalies. This section describes additional situations under which denormalization may be justified.

Normalized Designs

- Have better update performance.
- Require less coding to enforce integrity constraints.
- Support more indexes to improve query performance.

Repeating Groups A repeating group is a collection of associated values such as sales history, lines of an order, or payment history. The rules of normalization force repeating groups to be stored in a child table separate from the associated parent table. For example, the lines of an order are stored in an order line table, separate from a related order table. If a repeating group is always accessed with its associated parent table, denormalization may be a reasonable alternative.

Figure 8.30 shows a denormalization example of quarterly sales data. Although the denormalized design does not violate BCNF, it is less flexible for updating than the

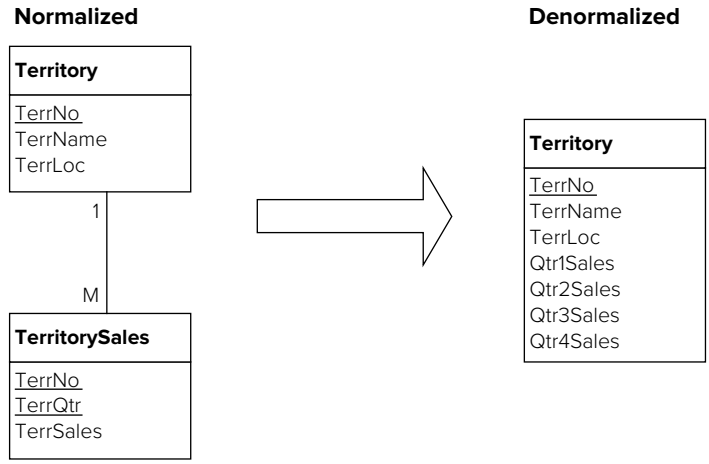


FIGURE 8.30
Denormalizing a Repeating Group

normalized design. The normalized design supports an unlimited number of quarterly sales as compared to only four quarters of sales results for the denormalized design. However, the denormalized design does not require a join to combine territory and sales data.

Generalization Hierarchies Following the conversion rule for generalization hierarchies in Chapter 6 can result in many tables. If queries often need to combine these separate tables, it may be reasonable to store the separate tables as one table. Figure 8.31 demonstrates denormalization of the *Emp*, *HourlyEmp*, and *SalaryEmp* tables. They have 1-1 relationships because they represent a generalization hierarchy. Although the denormalized design does not violate BCNF, the combined table may waste space because of null values. However, the denormalized design avoids the outer join operator to combine the tables.

Codes and Meanings Normalization rules require that foreign keys be stored alone to represent 1-M relationships. If a foreign key represents a code, the user often requests an associated name or description in addition to the foreign key value. For example, the user may want to see the state name in addition to the state code. Storing the name or description column along with the code violates BCNF, but it eliminates some join operations. If the name or description column is not changed often, denormalization may be a reasonable choice. Figure 8.32 demonstrates denormalization for the *Dept* and *Emp* tables. In the denormalized design, the *DeptName* column has been added to the *Emp* table.

8.6.2 Record Formatting

Record formatting decisions involve compression and derived data. With an increasing emphasis on storing complex data types such as audio, video, and images,

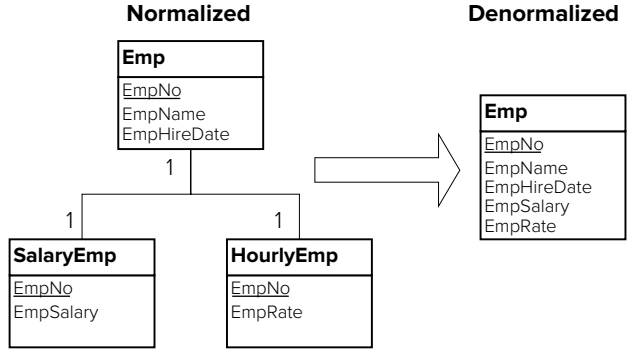


FIGURE 8.31
Denormalizing a Generalization Hierarchy

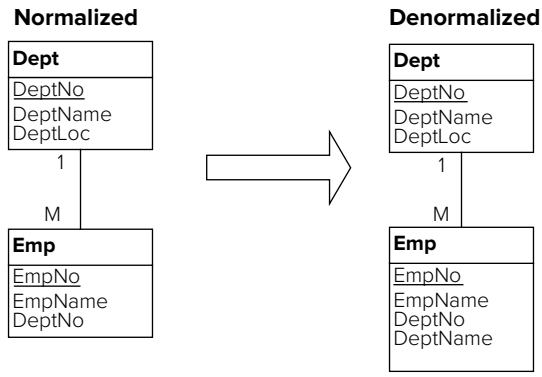


FIGURE 8.32
Denormalizing to Combine Code and Meaning Columns

compression is an important issue. In some situations, there are multiple compression alternatives available. Compression is a trade-off between input-output and processing effort. Compression reduces the number of physical records transferred but may require considerable processing effort to compress and decompress the data.

Decisions about derived data involve trade-offs between query and update operations. For query purposes, storing derived data reduces the need to retrieve data needed to calculate the derived data. However, updates to the underlying data require additional updates to the derived data. Storing derived data to reduce join operations may be reasonable. Figure 8.33 demonstrates derived data in the *Order* table. If the total amount of an order is frequently requested, storing the derived column *OrdAmt* may be reasonable. Calculating order amount requires a summary or aggregate calculation of related *OrdLine* and *Product* rows to obtain the *Qty* and *ProdPrice* columns. Storing the *OrdAmt* column avoids two join operations.

8.6.3 Parallel Processing

Retrieval and modification performance can be improved significantly through parallel processing. Retrievals involving many records can be improved by reading physical records in parallel. For example, a report to summarize daily sales activity may read thousands of records from several tables. Parallel reading of physical records can reduce significantly the execution time of the report. In addition, performance can be improved significantly for batch applications with many write operations and read/write of large logical records such as images.

As a response to the potential performance improvements, many DBMSs provide parallel processing capabilities. Chapter 18 describes architectures for parallel database processing. The presentation here is limited to an important part of any parallel

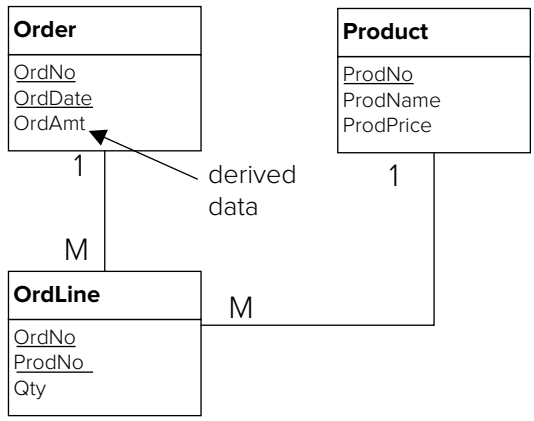


FIGURE 8.33
Storing Derived Data to Improve Query Performance

RAID
a collection of disks (a disk array) that operates as a single disk. RAID storage supports parallel read and write operations with high reliability.

database processing architecture, Redundant Arrays of Independent Disks (RAID)[3]. The **RAID** controller (Figure 8.34) enables an array of disks to appear as one large disk to the DBMS. For very high performance, a RAID controller can control as many as 90 disks. Because of the controller, RAID storage requires no changes in applications and queries. However, the query optimization component may be changed to account for the effect of parallel processing on access plan evaluation.

Striping is an important concept for RAID storage. Striping involves the allocation of physical records to different disks. A stripe is the set of physical records that can be read or written in parallel. Normally, a stripe contains a set of adjacent physical records. Figure 8.35 depicts an array of four disks that allows the reading or writing of four physical records in parallel.

To utilize RAID storage, a number of architectures have emerged. The architectures, known as RAID-X, support parallel processing with varying amounts of performance and reliability. Reliability is an important issue because the mean time between failures (a measure of disk drive reliability) decreases as the number of disk drives increases. To combat reliability concerns, RAID architectures incorporate redundancy using mirrored disks, error-correcting codes, and spare disks. Here are RAID architectures that provide varying amounts of performance and reliability.

- RAID-1: involves a full mirror or redundant array of disks to improve reliability. Each physical record is written to both disk arrays in parallel. Read operations from separate queries can access a disk array in parallel to improve performance across queries. RAID-1 involves the most storage overhead as compared to other RAID architectures.

FIGURE 8.34
Components of a RAID Storage System

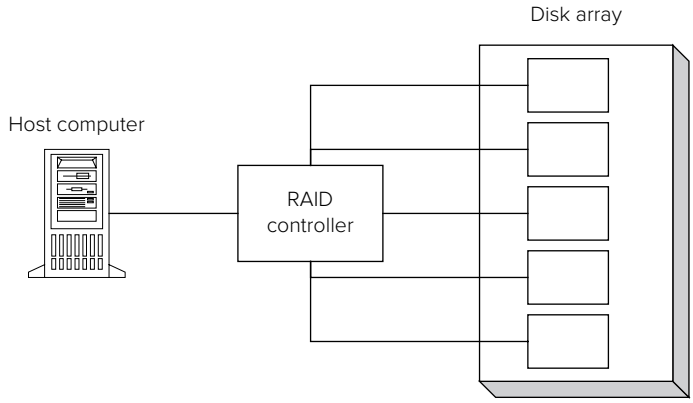
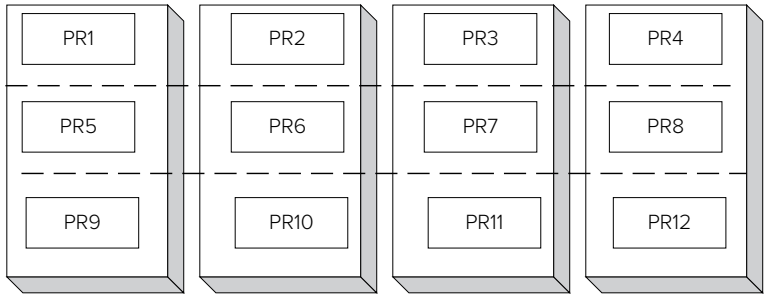


FIGURE 8.35
Striping in RAID Storage Systems

Each stripe consists of four adjacent physical records. Three stripes are shown separated by dotted lines.



[3] RAID originally was an acronym for Redundant Arrays of Inexpensive Disks. Because prices of disk drives have fallen dramatically since the invention of the RAID idea (1988), inexpensive has been replaced by independent.

- RAID-5: uses both data and error-correcting pages (known as parity pages) to improve reliability. Read operations can be performed in parallel on stripes. Write operations involve a data page and an error-correcting page on another disk. To reduce disk contention, the error-correcting pages are randomly located across disks. RAID-5 uses storage space more efficiently than RAID-1 but can involve slower write times because of the error-correcting pages. Thus, RAID-1 is often preferred for highly volatile parts of a database.
- Advanced architectures use a two-dimensional arrangement of mirroring and striping. A mirror of striped volumes tolerates failure to one volume because a volume is chosen and then used like striped RAID. At a higher cost, striping of mirrored volumes provides even higher reliability because it does not fail until one disk in each stripe fails.

Beyond these basic architectures, the RAID naming convention incorporates the number of data disks, parity disks, and spares. For example, 5+1+1 involves 5 data disks, 1 parity disk, and 1 hot spare disk.

To increase capacity beyond RAID and remove the reliance on storage devices attached to a server, Storage Area Networks (SANs) have been developed. A SAN provides a specialized high-speed network that connects storage devices and servers. The goal of SAN technology is to integrate different types of storage subsystems into a single system and to eliminate the potential bottleneck of a single server controlling storage devices. Many large organizations use SANs to integrate storage systems for operational databases, data warehouses, archival storage of documents, and traditional file systems.

An emerging trend for SANs is the usage of two types of permanent storage, traditional hard drives and solid state drives. Solid state drives provide substantial advantages with faster random data access and transfer times, lower power consumption, and improved reliability although cost per bit is substantially larger. Major SAN vendors support both types of drives allowing customers to make trade-offs between the usages of each type of drive in a storage network. Most SAN vendors provide a data movement feature (also known as tiering) in which inactive data percolates to lower cost but higher capacity storage (hard drives), thus releasing space on faster access but higher cost storage (solid state drives) for business-critical applications.

The usage two types of permanent storage can be generalized into the concept of Information Lifecycle Management (ILM), an important tool to manage big data. ILM uses multiple levels of storage and compression based on organization policies. For example, very active data is stored on solid state drives, active data on hard drives, less active data on optical drives, and historical data on archival storage. Each level of storage can have its own level of compression to balance data access costs against storage costs.

ILM products offered by storage providers and DBMS vendors provide tools to manage the migration of data between storage and compression levels. The Oracle ILM product automatically moves and compresses data according to organization policies. Oracle 12c provides two new features, the Heat Map and Automatic Data Optimization to support automatic migration and compression. The Heat Map tracks fine-grained usage to the row and segment levels, providing a detailed view of data access patterns over time. Automatic Data Optimization supports policy creation for data compression and movement. Using the Heat Map access patterns and lifecycle policies, Oracle ILM automatically compresses and moves data among storage and compression tiers.

8.6.4 Other Ways to Improve Performance

There are a number of other ways to improve database performance that are related to a specific kind of processing. For transaction processing (Chapter 17), you can add computing capacity (faster and more processors, memory, hard disks, and solid state disks) and make trade-offs in transaction design. For data warehouses (Chapters 12

to 15), you can add computing capacity, utilize specialized architectures, and design new tables with derived data. For distributed database processing (Chapter 18), you can allocate processing and data to various computing locations, partition data, and utilize parallel database processing. Data can be allocated by partitioning a table vertically (column subset) and horizontally (row subset) to improve performance and reliability. These design choices are discussed in the respective chapters in later parts of the textbook.

In addition to tuning performance for specific processing requirements, you also can improve performance by utilizing options specific to a DBMS. Fragmentation is an important concern in database storage as it is with any disk storage. Most DBMSs provide guidelines and tools to monitor and control fragmentation. In addition, most DBMSs have options for file structures that can improve performance. You must carefully study the specific DBMS to understand these options. It may take several years of experience and specialized education to understand options of a particular DBMS. However, the payoff of increased salary and demand for your knowledge can be worth the study.

CLOSING THOUGHTS

This chapter described the nature of the physical database design process and details about the inputs, environment, and design decisions. Physical database design involves details closest to the operating system such as movement of physical records. The objective of physical database design is to minimize computing resources (physical record accesses and central processing effort) without compromising the meaning of the database. Physical database design is a difficult process because the inputs can be difficult to specify, the environment is complex, and the number of choices can be overwhelming.

To improve your proficiency in performing physical database design, this chapter described details about the inputs and the environment of physical database design. This chapter described table profiles and application profiles as inputs that must be specified in sufficient detail to achieve an efficient design. The environment consists of file structures and the query optimization component of the DBMS. For file structures, this chapter described characteristics of sequential, hash, Btree, and bitmap structures used by many DBMSs. For query optimization, this chapter described the tasks of query optimization and tips to produce better optimization results.

After establishing the background for the physical database design process, the inputs, and the environment, this chapter described decisions about index selection, denormalization, and record formatting. For index selection, this chapter described trade-offs between retrieval and update applications and presented rules for selecting indexes. For denormalization and data formatting, this chapter presented a number of situations when they are useful.

This chapter concludes the database development process. After completing these steps, you should have an efficient table design that represents the needs of an organization. To complete your understanding of the database development process, the textbook's website provides a detailed case study in which to apply the ideas in preceding parts of this book.

REVIEW CONCEPTS

- Relationship between physical records and logical records
- Objective of physical database design
- Difficulties of physical database design
- Level of detail in table and application profiles

- Equal-height histograms to specify distribution of column values
- Characteristics of sequential, hash, Btree, and column store file structures
- Possible meanings of the letter *B* in the name *Btree*: balanced, bushy, block-oriented
- Index matching rules for determination if an index can be used for a search condition in a query
- Bitmap indexes for stable columns with few values
- Bitmap join indexes for frequent join operations using conditions on stable nonjoin columns
- Columnstore indexes for grouping queries on large tables
- Oracle storage terminology: tablespace, file, and extent
- Oracle storage parameters to control free space in a block: PCTFREE and PCTUSED
- Tasks of data language translation
- The usage of cost formulas and table profiles to evaluate access plans
- The importance of table profiles with sufficient detail for access plan evaluation
- Oracle tools (hints, extended statistics, and dynamic sampling) to overcome poor estimates about the number of rows in a query result
- Coding practices to avoid poorly executing queries
- Difference between clustering and nonclustering indexes
- Trade-offs in selecting indexes
- Index selection rules to avoid poor index choices
- Denormalization to improve join performance
- Record formatting to reduce physical record accesses and improve query performance
- RAID storage to provide parallel processing for retrievals and updates
- RAID architectures to provide parallel processing with high reliability
- Storage Area Networks (SANs) to integrate storage subsystems and to eliminate reliance upon server-attached storage devices
- Usage of solid state drives with faster access times, lower power consumption, and higher reliability to complement traditional hard drives in SANs
- Information Lifecycle Management (ILM), an important tool to manage big data, using multiple levels of storage and compression along with data movement among levels

QUESTIONS

1. What is the difference between a physical record access and a logical record access?
2. Why is it difficult to know when a logical record access results in a physical record access?
3. What is the objective of physical database design?
4. What computing resources are constraints rather than being part of the objective of physical database design?
5. What are the contents of table profiles?
6. What are the contents of application profiles?
7. Describe two ways to specify distributions of columns used in table and application profiles.

8. Why do most enterprise DBMSs use equal-height histograms to represent column distributions instead of equal-width histograms?
9. What is a file structure?
10. What is the difference between a primary and a secondary file structure?
11. Describe the uses of sequential files for sequential search, range search, and key search.
12. What is the purpose of a hash function?
13. Describe the uses of hash files for sequential search, range search, and key search.
14. What is the difference between a static hash file and a dynamic hash file?
15. Define the terms *balanced*, *bushy*, and *block-oriented* as they relate to Btree files.
16. Briefly explain the use of node splits and concatenations in the maintenance of Btree files.
17. What does it mean to say that Btrees have logarithmic search cost?
18. What is the difference between a Btree and a B+tree?
19. What is a bitmap?
20. How does a DBMS use a bitmap?
21. What are the components of a bitmap index record?
22. What is the difference between a bitmap column index and a bitmap join index?
23. When should bitmap indexes be used?
24. What is the difference between a primary and secondary file structure?
25. What does it mean to say that an index matches a column?
26. Why should composite indexes be used sparingly?
27. What happens in the query transformation phase of database language translation?
28. What is an access plan?
29. What is a multiple index scan?
30. How are access plans evaluated in query optimization?
31. Why does the uniform value assumption sometimes lead to poor access plans?
32. What does it mean to bind a query?
33. What join algorithm can be used for all joins operations?
34. For what join algorithms must the optimization component choose the outer and inner tables?
35. What join algorithm can combine more than two tables at a time?
36. When is the sort merge algorithm a good choice for combining tables?
37. When is the hash join algorithm a good choice for combining tables?
38. What is an optimizer hint? Why should hints be used cautiously?
39. Identify a situation in which an optimizer hint should not be used.
40. Identify a situation in which an optimizer hint may be appropriate.
41. What is the difference between a clustering and a nonclustering index?
42. When is a nonclustering index useful?
43. When is a clustering index useful?
44. What is the relationship of index selection to query optimization?
45. What are the trade-offs in index selection?
46. Why is index selection difficult?

47. When should you use the index selection rules?
48. Why should you be careful about denormalization?
49. Identify two situations when denormalization may be useful.
50. What is RAID storage?
51. For what kinds of applications can RAID storage improve performance?
52. What is striping in relation to RAID storage?
53. What techniques are used in RAID storage to improve reliability?
54. What are the advantages and disadvantages of RAID-1 versus RAID-5?
55. What is a Storage Area Network (SAN)?
56. What is the relationship of a SAN to RAID storage?
57. What are the trade-offs in storing derived data?
58. What processing environments also involve physical database design decisions?
59. What are some DBMS-specific concerns for performance improvement?
60. What is an implicit type conversion? Why may implicit type conversions cause poor query performance?
61. Why do unnecessary joins cause poor query performance?
62. Why should row conditions in the HAVING clause be moved to the WHERE clause?
63. How are the Oracle PCTFREE and PCTUSED parameters used to control free space in a data block?
64. What is the relationship among the Oracle storage terminology tablespace, file, and extent?
65. In Oracle, can a database object such as a table be stored in more than one file? (Hint: you can find the answer in the Oracle Database Concepts document.)
66. In Oracle, what is an indexed-organized file structure?
67. What tasks are performed by the Oracle SQL Tuning Advisor?
68. How does the Automatic Database Diagnostic Monitor support a DBA when using the SQL Tuning Advisor?
69. What tasks are performed by the Oracle SQL Access Advisor?
70. How does the Automatic Workload Repository support a DBA when using the SQL Access Advisor?
71. If the SQL Tuning Advisor recommends creating additional indexes to support a high-load SQL statement, should a DBA accept the recommendation or perform additional analysis?
72. Compare traditional hard drives and solid state drives on random access times, transfer times, power consumption, reliability, and cost per bit.
73. What feature is provided by Storage Area Networks (SANs) to support usage of both traditional hard drives and solid state drives?
74. What tools does Oracle provide to overcome poor estimates about the number of rows in a query result?
75. How does Information Lifecycle Management address problems of big data?
76. How is a columnstore different that traditional row storage?
77. What is a columnstore index?
78. What kind of queries does a columnstore index support?
79. What advantages does a columnstore provide for the kind of queries that it supports?

PROBLEMS

Besides the problems presented here, the case study in the textbook's website provides additional practice with a complete database design case including physical database design.

1. Use the following data to perform the indicated calculations. Show formulas that you used to perform the calculations.

   Row size = 100 bytes

   Number of rows = 100,000

   Primary key size = 6 bytes

   Physical record size = 4,096 bytes

   Pointer size = 4 bytes

   Floor(X) is the largest integer less than or equal to *X*.

   Ceil(X) is the smallest integer greater than or equal to *X*.

   1.1. Calculate the number of rows that can fit in a physical record. Assume that only complete rows can be stored (use the *Floor* function).

   1.2. Calculate the number of physical records necessary for a sequential file. Assume that physical records are filled to capacity except for the last physical record (use the *Ceil* function).

   1.3. If an unordered sequential file is used, calculate the number of physical record accesses on the average to retrieve a row with a specified key value.

   1.4. If an ordered sequential file is used, calculate the number of physical record accesses on the average to retrieve a row with a specified key value. Assume that the key exists in the file.

   1.5. Calculate the average number of physical record accesses to find a key that does not exist in an unordered sequential file and an ordered sequential file.

   1.6. Calculate the number of physical records for a static hash file. Assume that each physical record of the hash file is 70 percent full.

   1.7. Calculate the maximum branching factor on a node in a Btree. Assume that each record in a Btree consists of <key value, pointer> pairs.

   1.8. Using your calculation from problem 1.7, calculate the maximum height of a Btree index.

   1.9. Calculate the maximum number of physical record accesses to find a node in the Btree with a specific key value.

2. Answer query optimization questions for the following SQL statement:

```
SELECT * FROM Customer
 WHERE CustCity = 'DENVER' AND CustBal > 5000
   AND CustState = 'CO'
```

   2.1. Show four access plans for this query assuming that nonclustered indexes exist on the columns *CustCity, CustBal* (new column storing customer balances), and *CustState*. There is also a clustered index on the primary key column, *CustNo*.

   2.2. Using the uniform value assumption, estimate the fraction of rows that satisfy the condition on *CustBal*. The smallest balance is 0 and the largest balance is $10,000.

   2.3. Using the following histogram, estimate the fraction of rows that satisfy the condition on *CustBal*.

Histogram for *CustBal*

Range	Rows
0 – 100	1,000
101 – 250	950
251 – 500	1,050
501 – 1,000	1,030
1,001 – 2,000	975
2,001 – 4,500	1,035
4,501 –	1,200

3. Answer query optimization questions for the following SQL statement

```
SELECT OrdNo, OrdDate, Vehicle.ModelNo
 FROM Customer, Order, Vehicle
 WHERE CustBal > 5000
   AND Customer.CustNo = Vehicle.CustNo
   AND Vehicle.SerialNo = Order.SerialNo
```

   3.1. List the possible orders to join the *Customer*, *Order*, and *Vehicle* tables.

   3.2. For one of these join orders, make an access plan. Assume that Btree indexes only exist for the primary keys, *Customer.CustNo*, *Order.OrdNo*, and V*ehicle. SerialNo*.

4. For the following tables and SQL statements, select indexes that balance retrieval and update requirements. For each table, justify your choice using the rules discussed in Section 8.5.3.

   Customer(CustNo, CustName, CustCity, CustState, CustZip, CustBal)

   Order(OrdNo, OrdDate, *CustNo*)

   FOREIGN KEY CustNo REFERENCES Customer

SQL Statement	Frequency
1. `INSERT INTO Customer …`	100/day
2. `INSERT INTO Product …`	100/month
3. `INSERT INTO Order …`	3,000/day
4. `INSERT INTO OrdLine …`	9,000/day
5. `DELETE Product WHERE ProdNo = $X`	100/year
6. `DELETE Customer WHERE CustNo = $X`	1,000/year
7. `SELECT * FROM Order, Customer WHERE OrdNo = $X AND Order.CustNo = Customer.CustNo`	300/day
8. `SELECT * FROM OrdLine, Product WHERE OrdNo = $X AND OrdLine.ProdNo = Product.ProdNo`	300/day
9. `SELECT * FROM Customer, Order, OrdLine, Product WHERE CustName = $X AND OrdDate = $Y AND Customer.CustNo = Order.CustNo AND Order.OrdNo = OrdLine.OrdNo AND Product.ProdNo = OrdLine.ProdNo`	500/day
10. `UPDATE OrdLine SET OrdQty = $X WHERE OrdNo = $Y`	300/day
11. `UPDATE Product SET ProdPrice = $X WHERE ProdNo = $Y`	300/month

OrdLine(*OrdNo*, *ProdNo*, OrdQty)

FOREIGN KEY OrdNo REFERENCES Order

FOREIGN KEY ProdNo REFERENCES Product

Product(ProdNo, ProdName, ProdColor, ProdPrice)

4.1. For the *Customer* table, what columns are good choices for the clustered index? Nonclustered indexes?

4.2. For the *Product* table, what columns are good choices for the clustered index? Nonclustered indexes?

4.3. For the *Order* table, what columns are good choices for the clustered index? Nonclustered indexes?

4.4. For the *OrdLine* table, what columns are good choices for the clustered index? Nonclustered indexes?

5. Indexes on combinations of columns are not as useful as indexes on individual columns. Consider a combination index on two columns, *CustState* and C*ustCity*, where C*ustState* is the primary ordering and *CustCity* is the secondary ordering. For what kinds of conditions can the index be used? For what kinds of conditions is the index not useful?

6. For query 9 in problem 4, list the possible join orders considered by the query optimization component.

7. For the following tables of a financial planning database, identify possible uses of denormalization and derived data to improve performance. In addition, identify denormalization and derived data already appearing in the tables.

   The tables track financial assets held and trades made by customers. A trade involves a purchase or sale of a specified quantity of an asset by a customer. Assets include stocks and bonds. The *Holding* table contains the net quantity of each asset held by a customer. For example, if a customer has purchased 10,000 shares of IBM and sold 4,000, the *Holding* table shows a net quantity of 6,000. A frequent query is to list the most recent valuation for each asset held by a customer. The most recent valuation is the net quantity of the asset times the most recent price.

   Customer(CustNo, CustName, CustAddress, CustCity, CustState, CustZip, CustPhone)

   Asset(AssetNo, SecName, LastClose)

   Stock(AssetNo, OutShares, IssShares)

   Bond(AssetNo, BondRating, FacValue)

   PriceHistory(*AssetNo*, PHistDate, PHistPrice)

   FOREIGN KEY AssetNo REFERENCES Asset

   Holding(*CustNo*, *AssetNo*, NetQty)

   FOREIGN KEY CustNo REFERENCES Customer

   FOREIGN KEY AssetNo REFERENCES Asset

   Trade(TradeNo, *CustNo*, *AssetNo*, TrdQty, TrdPrice, TrdDate, TrdType, TrdStatus)

   FOREIGN KEY CustNo REFERENCES Customer

   FOREIGN KEY AssetNo REFERENCES Asset

8. Rewrite the following SQL statement to improve its performance on most DBMSs. Use the tips in Section 8.4.2 to rewrite the statement. The Oracle SQL statement uses the financial trading database shown in problem 7. The purpose of the statement is to list the customer number and the name of customers

and the sum of the amount of their completed October 2017 buy trades. The amount of a trade is the quantity (number of shares) times the price per share. A customer should be in the result if the sum of the amount of his/her completed October 2017 buy trades exceeds by 25 percent the sum of the amount of his/her completed September 2017 buy trades.

```
SELECT Customer.Custno, CustName,
       SUM(TrdQty * TrdPrice) AS SumTradeAmt
  FROM Customer, Trade
  WHERE Customer.CustNo = Trade.CustNo
    AND TrdDate BETWEEN '1-Oct-2017' AND '31-Oct-2017'
  GROUP BY Customer.CustNo, CustName
  HAVING TrdType = 'BUY' AND SUM(TrdQty * TrdPrice) >
   ( SELECT 1.25 * SUM(TrdQty * TrdPrice) FROM Trade
      WHERE TrdDate BETWEEN '1-Sep-2017' AND '30-Sep-2017'
        AND TrdType = 'BUY'
        AND Trade.CustNo = Customer.CustNo )
```

9. Rewrite the following SELECT statement to improve its performance on most DBMSs. Use the tips in Section 8.4.2 to rewrite the statement. The Oracle SQL statement uses the financial trading database shown in problem 7. Note that the *CustNo* column uses the integer data type. The foreign keys in the *Trade* table (*CustNo* and *AssetNo*) are required.

```
SELECT Customer.CustNo, CustName,
       TrdQty * TrdPrice, TrdDate, Asset.AssetNo
  FROM Customer, Trade, Asset
  WHERE Customer.CustNo = Trade.CustNo
    AND Trade.AssetNo = Asset.AssetNo
    AND TrdType = 'BUY' AND Trade.CustNo = '10001'
    AND TrdDate BETWEEN '1-Oct-2017' AND '31-Oct-2017'
```

10. For the following conditions and indexes, indicate if the index matches the condition.

- Index on TrdDate: TrdDate BETWEEN '1-Oct-2017' AND '31-Oct-2017'
- Index on CustPhone: CustPhone LIKE '(303)%'
- Index on TrdType: TrdType <> 'BUY'
- Bitmap column index on BondRating: BondRating IN ('AAA', 'AA', 'A')
- Index on <CustState, CustCity, CustZip>:
  - CustState = 'CO' AND CustCity = 'Denver'
  - CustState IN ('CO', 'CA') AND CustCity LIKE '%er'
  - CustState IN ('CO', 'CA') AND CustZip LIKE '8%'
  - CustState = 'CO' AND CustCity IN ('Denver', 'Boulder') AND CustZip LIKE '8%'

11. For the sample *Customer* and *Trade* tables below, construct bitmap indexes as indicated.

- Bitmap column index on *Customer.CustState*
- Join bitmap index on *Customer.CustNo* to the *Trade* table
- Bitmap join index on *Customer.CustState* to the *Trade* table

12. For the following tables and SQL statements, select indexes (clustering and nonclustering) that balance retrieval and update requirements. For each table, justify your choice using the rules discussed in Section 8.5.3.

Customer(CustNo, CustName, CustAddress, CustCity, CustState, CustZip, CustPhone)

Customer Table

RowID	CustNo	...	CustState
1	113344		CO
2	123789		CA
3	145789		UT
4	111245		NM
5	931034		CO
6	998245		CA
7	287341		UT
8	230432		CO
9	321588		CA
10	443356		CA
11	559211		UT
12	220688		NM

Trade Table

RowID	TradeNo	...	CustNo
1	1111		113344
2	1234		123789
3	1345		123789
4	1599		145789
5	1807		145789
6	1944		931034
7	2100		111245
8	2200		287341
9	2301		287341
10	2487		230432
11	2500		443356
12	2600		559211
13	2703		220688
14	2801		220688
15	2944		220688
16	3100		230432
17	3200		230432
18	3258		321588
19	3302		321588
20	3901		559211
21	4001		998245
22	4205		998245
23	4301		931034
24	4455		443356

Asset(AssetNo, AssetName, AssetType)
PriceHistory(*AssetNo*, PHistDate, PHistPrice)
FOREIGN KEY AssetNo REFERENCES Asset

Holding(*CustNo*, *AssetNo*, NetQty)

FOREIGN KEY CustNo REFERENCES Customer

FOREIGN KEY AssetNo REFERENCES Asset

Trade(TradeNo, *CustNo*, *AssetNo*, TrdQty, TrdPrice, TrdDate, TrdType, TrdStatus)

FOREIGN KEY CustNo REFERENCES Customer

FOREIGN KEY AssetNo REFERENCES Asset

13. For the workload of problem 12, are there any SELECT statements in which a DBA might want to use optimizer hints? Please explain the kind of hint that could be used and your reasoning for using it.
14. Investigate tools for managing access plans of an enterprise DBMS. You should investigate tools for textual display of access plans, graphical display of access plans, and hints to influence the judgment of the optimizer.
15. Investigate the database design tools of an enterprise DBMS or CASE tool. You should investigate command-level tools and graphical tools for index selection, table profiles, and application profiles.
16. Investigate the query optimization component of an enterprise DBMS or CASE tool. You should investigate the access methods for single table access, join algorithms, and usage of optimizer statistics.
17. Show the state of the Btree in Figure 8P.1 after insertion of the following keys: 115, 142, 111, 134, 170, 175, 127, 137, 108, and 140. The Btree has a maximum key capacity of 4. Show the node splits that occur while inserting the keys. You may use the interactive Btree tool on the website http://slady.net/java/bt/view.php for help with this problem.

SQL Statement	Frequency
1. `INSERT INTO Customer …`	100/day
2. `INSERT INTO Asset …`	100/quarter
3. `INSERT INTO Trade …`	10,000/day
4. `INSERT INTO Holding …`	200/day
5. `INSERT INTO PriceHistory …`	5,000/day
6. `DELETE Asset WHERE AssetNo = $X`	300/year
7. `DELETE Customer WHERE CustNo = $X`	3,000/year
8. `SELECT * FROM Holding, Customer, Asset, PriceHistory WHERE CustNo = $X AND Holding.CustNo = Customer.CustNo AND Holding.AssetNo = Asset.AssetNo AND Asset.AssetNo = PriceHistory.AssetNo AND PHistDate = $Y`	15,000/month
9. `SELECT * FROM Trade WHERE TradeNo = $X`	1,000/day
10. `SELECT * FROM Customer, Trade, Asset WHERE Customer.CustNo = $X AND TrdDate BETWEEN $Y AND $Z AND Customer.CustNo = Trade.CustNo AND Trade.AssetNo = Asset.AssetNo`	10,000/month
11. `UPDATE Trade SET TrdStatus = $X WHERE TradeNo = $Y`	1,000/day
12. `UPDATE Holding SET NetQty = $X WHERE CustNo = $Y AND AssetNo = $Z`	10,000/day
13. `SELECT * FROM Customer WHERE CustZip = $X AND CustPhone LIKE $Y%`	500/day
14. `SELECT * FROM Trade WHERE TrdStatus = $X AND TrdDate = $Y`	10/day
15. `SELECT * FROM Asset WHERE AssetName LIKE $X%`	500/day

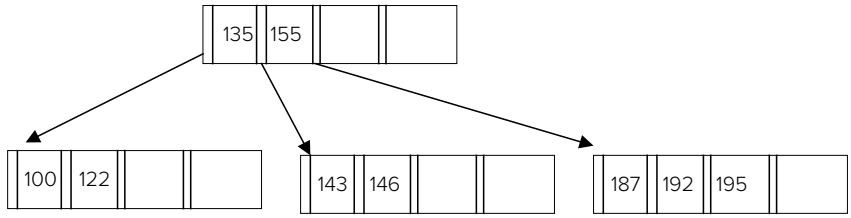


FIGURE 8P.1
Initial Btree Before Insertions and Deletions

18. Following on problem 17, show the state of the Btree after deleting the following keys: 108, 111, and 137. Show the node concatenations and key borrowings after deleting the keys. You may use the interactive Btree tool on the website http://slady.net/java/bt/view.php for help with this problem.
19. List the feasible join orders for joining tables T1, T2, T3, T4, and T5 on join conditions T1.T1No = T2.T1No, T2.T2No = T3.T2No, T3.T3No = T4.T3No, and T4.T4No = T5.T4No. How many infeasible join orders exist?
20. Use the following data to perform the indicated calculations. Show formulas that you used to perform the calculations.

    Row size = 180 bytes

    Number of rows = 2,000,000

    Primary key size = 8 bytes

    Physical record size = 4,096 bytes

    Pointer size = 8 bytes

    Floor(X) is the largest integer less than or equal to *X*.

    Ceil(X) is the smallest integer greater than or equal to *X*.

    20.1. Calculate the number of rows that can fit in a physical record. Assume that only complete rows can be stored (use the *Floor* function).

    20.2. Calculate the number of physical records necessary for a sequential file. Assume that physical records are filled to capacity except for the last physical record (use the *Ceil* function).

    20.3. If an unordered sequential file is used, calculate the number of physical record accesses on the average to retrieve a row with a specified key value.

    20.4. If an ordered sequential file is used, calculate the number of physical record accesses on the average to retrieve a row with a specified key value. Assume that the key exists in the file.

    20.5. Calculate the average number of physical record accesses to find a key that does not exist in an unordered sequential file and an ordered sequential file.

    20.6. Calculate the number of physical records for a static hash file. Assume that each physical record of the hash file is 70 percent full.

    20.7. Calculate the maximum branching factor on a node in a Btree. Assume that each record in a Btree consists of <key value, pointer> pairs.

    20.8. Using your calculation from problem 20.7, calculate the maximum height of a Btree index.

    20.9. Calculate the maximum number of physical record accesses to find a node in the Btree with a specific key value.

21. Use the following data to perform the indicated calculations. Show formulas that you used to perform the calculations.

    Row size = 520 bytes

    Number of rows = 3,000,000

    Primary key size = 16 bytes

Physical record size = 2,048 bytes

Pointer size = 8 bytes

Floor(X) is the largest integer less than or equal to *X*.

Ceil(X) is the smallest integer greater than or equal to *X*.

21.1. Calculate the number of rows that can fit in a physical record. Assume that only complete rows can be stored (use the *Floor* function).

21.2. Calculate the number of physical records necessary for a sequential file. Assume that physical records are filled to capacity except for the last physical record (use the *Ceil* function).

21.3. If an unordered sequential file is used, calculate the number of physical record accesses on the average to retrieve a row with a specified key value.

21.4. If an ordered sequential file is used, calculate the number of physical record accesses on the average to retrieve a row with a specified key value. Assume that the key exists in the file.

21.5. Calculate the average number of physical record accesses to find a key that does not exist in an unordered sequential file and an ordered sequential file.

21.6. Calculate the number of physical records for a static hash file. Assume that each physical record of the hash file is 70 percent full.

21.7. Calculate the maximum branching factor on a node in a Btree. Assume that each record in a Btree consists of <key value, pointer> pairs.

21.8. Using your calculation from problem 21.7, calculate the maximum height of a Btree index.

21.9. Calculate the maximum number of physical record accesses to find a node in the Btree with a specific key value.

22. List the feasible join orders for joining tables T2, T3, T4, and T5 on join conditions T2.T2No = T3.T2No, T3.T3No = T4.T3No, and T4.T4No = T5.T4No. How many infeasible join orders exist?

23. The following questions involve a SELECT statement and database with the following facts.

- Individual, non-clustering indexes exist on the *WageIncome* and *HighestDegree* columns in the *Census* table.
- The selectivity (fraction of rows) estimate of the condition, `HighestDegree = 'HS Graduate'`, is 0.30.
- The selectivity estimate of the condition, `WageIncome > 60000`, is 0.30.
- The actual selectivity of the condition, `WageIncome > 60000 AND HighestDegree = 'HS Graduate'`, is 0.015.

23.1. What would the Oracle optimizer estimate as the selectivity (fraction of rows) of the joint condition on *WageIncome* and *HighestDegree* when only using statistics on individual columns?

23.2. What access method would Oracle likely choose for the *Census* table if no other conditions in the query involve the *Census* table?

23.3. What access method should Oracle choose for the Census table if no other conditions in the query involve the Census table?

23.4. What Oracle hint should be used to force Oracle to combine indexes on *WageIncome* and *HighestDegree*?

23.5. What other methods (besides hints) can be used in Oracle to overcome the poor row estimates for conditions on combinations of columns?

23.6. Approximately how much estimation error would you expect for the condition on *WageIncome* assuming that the optimizer used a recently constructed equal height histogram with 10 bins?

24. The following questions involve a SELECT statement and database with the following facts.
    - Individual, non-clustering indexes exist on the *Age* and *HighestDegree* columns in the Census table.
    - The selectivity estimate of the condition, `HighestDegree = 'HS Graduate'`, is 0.30.
    - The selectivity estimate of the condition, `Age > 60`, is 0.20.
    - The actual selectivity of the condition, `Age > 60 AND HighestDegree = 'HS Graduate'`, is 0.10.

    24.1. What would the Oracle optimizer estimate as the selectivity of the joint condition on *Age* and *HighestDegree* when only using statistics on individual columns?

    24.2. What access method would Oracle likely choose for the *Census* table if no other conditions in the query involve the *Census* table?

    24.3. What access method should Oracle choose for the *Census* table if no other conditions in the query involve the *Census* table

    24.4. Should you use a hint to force Oracle to combine indexes on *Age* and *HighestDegree*?

    24.5. Approximately how much estimation error would you expect for the condition on *Age* assuming that the optimizer used a recently constructed equal height histogram with 20 bins?

25. Create a columnstore index for the following row group of the *Student* table. Show compression and sorting using Figure 8.20 as a guide.

Sample Row Group of Student Table

StdNo	StdMajor	StdGPA
5511	ISMG	3.2
5522	FIN	2.7
5533	FIN	3.2
5544	MKTG	4.0
5555	ISMG	3.5
5566	ISMG	3.2

REFERENCES FOR FURTHER STUDY

The subject of physical database design can be much more detailed and mathematical than described in this chapter. For a more detailed description of file structures and physical database design, consult computer science books such as Elmasri and Navathe (2017) and Teorey (2005). Abadi, Madden, and Hachem (2008) describe performance advantages of columnstores. For detailed tutorials about query optimization, consult Chaudhuri (1998), Jarke and Koch (1984) and Mannino, Chu, and Sager (1988). Finkelstein, Schkolnick, and Tiberio (1988) describe DBDSGN, an index selection tool for SQL/DS, an IBM relational DBMS. Chaudhuri. and Narasayya (1997, 2001) describe tools for index selection and statistics management for Microsoft SQL Server. Shasha and Bonnet (2003) provide more details about physical database design decisions. For details about the Oracle SQL Access Advisor and SQL Tuning Advisor, you should consult the Oracle online documentation.

part five

Application Development with Relational Databases

Part 5 provides a foundation for building database applications through conceptual background and skills for advanced query formulation, specification of data requirements for data entry forms and reports, and coding triggers and stored procedures. Chapter 9 extends query formulation skills by explaining advanced table matching problems using additional parts of the SQL SELECT statement. Chapter 10 describes motivation, definition, and usage of relational views along with specification of data requirements for data entry forms and reports. Chapter 11 presents concepts of database programming languages and coding practices for stored procedures and triggers in Oracle PL/SQL to support customization of database applications.

chapter

9

Advanced Query Formulation with SQL

Learning Objectives

This chapter extends your query formulation skills by explaining advanced table matching problems involving the outer join, difference, and division operators. To explain advanced matching problems, this chapter provides problem-solving guidelines and demonstrates additional parts of the SELECT statement. To help interpret query results involving null values, this chapter explains subtle effects of null values. To support problems involving hierarchically structured data, this chapter depicts extensions to the SELECT statement for hierarchical queries. After this chapter, you should have acquired the following knowledge and skills:

- Recognize Type I nested queries for joins and understand the associated conceptual evaluation process
- Recognize Type II nested queries and understand the associated conceptual evaluation process
- Recognize problems involving the outer join, difference, and division operators
- Adapt example SQL statements to matching problems involving the outer join, difference, and division operators
- Understand the effect of null values on conditions, aggregate calculations, and grouping
- Formulate problems involving hierarchically structured data using the Oracle proprietary notation and the SQL standard notation

OVERVIEW

As the first chapter in Part 5 of the textbook, this chapter builds on the query formulation foundation provided in Chapter 4. Most importantly, you learned an important subset of the SELECT statement and usage of the SELECT statement for problems involving joins and grouping. This chapter extends your knowledge of query formulation to advanced matching problems. To solve these advanced matching problems, additional parts of the SELECT statement are introduced.

This chapter continues with the learning approaches of Chapter 4: provide many examples to imitate and problem-solving guidelines to help you reason through difficult problems. You first will learn to formulate problems involving the outer join operator using new keywords in the FROM clause. Next you will learn to recognize nested queries and apply them to formulate problems involving the join and difference operators. Then you will learn to recognize problems involving the division operator and formulate them using the GROUP BY clause, nested queries in the HAVING clause, and the COUNT function. You will then learn the effect of null values on simple conditions, compound conditions with logical operators, aggregate calculations, and grouping. Finally, you will learn about problems involving hierarchically structured data and SQL extensions (both standard and proprietary) to formulate queries.

The presentation in this chapter covers additional features in SQL:2016, especially features not part of SQL-92. All examples execute in recent versions of Microsoft Access (2002 and beyond) and Oracle (9i and beyond) except where noted.

9.1 OUTER JOIN PROBLEMS

One of the powerful but sometimes confusing aspects of the SELECT statement is the number of ways to express a join. In Chapter 4, you formulated joins using the cross product style and the join operator style. In the cross product style, you list the tables in the FROM clause and the join conditions in the WHERE clause. In the join operator style, you write join operations directly in the FROM clause using the INNER JOIN and ON keywords.

The major advantage of the join operator style is that problems involving the outer join operator can be formulated. Outer join problems cannot be formulated with the cross product style except with proprietary SQL extensions. This section demonstrates the join operator style for outer join problems and combinations of inner and outer joins. In addition, the proprietary outer join extension of older Oracle versions (8i and previous versions) is shown in Appendix 9C. For your reference, the relationship diagram of the university database is repeated from Chapter 4 (see Figure 9.1).

9.1.1 SQL Support for Outer Join Problems

One-Sided Outer Join
an operator that generates the join result (the matching rows) plus the non-matching rows from one of the input tables. SQL supports the one-sided outer join operator through the LEFT JOIN and RIGHT JOIN keywords.

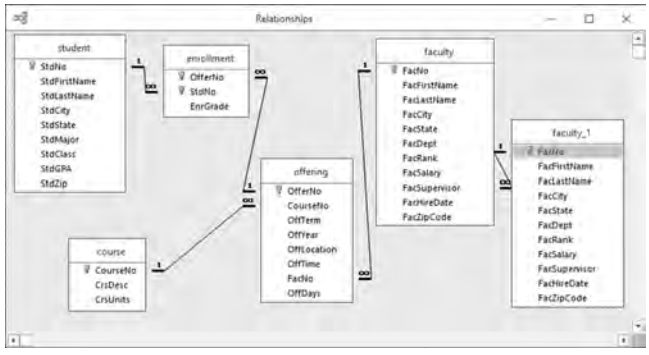
A join between two tables generates a table with the rows that match on the join column(s). The outer join operator generates the join result (the matching rows) plus the non-matching rows. A **one-sided outer join** generates a new table with the matching rows plus the non-matching rows from one of the tables. For example, it can be useful to see all offerings listed in the output even if an offering does not have an assigned faculty.

SQL uses the LEFT JOIN and RIGHT JOIN keywords[1] to specify a one-sided outer join. The LEFT JOIN keyword creates a result table containing the matching rows and the non-matching rows of the left table. The RIGHT JOIN keyword creates a result table containing the matching rows and the non-matching rows of the right table. Thus, the result of a one-sided outer join depends on the direction (RIGHT or LEFT) and the position of the table names. Examples 9.1 and 9.2 demonstrate one-sided outer joins using both the LEFT and RIGHT keywords. The result rows with blank values for certain columns are non-matched rows.

Full Outer Join
an operator that generates the join result (the matching rows) plus the nonmatching rows from both input tables. SQL supports the full outer join operator through the FULL JOIN keyword.

A **full outer join** generates a table with the matching rows plus the nonmatching rows from both input tables. Typically, a full outer join is used to combine two similar but not union compatible tables. For example, the *Student* and *Faculty* tables are similar because they contain information about university people. However, they are not union compatible. They have common columns such as first name, last name, and city but also unique columns such as GPA and salary. Occasionally, you will need to write

FIGURE 9.1
Relationship Window for the University Database



[1] The full SQL keywords are LEFT OUTER JOIN and RIGHT OUTER JOIN. The SQL:2016 standard and most DBMSs allow omission of the OUTER keyword.

Example 9.1 (Access)

One-Sided Outer Join using LEFT JOIN

For offerings beginning with "IS" in the associated course number, retrieve the offer number, the course number, the faculty number, and the faculty name. Include an offering in the result even if the faculty is not yet assigned. The Oracle[2] counterpart of this example uses % instead of * as the wild card character.

```
SELECT OfferNo, CourseNo, Offering.FacNo, Faculty.FacNo,
       FacFirstName, FacLastName
 FROM Offering LEFT JOIN Faculty
   ON Offering.FacNo = Faculty.FacNo
 WHERE CourseNo LIKE 'IS*'
```

OfferNo	CourseNo	Offering.FacNo	Faculty.FacNo	FacFirstName	FacLastName
1111	IS320				
2222	IS460				
1234	IS320	098-76-5432	098-76-5432	LEONARD	VINCE
3333	IS320	098-76-5432	098-76-5432	LEONARD	VINCE
4321	IS320	098-76-5432	098-76-5432	LEONARD	VINCE
4444	IS320	543-21-0987	543-21-0987	VICTORIA	EMMANUEL
8888	IS320	654-32-1098	654-32-1098	LEONARD	FIBON
9876	IS460	654-32-1098	654-32-1098	LEONARD	FIBON
5679	IS480	876-54-3210	876-54-3210	CRISTOPHER	COLAN
5678	IS480	987-65-4321	987-65-4321	JULIA	MILLS

Example 9.2 (Access)

One-Sided Outer Join using RIGHT JOIN

For offerings beginning with "IS" in the associated course number, retrieve the offer number, the course number, the faculty number, and the faculty name. Include an offering in the result even if the faculty is not yet assigned. The result is identical to Example 9.1. The Oracle counterpart of this example uses % instead of * as the wild card character.

```
SELECT OfferNo, CourseNo, Offering.FacNo, Faculty.FacNo,
       FacFirstName, FacLastName
 FROM Faculty RIGHT JOIN Offering
   ON Offering.FacNo = Faculty.FacNo
 WHERE CourseNo LIKE 'IS*'
```

a query that combines both tables. For example, a full outer join should be used to find all details about university people within a certain city.

SQL:2016 provides the FULL JOIN keyword as demonstrated in Example 9.3. Note the null values in both halves (*Student* and *Faculty*) of the result.

Some DBMSs (such as Microsoft Access) do not directly support the full outer join operator. In these systems, a full outer join is formulated by taking the union of

[2] Appendix 9C shows the proprietary notation used in Oracle 8i for outer joins. Although the proprietary notation still works in the latest Oracle version, the SQL standard notation is preferred.

Example 9.3 (SQL:2016 and Oracle 9i and beyond)

Full Outer Join

Combine the *Faculty* and *Student* tables using a full outer join. List the person number (faculty or student number), the name (first and last), the salary (faculty only), and the GPA (students only) in the result. This SQL statement does not execute in Microsoft Access.

```
SELECT FacNo, FacFirstName, FacLastName, FacSalary,
       StdNo, StdFirstName, StdLastName, StdGPA
 FROM Faculty FULL JOIN Student
   ON Student.StdNo = Faculty.FacNo
```

FacNo	FacFirstName	FacLastName	FacSalary	StdNo	StdFirstName	StdLastName	StdGPA
				123456789	HOMER	WELLS	3.0
				124567890	BOB	NORBERT	2.7
				234567890	CANDY	KENDALL	3.5
				345678901	WALLY	KENDALL	2.8
				456789012	JOE	ESTRADA	3.2
				567890123	MARIAH	DODGE	3.6
				678901234	TESS	DODGE	3.3
				789012345	ROBERTO	MORALES	2.5
				890123456	LUKE	BRAZZI	2.2
				901234567	WILLIAM	PILGRIM	3.8
098765432	LEONARD	VINCE	35000				
543210987	VICTORIA	EMMANUEL	120000				
654321098	LEONARD	FIBON	70000				
765432109	NICKI	MACON	65000				
876543210	CRISTOPHER	COLAN	40000	876543210	CRISTOPHER	COLAN	4.0
987654321	JULIA	MILLS	75000				

two one-sided outer joins using the steps shown below. The SELECT statement implementing these steps is shown in Example 9.4.

1. Construct a right join of *Faculty* and *Student* (non-matched rows of *Student*).
2. Construct a left join of *Faculty* and *Student* (non-matched rows of *Faculty*).
3. Construct a union of these two temporary tables. Remember when using the UNION operator, the two table arguments must be "union compatible": each corresponding column from both tables must have compatible data types. Otherwise, the UNION operator will not work as expected.

9.1.2 Mixing Inner and Outer Joins

Inner and outer joins can be mixed as demonstrated in Examples 9.5 and 9.6. For readability, it is generally preferred to use the join operator style rather than to mix the join operator and cross product styles.

In most queries, you can change the order of inner and outer joins without any problem. The inner join operator is associative meaning that the order of inner join operations does not matter. The same result always occurs when changing the order of inner join operations. Although the one-sided outer join is not associative, the order

Example 9.4 (Access)

Full Outer Join Using a Union of Two One-Sided Outer Joins

Combine the *Faculty* and *Student* tables using a full outer join. List the person number (faculty or student number), the name (first and last), the salary (faculty only), and the GPA (students only) in the result. The result is identical to Example 9.3. This statement executes in Oracle although the FULL JOIN syntax as demonstrated in Example 9.3 is preferred for Oracle.

```
SELECT FacNo, FacFirstName, FacLastName, FacSalary,
       StdNo, StdFirstName, StdLastName, StdGPA
 FROM Faculty RIGHT JOIN Student
   ON Student.StdNo = Faculty.FacNo
      UNION
SELECT FacNo, FacFirstName, FacLastName, FacSalary,
       StdNo, StdFirstName, StdLastName, StdGPA
 FROM Faculty LEFT JOIN Student
   ON Student.StdNo = Faculty.FacNo
```

Example 9.5 (Access)

Mixing a One-Sided Outer Join and an Inner Join

Combine columns from the *Faculty*, *Offering*, and *Course* tables for information systems courses (IS in the beginning of the course number) offered in 2017. Include a row in the result even if there is not an assigned instructor. The Oracle counterpart of this example uses % instead of * as the wild card character.

```
SELECT OfferNo, Offering.CourseNo, OffTerm, CrsDesc,
       Faculty.FacNo, FacFirstName, FacLastName
 FROM ( Faculty RIGHT JOIN Offering
   ON Offering.FacNo = Faculty.FacNo )
  INNER JOIN Course
   ON Course.CourseNo = Offering.CourseNo
 WHERE Course.CourseNo LIKE 'IS*' AND OffYear = 2017
```

OfferNo	CourseNo	OffTerm	CrsDesc	FacNo	FacFirstName	FacLastName
1111	IS320	SUMMER	FUNDAMENTALS OF BUSINESS PROGRAMMING			
3333	IS320	SPRING	FUNDAMENTALS OF BUSINESS PROGRAMMING	098-76-5432	LEONARD	VINCE
4444	IS320	WINTER	FUNDAMENTALS OF BUSINESS PROGRAMMING	543-21-0987	VICTORIA	EMMANUEL
5678	IS480	WINTER	FUNDAMENTALS OF DATABASE MANAGEMENT	987-65-4321	JULIA	MILLS
5679	IS480	SPRING	FUNDAMENTALS OF DATABASE MANAGEMENT	876-54-3210	CRISTOPHER	COLAN
8888	IS320	SUMMER	FUNDAMENTALS OF BUSINESS PROGRAMMING	654-32-1098	LEONARD	FIBON
9876	IS460	SPRING	SYSTEMS ANALYSIS	654-32-1098	LEONARD	FIBON

of operations does not matter in most queries. For example, Example 9.6a returns the same results as Example 9.6.

Queries with Ambiguous Combinations of Joins and Outer Joins Some queries combining inner and outer joins are ambiguous. In ambiguous queries, different orders of operations may produce different results. A query is ambiguous if a non-preserved table (table with only matching rows in the result) in a one-sided outer

Example 9.6 (Access)

Mixing a One-Sided Outer Join and Two Inner Joins

List the rows of the *Offering* table where there is at least one student enrolled, in addition to the requirements of Example 9.5. Remove duplicate rows when there is more than one student enrolled in an offering. The Oracle counterpart of this example uses % instead of * as the wild card character.

```
SELECT DISTINCT Offering.OfferNo, Offering.CourseNo,
       OffTerm, CrsDesc, Faculty.FacNo, FacFirstName,
       FacLastName
 FROM ( ( Faculty RIGHT JOIN Offering
   ON Offering.FacNo = Faculty.FacNo )
     INNER JOIN Course
   ON Course.CourseNo = Offering.CourseNo )
     INNER JOIN Enrollment
   ON Offering.OfferNo = Enrollment.OfferNo
 WHERE Offering.CourseNo LIKE 'IS*' AND OffYear = 2017
```

OfferNo	CourseNo	OffTerm	CrsDesc	FacNo	FacFirstName	FacLastName
5678	IS480	WINTER	FUNDAMENTALS OF DATABASE MANAGEMENT	987-65-4321	JULIA	MILLS
5679	IS480	SPRING	FUNDAMENTALS OF DATABASE MANAGEMENT	876-54-3210	CRISTOPHER	COLAN
9876	IS460	SPRING	SYSTEMS ANALYSIS	654-32-1098	LEONARD	FIBON

Example 9.6a (Access)

Mixing a One-Sided Outer Join and Two Inner Joins with the Outer Join Performed Last

List the rows of the *Offering* table where there is at least one student enrolled, in addition to the requirements of Example 9.5. Remove duplicate rows when there is more than one student enrolled in an offering. The Oracle counterpart of this example uses % instead of * as the wild card character. The result is identical to Example 9.6.

```
SELECT DISTINCT Offering.OfferNo, Offering.CourseNo,
       OffTerm, CrsDesc, Faculty.FacNo, FacFirstName,
       FacLastName
 FROM Faculty RIGHT JOIN
  ( ( Offering INNER JOIN Course
   ON Course.CourseNo = Offering.CourseNo )
     INNER JOIN Enrollment
   ON Offering.OfferNo = Enrollment.OfferNo )
   ON Offering.FacNo = Faculty.FacNo
 WHERE Offering.CourseNo LIKE 'IS*' AND OffYear = 2017
```

join is involved in another join or outer join operation. In Examples 9.6 and 9.6a, the *Offering* table is preserved (both matching and non-matching rows) so no ambiguity exists. The *Offering* table can participate in other join and outer join operations without causing ambiguity.

Ambiguity occurs if the direction of the one-sided outer join is reversed to preserve the *Faculty* table rather than the *Offering* table. In Example 9.6b, the result is

different than Example 9.6c if the *Faculty* table has unmatched rows. Example 9.6b eliminates the unmatched rows because the outer join is performed before the inner join. Example 9.6c preserves the unmatched *Faculty* rows because the outer join is performed after the inner join. Microsoft Access will not execute either query. Oracle executes both examples but returns different results if the *Faculty* table contains unmatched rows.

Example 9.6b

Ambiguous Query Mixing a One-Sided Outer Join and Two Inner Joins with the Outer Join Performed First (Oracle)

This query eliminates the non-matching *Faculty* rows because the outer join is performed first. This statement cannot be saved nor executed in Microsoft Access because of the ambiguity rule.

```
SELECT Offering.OfferNo, Offering.CourseNo,
       OffTerm, CrsDesc, Faculty.FacNo, FacFirstName,
       FacLastName
 FROM (Faculty LEFT JOIN Offering
   ON Offering.FacNo = Faculty.FacNo)
 INNER JOIN Course ON Course.CourseNo = Offering.CourseNo
```

Example 9.6c

Ambiguous Query Mixing a One-Sided Outer Join and Two Inner Joins with the Outer Join Performed Last (Oracle)

This query preserves the non-matching *Faculty* rows because the outer join is performed last. This query cannot be saved nor executed in Microsoft Access because of the ambiguity rule.

```
SELECT Offering.OfferNo, Offering.CourseNo,
       OffTerm, CrsDesc, Faculty.FacNo, FacFirstName,
       FacLastName
 FROM (Offering INNER JOIN Course
   ON Course.CourseNo = Offering.CourseNo)
  RIGHT JOIN Faculty ON Offering.FacNo = Faculty.FacNo
```

Ambiguous Query: A query is ambiguous if a non-preserved table (table with only matching rows in the result) in a one-sided outer join is involved in another join or outer join operation. The result of an ambiguous query may depend on the order of joins and one-sided join operations in the FROM clause.

Despite the possibility for ambiguous queries, they are not common because the non-preserved table is typically not involved in other operations. The parent table is typically the non-preserved table. The child table is usually preserved and involved in other operations. An ambiguous query may indicate an error in query formulation rather than a valid formulation to address a legitimate business request.

9.2 UNDERSTANDING NESTED QUERIES

A nested query or subquery is a query (SELECT statement) inside a query. A nested query typically appears as part of a condition in the WHERE or HAVING clauses. Nested queries also can be used in the FROM clause. Nested queries can be used like a procedure (Type I nested query) in which the nested query is executed one time or like a loop (Type II nested query) in which the nested query is executed repeatedly. This section demonstrates examples of both kinds of nested queries and explains problems in which they can be applied.

9.2.1 Type I Nested Queries

Type I Nested Query
a nested query in which the inner query does not reference any tables used in the outer query. A Type I nested query executes one time. Type I nested queries can be used for some join problems and some difference problems.

Type I nested queries are like procedures in a programming language. A Type I nested query evaluates *one* time and produces a table. The nested (or inner) query does not reference the outer query. Using the IN comparison operator, a Type I nested query can be used to express a join. In Example 9.7, the nested query on the *Enrollment* table generates a list of qualifying student number values. A row is selected in the outer query on *Student* if the student number is an element of the nested query result.

Type I nested queries should be used only when the result does not contain any columns from the tables in the nested query. In Example 9.7, no columns from the *Enrollment* table are used in the result. In Example 9.8, the join between the *Student* and *Enrollment* tables cannot be performed with a Type I nested query because *EnrGrade* appears in the result.

It is possible to have multiple levels of nested queries although this practice is not encouraged because the statements can be difficult to read. In a nested query, you can have another nested query using the IN comparison operator in the WHERE clause. In Example 9.9, the nested query on the *Offering* table has a nested query on the *Faculty* table. No *Faculty* columns are needed in the main query or in the nested query on *Offering*.

Example 9.7

Using a Type I Nested Query to Perform a Join

List the student number, name, and major of students who have a high grade (≥ 3.5) in a course offering.

```
SELECT StdNo, StdFirstName, StdLastName, StdMajor
 FROM Student
 WHERE Student.StdNo IN
 ( SELECT StdNo FROM Enrollment
   WHERE EnrGrade >= 3.5 )
```

StdNo	StdFirstName	StdLastName	StdMajor
123-45-6789	HOMER	WELLS	IS
124-56-7890	BOB	NORBERT	FIN
234-56-7890	CANDY	KENDALL	ACCT
567-89-0123	MARIAH	DODGE	IS
789-01-2345	ROBERTO	MORALES	FIN
890-12-3456	LUKE	BRAZZI	IS
901-23-4567	WILLIAM	PILGRIM	IS

Example 9.8

Combining a Type I Nested Query and the Join Operator Style

Retrieve the name, city, and grade of students who have a high grade (≥ 3.5) in a course offered in fall 2016.

```
SELECT StdFirstName, StdLastName, StdCity, EnrGrade
 FROM Student INNER JOIN Enrollment
      ON Student.StdNo = Enrollment.StdNo
 WHERE EnrGrade >= 3.5 AND Enrollment.OfferNo IN
 ( SELECT OfferNo FROM Offering
   WHERE OffTerm = 'Fall' AND OffYear = 2016 )
```

StdFirstName	StdLastName	StdCity	EnrGrade
CANDY	KENDALL	TACOMA	3.5
MARIAH	DODGE	SEATTLE	3.8
HOMER	WELLS	SEATTLE	3.5
ROBERTO	MORALES	SEATTLE	3.5

Example 9.9

Using a Type I Nested Query inside Another Type I Nested Query

Retrieve the name, city, and grade of students who have a high grade (≥ 3.5) in a course offered in fall 2016 taught by Leonard Vince.

```
SELECT StdFirstName, StdLastName, StdCity, EnrGrade
 FROM Student, Enrollment
 WHERE Student.StdNo = Enrollment.StdNo
   AND EnrGrade >= 3.5 AND Enrollment.OfferNo IN
   ( SELECT OfferNo FROM Offering
      WHERE OffTerm = 'Fall' AND OffYear = 2016
        AND FacNo IN
        ( SELECT FacNo FROM Faculty
           WHERE FacFirstName = 'Leonard'
             AND FacLastName = 'Vince' ) )
```

StdFirstName	StdLastName	StdCity	EnrGrade
CANDY	KENDALL	TACOMA	3.5
MARIAH	DODGE	SEATTLE	3.8
HOMER	WELLS	SEATTLE	3.5
ROBERTO	MORALES	SEATTLE	3.5

The Type I style gives a visual feel to a query. You can visualize a Type I subquery as navigating between tables. Visit the table in the subquery to collect join values that can be used to select rows from the table in the outer query. The use of Type I nested queries is largely a matter of preference. Even if you do not prefer this join style, you should be prepared to interpret queries written by others with Type I nested queries.

DELETE and UPDATE statements provide another use of a Type I nested query. A Type I nested query is useful in a DELETE statement with conditions referencing related tables, as demonstrated in Example 9.10a. Using a Type I nested query is the standard way to reference related tables in DELETE statements. Similarly, a Type I nested query is useful in an UPDATE statement with conditions referencing related tables as shown in Example 9.11a. Chapter 4 demonstrated the join operator style inside DELETE and UPDATE statements, a proprietary extension of Microsoft Access. For your reference, Examples 9.10b and 9.11b show equivalent DELETE and UPDATE statements using the join operator style.

Example 9.10a

DELETE Statement Using a Type I Nested Query

Delete offerings taught by Leonard Vince. Three Offering rows are deleted. In addition, this statement deletes related rows in the *Enrollment* table because the ON DELETE clause is set to CASCADE.

```
DELETE FROM Offering
 WHERE Offering.FacNo IN
  ( SELECT FacNo FROM Faculty
     WHERE FacFirstName = 'Leonard'
       AND FacLastName = 'Vince' )
```

Example 9.10b (Access only)

DELETE Statement Using an INNER JOIN Operation

Delete offerings taught by Leonard Vince. Three *Offering* rows are deleted. In addition, this statement deletes related rows in the *Enrollment* table because the ON DELETE clause is set to CASCADE.

```
DELETE Offering.*
 FROM Offering INNER JOIN Faculty
   ON Offering.FacNo = Faculty.FacNo
 WHERE FacFirstName = 'Leonard'
   AND FacLastName = 'Vince'
```

Example 9.11a

UPDATE Statement Using a Type I Nested Query

Update the location of offerings taught by Leonard Fibon in 2017 to BLM412. Two *Offering* rows are updated.

```
UPDATE Offering SET OffLocation = 'BLM412'
 WHERE OffYear = 2017 AND FacNo IN
 ( SELECT FacNo FROM Faculty
    WHERE FacFirstName = 'LEONARD'
      AND FacLastName = 'FIBON')
```

Example 9.11b (Access)

UPDATE Statement Using an INNER JOIN Operation

Update the location of offerings taught by Leonard Fibon in 2017 to BLM412. Two *Offering* rows are updated.

```
UPDATE Offering INNER JOIN Faculty
    ON Offering.FacNo = Faculty.FacNo
 SET OffLocation = 'BLM412'
 WHERE OffYear = 2017 AND FacFirstName = 'LEONARD'
   AND FacLastName = 'FIBON'
```

9.2.2 Limited SQL Formulations for Difference Problems

Difference Problems
problem statements involving the difference operator often have a *not* relating two nouns in a sentence. For example, "students who are not faculty" and "employees who are not customers" are problem statements involving a difference operation.

You should recall from Chapter 3 that the difference operator combines tables by finding the rows of a first table not in a second table. A typical usage of the difference operator is to combine two tables with some similar columns but not entirely union compatible. For example, you may want to find faculty who are not students. Although the *Faculty* and *Student* tables contain some compatible columns, the tables are not union compatible. The placement of the word *not* in the problem statement indicates that the result contains rows only in the *Faculty* table, not in the *Student* table. This requirement involves a difference operation.

Some difference problems can be formulated using a Type I nested query with the NOT IN operator. As long as the comparison among tables involves a single column, a Type I nested query can be used. In Example 9.12, a Type I nested query can be used because the comparison only involves a single column from the *Faculty* table (*FacNo*).

Example 9.12

Using a Type I Nested Query for a Difference Problem

Retrieve the faculty number, name (first and last), department, and salary of faculty who are not students.

```
SELECT FacNo, FacFirstName, FacLastName, FacDept, FacSalary
 FROM Faculty
 WHERE FacNo NOT IN
 ( SELECT StdNo FROM Student )
```

FacNo	FacFirstName	FacLastName	FacDept	FacSalary
098-76-5432	LEONARD	VINCE	MS	$35,000.00
543-21-0987	VICTORIA	EMMANUEL	MS	$120,000.00
654-32-1098	LEONARD	FIBON	MS	$70,000.00
765-43-2109	NICKI	MACON	FIN	$65,000.00
987-65-4321	JULIA	MILLS	FIN	$75,000.00

Another formulation approach for some difference problems involves a one-sided outer join operation to generate a table with only non-matching rows. The IS NULL comparison operator can remove rows that match, as demonstrated in Example 9.13.

However, this formulation cannot be used with conditions to test on the non-preserved table (*Student* in Example 9.13) other than the IS NULL condition on the primary key column. If there are conditions to test on the *Student* table (such as on student class), another SQL formulation approach must be used.

Example 9.13

One-Sided Outer Join with Only Non-matching Rows

Retrieve the faculty number, name, department, and salary of faculty who are *not* students. The result is identical to Example 9.12.

```
SELECT FacNo, FacFirstName, FacLastName, FacSalary
 FROM Faculty LEFT JOIN Student
   ON Faculty.FacNo = Student.StdNo
 WHERE Student.StdNo IS NULL
```

Although SQL:2016 does have a difference operator (the EXCEPT keyword), it is sometimes not convenient because only the common columns can be shown in the result. Example 9.14 does not provide the same result as Example 9.12 because the columns unique to the *Faculty* table (*FacDept* and *FacSalary*) are not in the result. Another query that uses the first result must be formulated to retrieve the unique *Faculty* columns.

Example 9.14 (Oracle)

Difference Query

Show faculty who are *not* students (pure faculty). Only show the common columns in the result. Note that Microsoft Access does not support the EXCEPT keyword. Oracle uses the MINUS keyword instead of EXCEPT. The result is identical to Example 9.12 except for *FacCity* and *FacState* instead of *FacDept* and *FacSalary*.

```
SELECT FacNo AS PerNo, FacFirstName AS FirstName,
       FacLastName AS LastName, FacCity AS City,
       FacState AS State
 FROM Faculty
   MINUS
SELECT StdNo AS PerNo, StdFirstName AS FirstName,
       StdLastName AS LastName, StdCity AS City,
       StdState AS State
 FROM Student
```

Difference Problems Cannot Be Solved with Inequality Joins It is important to note that difference problems such as Example 9.12 cannot be solved with a join alone. Example 9.12 requires that every row of the *Student* table be searched to select a faculty row. In contrast, a join selects a faculty row when the first matching student row is found. To contrast difference and join problems, you should examine Example 9.15. Although it looks correct, it does not provide the desired result. Every faculty row will be in the result because there is at least one student row that does not match every faculty row.

Example 9.15

Inequality Join

Erroneous formulation for the problem "Retrieve the faculty number, name (first and last), and rank of faculty who are *not* students." The result contains all faculty rows.

```
SELECT DISTINCT FacNo, FacFirstName, FacLastName, FacRank
 FROM Faculty, Student
 WHERE Student.StdNo <> Faculty.FacNo
```

To understand Example 9.15, you can use the conceptual evaluation process discussed in Chapter 4 (Section 4.3). The result tables show the cross product (Table 9-3) of Tables 9-1 and 9-2 followed by the rows that satisfy the WHERE condition (Table 9-4). Notice that only one row of the cross product is deleted. The final result (Table 9-5) contains all rows of Table 9-2.

TABLE 9-1
Sample *Student* Table

StdNo	StdFirstName	StdLastName	StdMajor
123-45-6789	HOMER	WELLS	IS
124-56-7890	BOB	NORBERT	FIN
876-54-3210	CHRISTOPHERR	COLAN	IS

TABLE 9-2
Sample *Faculty* Table

FacNo	FacFirstName	FacLastName	FacRank
098-76-5432	LEONARD	VINCE	ASST
543-21-0987	VICTORIA	EMMANUEL	PROF
876-54-3210	CRISTOPHER	COLAN	ASST

FacNo	FacFirstName	FacLastName	FacRank	StdNo	StdFirstName	StdLastName	StdMajor
098-76-5432	LEONARD	VINCE	ASST	123-45-6789	HOMER	WELLS	IS
098-76-5432	LEONARD	VINCE	ASST	124-56-7890	BOB	NORBERT	FIN
098-76-5432	LEONARD	VINCE	ASST	876-54-3210	CRISTOPHER	COLAN	IS
543-21-0987	VICTORIA	EMMANUEL	PROF	123-45-6789	HOMER	WELLS	IS
543-21-0987	VICTORIA	EMMANUEL	PROF	124-56-7890	BOB	NORBERT	FIN
543-21-0987	VICTORIA	EMMANUEL	PROF	876-54-3210	CRISTOPHER	COLAN	IS
876-54-3210	CRISTOPHER	COLAN	ASST	123-45-6789	HOMER	WELLS	IS
876-54-3210	CRISTOPHER	COLAN	ASST	124-56-7890	BOB	NORBERT	FIN
876-54-3210	CRISTOPHER	COLAN	ASST	876-54-3210	CRISTOPHER	COLAN	IS

TABLE 9-3
Cross Product of the Sample *Student* and *Faculty* Tables

FacNo	FacFirstName	FacLastName	FacRank	StdNo	StdFirstName	StdLastName	StdMajor
098-76-5432	LEONARD	VINCE	ASST	123-45-6789	HOMER	WELLS	IS
098-76-5432	LEONARD	VINCE	ASST	124-56-7890	BOB	NORBERT	FIN
098-76-5432	LEONARD	VINCE	ASST	876-54-3210	CRISTOPHER	COLAN	IS
543-21-0987	VICTORIA	EMMANUEL	PROF	123-45-6789	HOMER	WELLS	IS
543-21-0987	VICTORIA	EMMANUEL	PROF	124-56-7890	BOB	NORBERT	FIN
543-21-0987	VICTORIA	EMMANUEL	PROF	876-54-3210	CRISTOPHER	COLAN	IS
876-54-3210	CRISTOPHER	COLAN	ASST	123-45-6789	HOMER	WELLS	IS
876-54-3210	CRISTOPHER	COLAN	ASST	124-56-7890	BOB	NORBERT	FIN

TABLE 9-4
Restriction of Table 9-3 to Eliminate Matching Rows

TABLE 9-5
Projection of Table 9-4 to Eliminate *Student* Columns

FacNo	FacFirstName	FacLastName	FacRank
098-76-5432	LEONARD	VINCE	ASST
543-21-0987	VICTORIA	EMMANUEL	PROF
876-54-3210	CRISTOPHER	COLAN	ASST

Summary of Limited Formulations for Difference Problems This section has discussed three SQL formulations for difference problems. Each formulation has limitations as noted in Table 9-6. In practice, the one-sided outer join approach is the most restrictive as many problems involve conditions on the excluded table. Section 9.2.3 presents a more general formulation without the restrictions noted in Table 9-6.

9.2.3 Using Type II Nested Queries for Difference Problems

Although Type II nested queries provide a more general solution for difference problems, they are conceptually more complex than Type I nested queries. Type II nested queries have two distinguishing features. First, Type II nested queries reference one or more columns from an outer query. Type II nested queries are sometimes known as correlated subqueries because they reference columns used in outer queries. In contrast, Type I nested queries are not correlated with outer queries. In Example 9.16, the nested query references the *Faculty* table used in the outer query in the comparison `Student.StdNo = Faculty.FacNo`.

TABLE 9-6
Limitations of SQL Formulations for Difference Problems

SQL Formulation	Limitations
Type I nested query with the NOT IN operator	Only one column for comparing rows of the two tables
One-sided outer join with an IS NULL condition	No conditions (except the IS NULL condition) on the non-preserved table
Difference operation using the EXCEPT or MINUS keywords	Result must contain only union-compatible columns

Example 9.16

Using a Type II Nested Query for a Difference Problem

Retrieve the faculty number, the name (first and last), the department, and the salary of faculty who are *not* students.

```
SELECT FacNo, FacFirstName, FacLastName, FacDept, FacSalary
 FROM Faculty
 WHERE NOT EXISTS
 ( SELECT * FROM Student
      WHERE Student.StdNo = Faculty.FacNo )
```

FacNo	FacFirstName	FacLastName	FacDept	FacSalary
098-76-5432	LEONARD	VINCE	MS	$35,000.00
543-21-0987	VICTORIA	EMMANUEL	MS	$120,000.00
654-32-1098	LEONARD	FIBON	MS	$70,000.00
765-43-2109	NICKI	MACON	FIN	$65,000.00
987-65-4321	JULIA	MILLS	FIN	$75,000.00

The second distinguishing feature of Type II nested queries involves execution. A **Type II nested query** executes one time for *each* row in the outer query. In this sense, a Type II nested query is similar to a nested loop that executes one time for each execution of the outer loop. In each execution of the inner loop, variables used in the outer loop are used in the inner loop. In other words, the inner query uses one or more values from the outer query in each execution.

Type II Nested Query
a nested query in which the inner query references a table used in the outer query. Because a Type II nested query executes for each row of its outer query, Type II nested queries are more difficult to understand and execute than Type I nested queries.

To help you understand Example 9.16, Table 9-9 traces the execution of the nested query using Tables 9-7 and 9-8. The EXISTS operator is true if the nested query returns one or more rows. In contrast, the **NOT EXISTS operator** is true if the nested query returns 0 rows. Thus, a faculty row in the outer query is selected only if there are no matching student rows in the nested query. For example, the first two rows in Table 9-7 are selected because there are no matching rows in Table 9-8. The third row is *not* selected because the nested query returns one row (the third row of Table 9-7).

NOT EXISTS operator
a table comparison operator often used with Type II nested queries. NOT EXISTS is true for a row in an outer query if the inner query returns no rows and false if the inner query returns one or more rows.

Example 9.17 shows another formulation that clarifies the meaning of the NOT EXISTS operator. Here, a faculty row is selected if the number of rows in the nested query is 0. Using the sample tables (Tables 9-7 and 9-8), the nested query result is 0 for the first two faculty rows.

More Difficult Difference Problems More difficult difference problems combine a difference operation with join operations.

For example, consider the query to list students who took all of their information systems (IS) offerings in winter 2017 from the same instructor. The query results should include students who took only one offering as well as students who took more than one offering.

- Construct a list of students who have taken IS courses in winter 2017 (a join operation).
- Construct another list of students who have taken IS courses in winter 2017 from more than one instructor (a join operation).
- Use a difference operation (first student list minus the second student list) to produce the result.

TABLE 9-7
Sample *Faculty* Table

FacNo	FacFirstName	FacLastName	FacRank
098-76-5432	LEONARD	VINCE	ASST
543-21-0987	VICTORIA	EMMANUEL	PROF
876-54-3210	CRISTOPHER	COLAN	ASST

TABLE 9-8
Sample *Student* Table

StdNo	StdFirstName	StdLastName	StdMajor
123-45-6789	HOMER	WELLS	IS
124-56-7890	BOB	NORBERT	FIN
876-54-3210	CRISTOPHER	COLAN	IS

TABLE 9-9
Execution Trace of Nested Query in Example 9.16

FacNo	Result of subquery execution	NOT EXISTS
098-76-5432	0 rows retrieved	true
543-21-0987	0 rows retrieved	true
876-54-3210	1 row retrieved	false

Example 9.17

Using a Type II Nested Query with the COUNT Function

Retrieve the faculty number, the name, the department, and the salary of faculty who are *not* students. The result is the same as Example 9.16.

```
SELECT FacNo, FacFirstName, FacLastName, FacDept, FacSalary
 FROM Faculty
 WHERE 0 =
 ( SELECT COUNT(*) FROM Student
      WHERE Student.StdNo = Faculty.FacNo )
```

Conceptualizing a problem in this manner forces you to recognize that it involves a difference operation. If you recognize the difference operation, you can make a formulation in SQL involving a nested query (Type II with NOT EXISTS or Type I with NOT IN) or the EXCEPT keyword. Example 9.18 shows a NOT EXISTS solution in which the outer query retrieves a student row if the student does not have an offering from a *different* instructor in the inner query.

Example 9.18 (Access)

More Difficult Difference Problem Using a Type II Nested Query

List the student number and the name of students who took all of their information systems offerings in winter 2017 from the same instructor. Include students who took one or more offerings. Note that in the nested query, the columns *Enrollment.StdNo* and *Offering.FacNo* refer to the outer query.

```
SELECT DISTINCT Enrollment.StdNo, StdFirstName, StdLastName
 FROM Student, Enrollment, Offering
 WHERE Student.StdNo = Enrollment.StdNo
   AND Enrollment.OfferNo = Offering.OfferNo
   AND CourseNo LIKE 'IS*' AND OffTerm = 'Winter'
   AND OffYear = 2017 AND NOT EXISTS
   ( SELECT * FROM Enrollment E1, Offering O1
      WHERE E1.OfferNo = O1.OfferNo
        AND Enrollment.StdNo = E1.StdNo
        AND O1.CourseNo LIKE 'IS*'
        AND O1.OffYear = 2017
        AND O1.OffTerm = 'Winter'
        AND Offering.FacNo <> O1.FacNo )
```

StdNo	StdFirstName	StdLastName
123-45-6789	HOMER	WELLS
234-56-7890	CANDY	KENDALL
345-67-8901	WALLY	KENDALL
456-78-9012	JOE	ESTRADA
567-89-0123	MARIAH	DODGE

Example 9.18 (Oracle)

More Difficult Difference Problem Using a Type II Nested Query

List the student number and name of the students who took all of their information systems offerings in winter 2017 from the same instructor. Include students who took one or more offerings.

```
SELECT DISTINCT Enrollment.StdNo, StdFirstName, StdLastName
 FROM Student, Enrollment, Offering
 WHERE Student.StdNo = Enrollment.StdNo
   AND Enrollment.OfferNo = Offering.OfferNo
   AND CourseNo LIKE 'IS%' AND OffTerm = 'Winter'
   AND OffYear = 2017 AND NOT EXISTS
   ( SELECT * FROM Enrollment E1, Offering O1
      WHERE E1.OfferNo = O1.OfferNo
        AND Enrollment.StdNo = E1.StdNo
        AND O1.CourseNo LIKE 'IS%'
        AND O1.OffYear = 2017
        AND O1.OffTerm = 'Winter'
        AND Offering.FacNo <> O1.FacNo )
```

Example 9.19 shows a second example using the NOT EXISTS operator to solve a complex difference problem. Conceptually this problem involves a difference operation between two sets: the set of all faculty members and the set of faculty members teaching in the specified term. The difference operation can be implemented by selecting a faculty in the outer query list if the faculty does not teach an offering during the specified term in the inner query result.

Example 9.19

Another Difference Problem Using a Type II Nested Query

List the name (first and last) and department of faculty who are *not* teaching in winter term 2017.

```
SELECT DISTINCT FacFirstName, FacLastName, FacDept
 FROM Faculty
 WHERE NOT EXISTS
 ( SELECT * FROM Offering
   WHERE Offering.FacNo = Faculty.FacNo
     AND OffTerm = 'WINTER' AND OffYear = 2017 )
```

FacFirstName	FacLastName	FacDept
CRISTOPHER	COLAN	MS
LEONARD	FIBON	MS
LEONARD	VINCE	MS

Example 9.20 shows a third example using the NOT EXISTS operator to solve a complex difference problem. In this problem, the word *only* connecting different parts of the sentence indicates a difference operation. Conceptually this problem involves a difference operation between two sets: the set of all faculty members teaching in

winter 2017 and the set of faculty members teaching in winter 2017 in addition to teaching in another term. The difference operation can be implemented by selecting a faculty teaching in winter 2017 in the outer query if the same faculty does not teach an offering in a different term in the nested query.

Example 9.20

Another Difference Problem Using a Type II Nested Query

List the name (first and last) and department of faculty who are *only* teaching in winter term 2017.

```
SELECT DISTINCT FacFirstName, FacLastName, FacDept
 FROM Faculty F1, Offering O1
 WHERE F1.FacNo = O1.FacNo
   AND OffTerm = 'WINTER' AND OffYear = 2017
   AND NOT EXISTS
   ( SELECT * FROM Offering O2
      WHERE O2.FacNo = F1.FacNo
        AND ( OffTerm <> 'WINTER' OR OffYear <> 2017 ) )
```

FacFirstName	FacLastName	FacDept
EMMANUEL	VICTORIA	MS
MILLS	JULIA	FIN

9.2.4 Nested Queries in the FROM Clause

So far, you have seen nested queries in the WHERE clause with certain comparison operators (IN and EXISTS) as well as with traditional comparison operators when the nested query produces a single value such as the count of the number of rows. Similar to the usage in the WHERE clause, nested queries also can appear in the HAVING clause as demonstrated in the next section. Nested queries in the WHERE and the HAVING clauses have been part of SQL since its initial design.

In contrast, nested queries in the FROM clause were supported beginning with SQL:1999. The design of SQL:1999 began a philosophy of consistency in language design. Consistency means that wherever an object is permitted, an object expression should be permitted. In the FROM clause, this philosophy means that wherever a table is permitted, a table expression (a nested query) should be allowed. Nested queries in the FROM clause are not as widely used as nested queries in the WHERE and HAVING clauses. The remainder of this section demonstrates some specialized uses of nested queries in the FROM clause.

One usage of nested queries in the FROM clause is to compute an aggregate function within an aggregate function (nested aggregates). SQL does not permit an aggregate function inside another aggregate function. A nested query in the FROM clause overcomes the prohibition against nested aggregates as demonstrated in Example 9.21. Without a nested query in the FROM clause, two queries would be necessary to produce the output. In Access, the nested query would be a stored query. In Oracle, the nested query would be a view (see Chapter 10 for an explanation of views).

Another usage of a nested query in the FROM clause is to compute aggregates from multiple groupings. Without a nested query in the FROM clause, a query can contain aggregates from only one grouping. For example, multiple groupings are needed to summarize the number of students per offering and the number of resources per offering. This query would be useful if the design of the university database was extended with a *Resource* table and an associative table (*ResourceUsage*) connected to the *Offering* and the *Resource* tables via 1-M relationships. The query would require two nested queries in the FROM clause, one to retrieve the enrollment count for offerings and the other to retrieve the resource count for offerings.

Example 9.21

Using a Nested Query in the FROM Clause

List the course number, the course description, the number of offerings, and the average enrollment count across offerings.

```
SELECT T.CourseNo, T.CrsDesc, COUNT(*) AS NumOfferings,
       AVG(T.EnrollCount) AS AvgEnroll
 FROM
 ( SELECT Course.CourseNo, CrsDesc,
          Offering.OfferNo, COUNT(*) AS EnrollCount
   FROM Offering, Enrollment, Course
   WHERE Offering.OfferNo = Enrollment.OfferNo
     AND Course.CourseNo = Offering.CourseNo
   GROUP BY Course.CourseNo, CrsDesc, Offering.OfferNo
   ) T
 GROUP BY T.CourseNo, T.CrsDesc
```

CourseNo	CrsDesc	NumOfferings	AvgEnroll
FIN300	FUNDAMENTALS OF FINANCE	1	2
FIN450	PRINCIPLES OF INVESTMENTS	1	2
FIN480	CORPORATE FINANCE	1	3
IS320	FUNDAMENTALS OF BUSINESS PROGRAMMING	2	6
IS460	SYSTEMS ANALYSIS	1	7
IS480	FUNDAMENTALS OF DATABASE MANAGEMENT	2	5.5

In Access, a nested query in the FROM clause can compensate for the inability to use the DISTINCT keyword inside aggregate functions. For example, the DISTINCT keyword is necessary to compute the number of distinct courses taught by faculty as shown in Example 9.22. To produce the same results in Access, a nested query in the FROM clause is necessary as shown in Example 9.23. The nested query in the FROM clause uses the DISTINCT keyword to eliminate duplicate course numbers. Section 9.3.3 contains additional examples using nested queries in the FROM clause to compensate for the DISTINCT keyword inside the COUNT function.

Example 9.22 (Oracle)

Using the DISTINCT Keyword inside the COUNT Function

List the faculty number, the last name, and the number of unique courses taught.

```
SELECT Faculty.FacNo, FacLastName,
    COUNT(DISTINCT CourseNo) AS NumPreparations
 FROM Faculty, Offering
 WHERE Faculty.FacNo = Offering.FacNo
 GROUP BY Faculty.FacNo, FacLastName
```

FacNo	FacLastName	NumPreparations
098-76-5432	VINCE	1
543-21-0987	EMMANUEL	1
654-32-1098	FIBON	2
765-43-2109	MACON	2
876-54-3210	COLAN	1
987-65-4321	MILLS	2

Example 9.23

Using a Nested Query in the FROM Clause Instead of the DISTINCT Keyword inside the COUNT Function

List the faculty number, the last name, and the number of unique courses taught. The result is identical to Example 9.22. Although this SELECT statement executes in Access and Oracle, you should use the statement in Example 9.22 in Oracle because it will execute faster.

```
SELECT T.FacNo, T.FacLastName, COUNT(*) AS NumPreparations
FROM
 ( SELECT DISTINCT Faculty.FacNo, FacLastName, CourseNo
   FROM Offering, Faculty
   WHERE Offering.FacNo = Faculty.FacNo ) T
 GROUP BY T.FacNo, T.FacLastName
```

9.3 FORMULATING DIVISION PROBLEMS

Division problems can be some of the most difficult problems. Because of the difficulty, the divide operator of Chapter 3 is briefly reviewed. After this review, this section discusses some easier division problems before moving to more advanced problems.

TABLE 9-10
Student1 Table Listing

StdNo	SName	SCity
S1	JOE	SEATTLE
S2	SALLY	SEATTLE
S3	SUE	PORTLAND

9.3.1 Review of the Divide Operator

To review the divide operator, consider a simplified university database consisting of three tables: *Student1* (Table 9-10), *Club* (Table 9-11), and *StdClub* (Table 9-12) showing student membership in clubs. The divide operator is typically applied to associative or linking tables showing M-N relationships. The *StdClub* table links the *Student1* and *Club* tables in a M-N relationship as a student may belong to many clubs and a club may have many students.

TABLE 9-11
Club Table Listing

ClubNo	CName	CPurpose	CBudget	CActual
C1	DELTA	SOCIAL	$1,000.00	$1,200.00
C2	BITS	ACADEMIC	$500.00	$350.00
C3	HELPS	SERVICE	$300.00	$330.00
C4	SIGMA	SOCIAL		$150.00

TABLE 9-12
StdClub Table Listing

StdNo	ClubNo
S1	C1
S1	C2
S1	C3
S1	C4
S2	C1
S2	C4
S3	C3

The divide operator builds a table consisting of the values of one column (*StdNo*) that match *all* of the values in a specified column (*ClubNo*) of a second table (*Club*). A typical division problem is to list the students who belong to *all* clubs. The resulting table contains only student S1 because S1 is associated with all four clubs.

Divide: an operator of relational algebra that combines rows from two tables. The divide operator produces a table in which values of a column from one input table are associated with all the values from a column of the second table.

Division is more conceptually difficult than join because division matches on all values whereas join matches on a single value. If this problem involved a join, it would be stated as "list students who belong to *any* club." The key difference is the word *any* versus *all*. Most division problems can be written with adjectives *every* or *all* between a verb phrase representing a table and a noun representing another table. In this example, the phrase "students who belong to all clubs" fits this pattern. Another example is "students who have taken every course."

9.3.2 Simple Division Problems

There are a number of ways to perform division in SQL. Some textbooks describe an approach using Type II nested queries. Because this approach can be difficult to understand if you have not had a course in logic, a different approach is used here. The approach here uses the COUNT function with a nested query in the HAVING clause.

The basic idea is to compare the number of clubs associated with a student in the *StdClub* table with the number of clubs in the *Club* table. To perform this operation, group the *StdClub* table on *StdNo* and compare the count of rows in each *StdNo* group with the count of rows in the *Club* table. You can make this comparison using a nested query in the HAVING clause as shown in Example 9.24.

Example 9.24

Simplest Division Problem

List the student number of students who belong to all of the clubs.

```
SELECT StdNo
 FROM StdClub
 GROUP BY StdNo
 HAVING COUNT(*) = ( SELECT COUNT(*) FROM Club )
```

StdNo
S1

Note that the COUNT(*) on the left-hand side tallies the count or number of rows in a *StdNo* group. The right-hand side contains a nested query with only a COUNT(*) in the result. The nested query is Type I because there is no connection to the outer query. Therefore, the nested query only executes one time and returns a single row with a single value (the number of rows in the *Club* table).

Now let us examine some variations of the first problem. The most typical variation is to retrieve students who belong to a subset of the clubs rather than all of the clubs. For example, retrieve students who belong to all social clubs. To accomplish this change, you should modify Example 9.24 by including a WHERE condition in both the outer and the nested query. Instead of counting all *Student1* rows in a *StdNo* group, count only the rows where the club's purpose is social. Compare this count to the count of social clubs in the *Club* table. Example 9.25 shows these modifications.

Example 9.25

Division Problem to Find a Subset Match

List the student number of students who belong to all of the social clubs.

```
SELECT StdNo
 FROM StdClub, Club
 WHERE StdClub.ClubNo = Club.ClubNo
   AND CPurpose = 'SOCIAL'
 GROUP BY StdNo
 HAVING COUNT(*) =
 ( SELECT COUNT(*) FROM Club
   WHERE CPurpose = 'SOCIAL' )
```

StdNo
S1
S2

Examples 9.26 and 9.27 show other variations. In Example 9.26, a join between *StdClub* and *Student* is necessary to obtain the student name. Example 9.27 reverses the previous problems by looking for clubs rather than students.

Example 9.26

Division Problem with Joins

List the student number and the name of students who belong to all social clubs.

```
SELECT Student1.StdNo, SName
 FROM StdClub, Club, Student1
 WHERE StdClub.ClubNo = Club.ClubNo
   AND Student1.StdNo = StdClub.StdNo
   AND CPurpose = 'SOCIAL'
 GROUP BY Student1.StdNo, SName
 HAVING COUNT(*) =
 ( SELECT COUNT(*) FROM Club
   WHERE CPurpose = 'SOCIAL' )
```

StdNo	SName
S1	JOE
S2	SALLY

Example 9.27

Another Division Problem

List the club numbers of clubs that have all Seattle students as members.

```
SELECT ClubNo
 FROM StdClub, Student1
 WHERE Student1.StdNo = StdClub.StdNo
   AND SCity = 'SEATTLE'
 GROUP BY ClubNo
 HAVING COUNT(*) =
 ( SELECT COUNT(*) FROM Student1
   WHERE SCity = 'SEATTLE' )
```

ClubNo
C1
C4

9.3.3 Advanced Division Problems

Example 9.28 (using the original university database tables) depicts another complication of division problems in SQL. Before tackling this additional complication, let us examine a simpler problem. Example 9.28 can be formulated with the same technique as shown in Section 9.3.2. First, join the *Faculty* and *Offering* tables, select rows

Example 9.28 (Access)

Division Problem with a Join

List faculty number and the name (first and last) of faculty who teach all of the fall 2016, information systems offerings.

```
SELECT Faculty.FacNo, FacFirstName, FacLastName
 FROM Faculty, Offering
 WHERE Faculty.FacNo = Offering.FacNo
   AND OffTerm = 'FALL' AND CourseNo LIKE 'IS*'
   AND OffYear = 2016
 GROUP BY Faculty.FacNo, FacFirstName, FacLastName
 HAVING COUNT(*) =
 ( SELECT COUNT(*) FROM Offering
   WHERE OffTerm = 'FALL' AND OffYear = 2016
     AND CourseNo LIKE 'IS*' )
```

FacNo	FacFirstName	FacLastName
098-76-5432	LEONARD	VINCE

Example 9.28 (Oracle)

Division Problem with a Join

List faculty number and the name (first and last) of faculty who teach all of the fall 2016, information systems offerings.

```
SELECT Faculty.FacNo, FacFirstName, FacLastName
 FROM Faculty, Offering
 WHERE Faculty.FacNo = Offering.FacNo
   AND OffTerm = 'FALL' AND CourseNo LIKE 'IS%'
   AND OffYear = 2016
 GROUP BY Faculty.FacNo, FacFirstName, FacLastName
 HAVING COUNT(*) =
 ( SELECT COUNT(*) FROM Offering
   WHERE OffTerm = 'FALL' AND OffYear = 2016
     AND CourseNo LIKE 'IS%' )
```

matching the WHERE conditions, and group the result by faculty name (first and last). Then, compare the count of the rows in each faculty *name* group with the number of fall 2016, information systems offerings from the *Offering* table.

Example 9.28 is not particularly useful because it is unlikely that any instructor has taught every offering. Rather, it is more useful to retrieve instructors who have taught at least one offering of every course as demonstrated in Example 9.29. Rather than counting the rows in each group, count the unique *CourseNo* values. This change is necessary because *CourseNo* is not unique in the *Offering* table. There can be multiple rows with the same *CourseNo*, corresponding to multiple offerings for the same course. The solution only executes in Oracle because Access does not support the DISTINCT keyword in aggregate functions. Example 9.30 shows an Access solution using two nested queries in FROM clauses. The second nested query occurs inside the nested query in the HAVING clause. Appendix 9.A shows an alternative to nested queries in the FROM clause using multiple SELECT statements.

Example 9.29 (Oracle)

Division Problem with DISTINCT inside COUNT

List the faculty number and the name (first and last) of faculty who teach at least one section of all of the fall 2016 information systems courses.

```
SELECT Faculty.FacNo, FacFirstName, FacLastName
 FROM Faculty, Offering
 WHERE Faculty.FacNo = Offering.FacNo
   AND OffTerm = 'FALL' AND CourseNo LIKE 'IS%'
   AND OffYear = 2016
 GROUP BY Faculty.FacNo, FacFirstName, FacLastName
 HAVING COUNT(DISTINCT CourseNo) =
 ( SELECT COUNT(DISTINCT CourseNo) FROM Offering
   WHERE OffTerm = 'FALL' AND OffYear = 2016
     AND CourseNo LIKE 'IS%' )
```

FacNo	FacFirstName	FacLastName
098-76-5432	LEONARD	VINCE

Example 9.30 (Access)

Division Problem Using Nested Queries in the FROM Clauses instead of the DISTINCT Keyword inside the COUNT Function

List the faculty number and the name (first and last) of faculty who teach at least one section of all of the fall 2016 information systems courses. The result is the same as Example 9.29.

```
SELECT FacNo, FacFirstName, FacLastName
 FROM
 (SELECT DISTINCT Faculty.FacNo, FacFirstName,
         FacLastName, CourseNo
  FROM Faculty, Offering
  WHERE Faculty.FacNo = Offering.FacNo
    AND OffTerm = 'FALL' AND OffYear = 2016
    AND CourseNo LIKE 'IS*' )
 GROUP BY FacNo, FacFirstName, FacLastName
 HAVING COUNT(*) =
 ( SELECT COUNT(*) FROM
  ( SELECT DISTINCT CourseNo
    FROM Offering
    WHERE OffTerm = 'FALL' AND OffYear = 2016
      AND CourseNo LIKE 'IS*' ) )
```

Example 9.31 is another variation of the technique used in Example 9.29. The DISTINCT keyword is necessary so that students taking more than one offering from the same instructor are not counted twice. Note that the DISTINCT keyword is not necessary for the nested query because only rows of the *Student* table are counted. Example 9.32 shows an Access solution using a nested query in the FROM clause.

Example 9.31 (Oracle)

Another Division Problem with DISTINCT inside COUNT

List the faculty who have taught all seniors in their fall 2016 information systems offerings.

```
SELECT Faculty.FacNo, FacFirstName, FacLastName
 FROM Faculty, Offering, Enrollment, Student
 WHERE Faculty.FacNo = Offering.FacNo
   AND OffTerm = 'FALL' AND CourseNo LIKE 'IS%'
   AND OffYear = 2016 AND StdClass = 'SR'
   AND Offering.OfferNo = Enrollment.OfferNo
   AND Student.StdNo = Enrollment.StdNo
 GROUP BY Faculty.FacNo, FacFirstName, FacLastName
 HAVING COUNT(DISTINCT Student.StdNo) =
 ( SELECT COUNT(*) FROM Student
   WHERE StdClass = 'SR' );
```

FacNo	FacFirstName	FacLastName
098-76-5432	LEONARD	VINCE

Example 9.32 (Access)

Another Division Problem Using Nested Queries in the FROM Clauses Instead of the DISTINCT Keyword inside the COUNT Function

List the faculty who have taught all seniors in their fall 2016 information systems offerings. The result is identical to Example 9.31. The Oracle version of this statement uses the % as the wild card character.

```
SELECT FacNo, FacFirstName, FacLastName
 FROM
 ( SELECT DISTINCT Faculty.FacNo, FacFirstName,
      FacLastName, Student.StdNo
   FROM Faculty, Offering, Enrollment, Student
   WHERE Faculty.FacNo = Offering.FacNo
     AND OffTerm = 'FALL' AND CourseNo LIKE 'IS*'
     AND OffYear = 2016 AND StdClass = 'SR'
     AND Offering.OfferNo = Enrollment.OfferNo
     AND Student.StdNo = Enrollment.StdNo )
 GROUP BY FacNo, FacFirstName, FacLastName
 HAVING COUNT(*) =
 ( SELECT COUNT(*) FROM Student
   WHERE StdClass = 'SR' );
```

9.4 NULL VALUE CONSIDERATIONS

This section does not involve difficult matching problems or new parts of the SELECT statement. Rather, this section explains interpretation of query results when tables contain null values. These effects have largely been ignored until this section to simplify the presentation. Because many databases use null values, you need to understand the effects to attain a deeper understanding of query formulation.

Null values affect simple conditions involving comparison operators, compound conditions involving logical operators, aggregate calculations, and grouping. As you will see, some of the null value effects are rather subtle. Because of these subtle effects, a good table design minimizes, although it usually does not eliminate, the use of null values. The null value effects described in this section are specified in the SQL standards (1992 through 2016). Because specific DBMSs may provide different results, you may need to experiment with your DBMS.

9.4.1 Effect on Simple Conditions

Simple conditions involve a comparison operator, a column or column expression, and a constant, column, or column expression. A simple condition results in a null value if either column (or column expression) in a comparison is null. A row qualifies in the result if the simple condition evaluates to true for the row. Rows evaluating to false or null are discarded. Example 9.33 depicts a simple condition evaluating to null for one of the rows.

A more subtle result can occur when a simple condition involves two columns and at least one column contains null values. If neither column contains null values, every row will be in the result of either the simple condition or the opposite (negation) of the simple condition. For example, if < is the operator of a simple condition, the opposite

Example 9.33

Simple Condition Using a Column with Null Values

List the clubs (Table 9-11) with a budget greater than $200. The club with a null budget (C4) is omitted because the condition evaluates as a null value.

```
SELECT *
 FROM Club
 WHERE CBudget > 200
```

ClubNo	CName	CPurpose	CBudget	CActual
C1	DELTA	SOCIAL	$1,000.00	$1,200.00
C2	BITS	ACADEMIC	$500.00	$350.00
C3	HELPS	SERVICE	$300.00	$330.00

condition contains ≥ assuming the columns remain in the same positions. If at least one column contains null values, some rows will not appear in the result of either the simple condition or its negation. More precisely, rows containing null values will be excluded in both results as demonstrated in Examples 9.34 and 9.35.

Example 9.34

Simple Condition Involving Two Columns

List the clubs with the budget greater than the actual spending. The club with a null budget (C4) is omitted because the condition evaluates to null.

```
SELECT *
 FROM Club
 WHERE CBudget > CActual
```

ClubNo	CName	CPurpose	CBudget	CActual
C2	BITS	ACADEMIC	$500.00	$350.00

Example 9.35

Opposite Condition of Example 9.34

List the clubs with the budget less than or equal to the actual spending. The club with a null budget (C4) is omitted because the condition evaluates to null.

```
SELECT *
 FROM Club
 WHERE CBudget <= CActual
```

ClubNo	CName	CPurpose	CBudget	CActual
C1	DELTA	SOCIAL	$1,000.00	$1,200.00
C3	HELPS	SERVICE	$300.00	$330.00

9.4.2 Effect on Compound Conditions

Compound conditions involve one or more simple conditions connected by the logical or Boolean operators AND, OR, and NOT. Like simple conditions, compound conditions evaluate to true, false, or null. A row is selected if the entire compound condition in the WHERE clause evaluates to true.

To evaluate the result of a compound condition, the SQL:2016 standard uses truth tables with three values. A truth table shows the results of combinations of values (true, false, and null) with the logical operators. Truth tables with three values define a three-valued logic. Tables 9-13 through 9-15 depict truth tables for the AND, OR, and NOT operators. The internal cells in these tables are the result values. For example, the first internal cell (True) in Table 9-13 results from the AND operator applied to two conditions with true values. You can test your understanding of the truth tables using Examples 9.36 and 9.37.

TABLE 9-13
AND Truth Table

AND	True	False	Null
True	True	False	Null
False	False	False	False
Null	Null	False	Null

TABLE 9-14
OR Truth Table

OR	True	False	Null
True	True	True	True
False	True	False	Null
Null	True	Null	Null

TABLE 9-15
NOT Truth Table

NOT	True	False	Null
	False	True	Null

9.4.3 Effect on Aggregate Calculations and Grouping

Null values are ignored in aggregate calculations. Although this statement seems simple, the results can be subtle. For the COUNT function, COUNT(*) returns a different value than COUNT(column) if the column contains null values. COUNT(*) always returns the number of rows. COUNT(column) returns the number of non-null values in the column. Example 9.38 demonstrates the difference between COUNT(*) and COUNT(column).

Example 9.36

Evaluation of a Compound OR Condition with a Null Value

List the clubs with a budget less than or equal to the actual spending or the actual spending less than $200. The club with a null budget (C4) is included because the second condition evaluates to true.

```
SELECT *
 FROM Club
 WHERE CBudget <= CActual OR CActual < 200
```

ClubNo	CName	CPurpose	CBudget	CActual
C1	DELTA	SOCIAL	$1,000.00	$1,200.00
C3	HELPS	SERVICE	$300.00	$330.00
C4	SIGMA	SOCIAL		$150.00

Example 9.37

Evaluation of a Compound AND Condition with a Null Value

List the clubs (Table 9-11) with the budget less than or equal to the actual spending and the actual spending less than $500. The club with a null budget (C4) is not included because the first condition evaluates to null.

```
SELECT *
 FROM Club
 WHERE CBudget <= CActual AND CActual < 500
```

ClubNo	CName	CPurpose	CBudget	CActual
C3	HELPS	SERVICE	$300.00	$330.00

Example 9.38

COUNT Function with Null Values

List the number of rows in the *Club* table and the number of non null values in the *CBudget* column.

```
SELECT COUNT(*) AS NumRows,
       COUNT(CBudget) AS NumBudgets
 FROM Club
```

NumRows	NumBudgets
4	3

An even more subtle effect can occur if the SUM or AVG functions are applied to a column with null values. Without regard to null values, the following equation is true: SUM(Column1) + SUM(Column2) = SUM(Column1 + Column2). With null values in at least one of the columns, the equation may not be true because a calculation involving a null value yields a null value. If Column1 has a null value in one row, the plus operation in SUM(Column1 + Column2) produces a null value for that row. However, the value of Column2 in the same row is counted in SUM(Column2). Example 9.39 demonstrates this subtle effect using the minus operator instead of the plus operator.

Example 9.39

SUM Function with Null Values

Using the *Club* table, list the sum of the budget values, the sum of the actual values, the difference of the two sums, and the sum of the differences (budget – actual). Parentheses enclose negative values in the result. The last two columns differ because of a null value in the *CBudget* column for *ClubNo* C4. The *CActual* value (150) in the C4 row counts in SUM(*CActual*). However, SUM(*CBudget* – *CActual*) uses null for the difference in the C4 row.

```
SELECT SUM(CBudget) AS SumBudget,
       SUM(CActual) AS SumActual,
       SUM(CBudget)-SUM(CActual) AS SumDifference,
       SUM(CBudget-CActual) AS SumOfDifferences
 FROM Club
```

SumBudget	SumActual	SumDifference	SumOfDifferences
$1,800.00	$2,030.00	($230.00)	($80.00)

Null values also can affect grouping operations performed in the GROUP BY clause. The SQL standard stipulates that all rows with null values are grouped together. The grouping column shows null values in the result. In the university database, this kind of grouping is useful to find offerings without assigned professors, as demonstrated in Example 9.40.

Example 9.40

Grouping on a Column with Null Values

For each faculty number in the *Offering* table, list the number of offerings. In Microsoft Access and Oracle, an *Offering* row with a null *FacNo* value displays as a blank. In Access, the null row displays before the non-null rows as shown below. In Oracle, the null row displays after non-null rows.

```
SELECT FacNo, COUNT(*) AS NumRows
 FROM Offering
 GROUP BY FacNo
```

FacNo	NumRows
	2
098-76-5432	3
543-21-0987	1
654-32-1098	2
765-43-2109	2
876-54-3210	1
987-65-4321	2

9.5 HIERARCHICAL QUERIES

Hierarchical queries involve self-referencing relationships in which a child entity is related to at most one parent entity. Classical organization charts, part explosion diagrams, chart of accounts, and XML documents are the most prominent examples amendable to hierarchical queries. A typical hierarchical query may involve finding details or summarizing features about employees managed directly or indirectly by a specified manager. Self-referencing relationships are specialized but important parts of applications so understanding hierarchical query formulation provides advanced skills not shared by typical database professionals.

This section covers two approaches for hierarchical queries. Oracle developed a proprietary extension of the SELECT statement using the CONNECT BY PRIOR clause. Standard SQL (SQL:1999 onwards) provides an extension to the SELECT statement involving the WITH clause and recursive common table expressions. Since the Oracle notation is more succinct, the proprietary Oracle approach is covered first in more detail. The standard SQL notation is presented in less detail to provide some background for other DBMSs. Hierarchical queries are not supported in Microsoft Access so none of the examples in this section execute in any version of Microsoft Access. Before covering the hierarchical query approaches, an example of hierarchically structured data is presented.

Hierarchical Query: a query involving self-referencing relationships in which a child row is related to at most one parent row. Hierarchical queries typically retrieve details about child rows (both direct and indirect) or summarize column values of child rows.

9.5.1 Hierarchical Data Example

To study hierarchical query formulation, you need to clearly understand hierarchically structured data. Table 9-16 shows the new *Faculty2* table expanded from the *Faculty* table with additional rows, some columns removed, and some rows slightly altered to fit into hierarchical query examples. The *Faculty2* table has a self-referencing relationship with *FacSupervisor* as a foreign key referencing *FacNo*. Each row has at most one parent row. Rows having a null value for *FacSupervisor* reside at the top of the organization chart. In Table 9-16, Victoria Emmanuel and Nicki Macon reside at the top of the organization chart.

To clarify the hierarchical structure, Figures 9.2 and 9.3 graphically depict organization charts along with important column values for easy comparison. A graphical representation of sample data can help you to formulate hierarchical queries.

TABLE 9-16
Sample *Faculty2* Table

FacNo	FacSupervisor	FacFirstName	FacLastName	FacHireDate	FacRank	FacSalary
098-76-5432	654-32-1098	LEONARD	VINCE	10-Apr-2004	ASST	$55,000
543-21-0987		VICTORIA	EMMANUEL	15-Apr-2005	PROF	$120,000
654-32-1098	543-21-0987	LEONARD	FIBON	01-May-2003	ASSC	$70,000
765-43-2109		NICKI	MACON	11-Apr-2006	ASSC	$105,000
876-54-3210	654-32-1098	CRISTOPHER	COLAN	01-Mar-2008	ASST	$90,000
987-65-4321	765-43-2109	JULIA	MILLS	15-Mar-2009	ASSC	$95,000
111-22-3333	543-21-0987	JOHN	MILLSON	01-May-2009	PROF	$110,000
333-22-4444	111-22-3333	SALLY	SCOTT	01-May-2010	ASST	$90,000
555-66-7777	111-22-3333	SUSAN	JONES	01-May-2011	ASSC	$125,000
777-11-4321	765-43-2109	AIMEE	MANNING	15-Mar-2010	ASST	$85,000
888-33-1111	987-65-4321	JAMES	BLOKE	15-Apr-2012	ASST	$85,000
789-12-3210	987-65-4321	JAIME	SANCHEZ	10-May-2013	PROF	$107,000

FIGURE 9.2
Organization Chart with Victoria Emmanuel at the Top

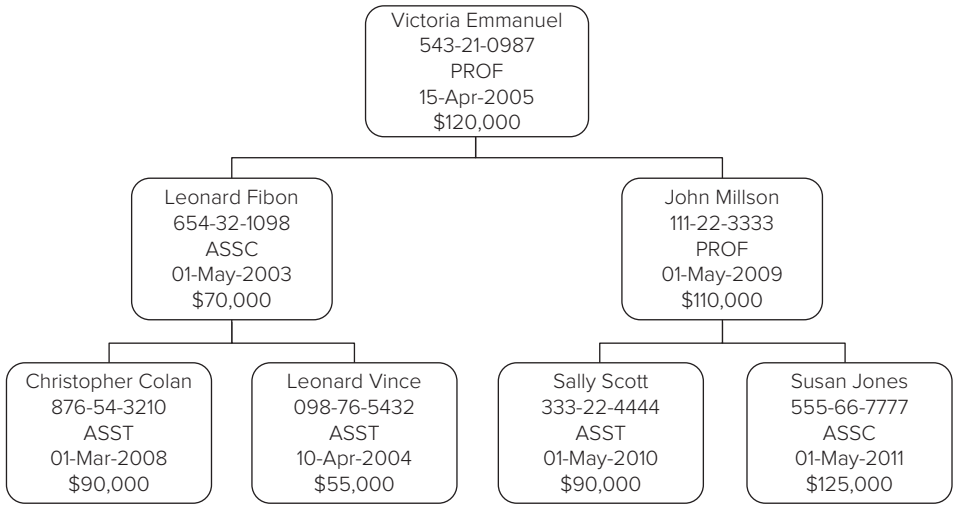
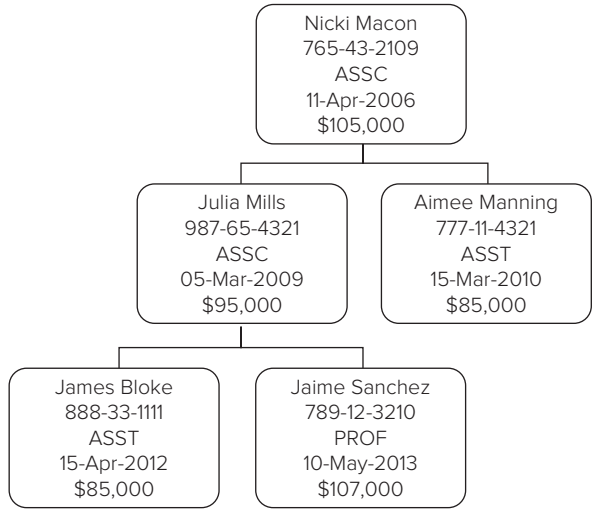


FIGURE 9.3
Organization Chart with Nicki Macon at the Top



The basic hierarchical query involves listing each top level faculty member along with all related subordinates, direct and indirect. The sequence of rows from root (top row) to leaf (lowest level) is known as a path. A SELECT statement cannot list all subordinates on a path unless the number of levels of subordinates is known. A self-join can be done for each level but a variable number of self-joins is necessary even if the maximum number of levels is known. Thus, an extension to the SELECT statement is necessary to formulate even the basic hierarchical query.

Compiler optimization and higher productivity are important advantages of query language support for hierarchical queries. SQL compilers have specialized algorithms and optimization methods for hierarchical queries. In contrast, coding a hierarchical query in a procedure with explicit loops eliminates the possibility of optimization by a SQL compiler. In addition, procedural coding reduces software productivity as more lines of code are necessary along with programming language knowledge.

9.5.2 Proprietary Oracle Extensions for Hierarchical Queries

Oracle provides the CONNECT BY PRIOR clause along with other clauses, operators, functions, and pseudo columns to support hierarchical queries. Syntactically, the CONNECT BY PRIOR clause and other clauses follow the FROM and WHERE clauses in a SELECT statement. The operators, functions, and pseudo columns can appear in expressions in the list of result columns and conditions. Pseudo columns are not actual columns in a table, but they behave like columns.

The examples begin with the simplest hierarchical query, although not particularly useful. The CONNECT BY PRIOR clause contains a condition relating parent and child rows, typically the self-join condition. Example 9.41 uses the CONNECT BY PRIOR clause to visit each row on a path. Each row is visited one time for each level on a path. For example, the row with COLAN appears three times in the result because it resides on level 3 as previously shown in Figure 9.2. The LEVEL pseudo column identifies the hierarchical level of a row starting with 1 for the root level. The *Faculty2* table contains two root rows appearing with level 1 in the result.

Example 9.41

Simple Hierarchical Query using CONNECT BY PRIOR (Oracle)

The result contains 28 rows (Table 9-17) with 18 rows for the six leaf rows (faculty not managing other faculty) at level 3 (6 * 3), 8 rows for the 4 rows at level 2 (4 * 2), and 2 root rows. The sorting order makes it easier to verify the visitation of rows.

```
SELECT FacNo, FacSupervisor, FacFirstName, FacLastName,
       FacHireDate, FacSalary, FacRank, LEVEL
 FROM Faculty2
 CONNECT BY PRIOR FacNo = FacSupervisor
 ORDER BY FacNo, LEVEL;
```

FacNo	FacSupervisor	FacFirstName	FacLastName	FacHireDate	FacRank	FacSalary	LEVEL
098-76-5432	654-32-1098	LEONARD	VINCE	10-APR-04	ASST	55000	1
098-76-5432	654-32-1098	LEONARD	VINCE	10-APR-04	ASST	55000	2
098-76-5432	654-32-1098	LEONARD	VINCE	10-APR-04	ASST	55000	3
111-22-3333	543-21-0987	JOHN	MILLSON	01-MAY-09	PROF	110000	1
111-22-3333	543-21-0987	JOHN	MILLSON	01-MAY-09	PROF	110000	2
333-22-4444	111-22-3333	SALLY	SCOTT	01-MAY-09	ASST	90000	1
333-22-4444	111-22-3333	SALLY	SCOTT	01-MAY-10	ASST	90000	2

FacNo	FacSupervisor	FacFirstName	FacLastName	FacHireDate	FacRank	FacSalary	LEVEL
333-22-4444	111-22-3333	SALLY	SCOTT	01-MAY-10	ASST	90000	3
543-21-0987		VICTORIA	EMMANUEL	15-APR-05	PROF	120000	1
555-66-7777	111-22-3333	SUSAN	JONES	01-MAY-11	ASSC	125000	1
555-66-7777	111-22-3333	SUSAN	JONES	01-MAY-11	ASSC	125000	2
555-66-7777	111-22-3333	SUSAN	JONES	01-MAY-11	ASSC	125000	3
654-32-1098	543-21-0987	LEONARD	FIBON	01-MAY-03	ASSC	70000	1
654-32-1098	543-21-0987	LEONARD	FIBON	01-MAY-03	ASSC	70000	2
765-43-2109		NICKI	MACON	11-APR-06	ASSC	105000	1
777-11-4321	765-43-2109	AIMEE	MANNING	15-MAR-10	ASST	85000	1
777-11-4321	765-43-2109	AIMEE	MANNING	15-MAR-10	ASST	85000	2
789-12-3210	987-65-4321	JAIME	SANCHEZ	10-MAY-13	PROF	107000	1
789-12-3210	987-65-4321	JAIME	SANCHEZ	10-MAY-13	PROF	107000	2
789-12-3210	987-65-4321	JAIME	SANCHEZ	10-MAY-13	PROF	107000	3
876-54-3210	654-32-1098	CRISTOPHER	COLAN	01-MAR-08	ASST	90000	1
876-54-3210	654-32-1098	CRISTOPHER	COLAN	01-MAR-08	ASST	90000	2
876-54-3210	654-32-1098	CRISTOPHER	COLAN	01-MAR-08	ASST	90000	3
888-33-1111	987-65-4321	JAMES	BLOKE	15-APR-12	ASST	85000	1
888-33-1111	987-65-4321	JAMES	BLOKE	15-APR-12	ASST	85000	2
888-33-1111	987-65-4321	JAMES	BLOKE	15-APR-12	ASST	85000	3
987-65-4321	765-43-2109	JULIA	MILLS	15-MAR-09	ASSC	95000	1
987-65-4321	765-43-2109	JULIA	MILLS	15-MAR-09	ASSC	95000	2

Example 9.41 treats all rows as roots of hierarchies. Typically, a small number of rows are designated as roots. Oracle provides the START WITH clause to identify root rows. In Example 9.42, the rows with null values for *FacSupervisor* are designated as the starting rows. The START WITH clause eliminates duplicate rows in the result caused by treating each row as a root.

Example 9.42

Hierarchical Query using START WITH (Oracle)

Example 9.42 revises Example 9.41 with a START WITH clause to limit starting rows to the roots of the faculty hierarchies. The results are conveniently sorted to indicate that each *Faculty2* row appears once in the result.

```
SELECT FacNo, FacSupervisor, FacFirstName, FacLastName,
       FacHireDate, FacSalary, FacRank, LEVEL
  FROM Faculty2
  START WITH FacSupervisor IS NULL
  CONNECT BY PRIOR FacNo = FacSupervisor
  ORDER BY LEVEL;
```

FacNo	FacSupervisor	FacFirstName	FacLastName	FacHireDate	FacRank	FacSalary	LEVEL
765-43-2109		NICKI	MACON	11-APR-06	ASSC	105000	1
543-21-0987		VICTORIA	EMMANUEL	15-APR-05	PROF	120000	1
654-32-1098	543-21-0987	LEONARD	FIBON	01-MAY-03	ASSC	70000	2

FacNo	FacSupervisor	FacFirstName	FacLastName	FacHireDate	FacRank	FacSalary	LEVEL
987-65-4321	765-43-2109	JULIA	MILLS	15-MAR-09	ASSC	95000	2
777-11-4321	765-43-2109	AIMEE	MANNING	15-MAR-10	ASST	85000	2
111-22-3333	543-21-0987	JOHN	MILLSON	01-MAY-09	PROF	110000	2
789-12-3210	987-65-4321	JAIME	SANCHEZ	10-MAY-13	PROF	107000	3
876-54-3210	654-32-1098	CRISTOPHER	COLAN	01-MAR-08	ASST	90000	3
888-33-1111	987-65-4321	JAMES	BLOKE	15-APR-12	ASST	85000	3
555-66-7777	111-22-3333	SUSAN	JONES	01-MAY-11	ASSC	125000	3
333-22-4444	111-22-3333	SALLY	SCOTT	01-MAY-10	ASST	90000	3
098-76-5432	654-32-1098	LEONARD	VINCE	10-APR-04	ASST	55000	3

In Example 9.42, the relationship of rows on the same level is not clear. To depict relationships among rows on the same level, Oracle provides the SIBLINGS keyword to specify a sort order for siblings, rows with the same parent. Example 9.43 sorts siblings by *FacLastName*, a more convenient order than *FacNo*. In the result, you can see the lexicographic order for siblings with COLAN followed by VINCE under parent row FIBON. The rows are indented in the first column to show the hierarchical relationships.

Example 9.43

Hierarchical Query using the SIBLNGS Keyword (Oracle)

The LPAD function adds spaces on the left to show the hierarchical structure. The LEVEL pseudo column determines the amount of padding with no padding for the root rows.

```
SELECT LPAD(' ',2*(LEVEL-1)) || FacLastName AS LastName,
       FacHireDate, FacSalary, FacRank, LEVEL
  FROM Faculty2
  START WITH FacSupervisor IS NULL
  CONNECT BY PRIOR FacNo = FacSupervisor
  ORDER SIBLINGS BY FacLastName;
```

LastName	FacHireDate	FacRank	FacSalary	LEVEL
EMMANUEL	15-APR-05	PROF	120000	1
FIBON	01-MAY-03	ASSC	70000	2
COLAN	01-MAR-08	ASST	90000	3
VINCE	10-APR-04	ASST	55000	3
MILLSON	01-MAY-09	PROF	110000	2
JONES	01-MAY-11	ASSC	125000	3
SCOTT	01-MAY-10	ASST	90000	3
MACON	11-APR-06	ASSC	105000	1
MANNING	15-MAR-10	ASST	85000	2
MILLS	15-MAR-09	ASSC	95000	2
BLOKE	15-APR-12	ASST	85000	3
SANCHEZ	10-MAY-13	PROF	107000	3

Instead of indenting to show the hierarchical structure, the complete path can be shown. Oracle provides the SYS_CONNECT_BY_PATH function to show the complete path with a column name and a separator character as parameters.

Example 9.44

Hierarchical Query using the SYS_CONNECT_BY_PATH function (Oracle)

The SYS_CONNECT_BY_PATH function uses the *FacLastName* column to identify rows and the / as the separator between rows on a path.

```
SELECT SYS_CONNECT_BY_PATH(FacLastName,'/') AS Path,
       FacHireDate, FacSalary, FacRank, LEVEL
 FROM Faculty2
 START WITH FacSupervisor IS NULL
 CONNECT BY PRIOR FacNo = FacSupervisor
 ORDER SIBLINGS BY FacLastName;
```

Path	FacHireDate	FacRank	FacSalary	LEVEL
/EMMANUEL	15-APR-05	PROF	120000	1
/EMMANUEL/FIBON	01-MAY-03	ASSC	70000	2
/EMMANUEL/FIBON/COLAN	01-MAR-08	ASST	90000	3
/EMMANUEL/FIBON/VINCE	10-APR-04	ASST	55000	3
/EMMANUEL/MILLSON	01-MAY-09	PROF	110000	2
/EMMANUEL/MILLSON/JONES	01-MAY-11	ASSC	125000	3
/EMMANUEL/MILLSON/SCOTT	01-MAY-10	ASST	90000	3
/MACON	11-APR-06	ASSC	105000	1
/MACON/MANNING	15-MAR-10	ASST	85000	2
/MACON/MILLS	15-MAR-09	ASSC	95000	2
/MACON/MILLS/BLOKE	15-APR-12	ASST	85000	3
/MACON/MILLS/SANCHEZ	10-MAY-13	PROF	107000	3

In addition to the SYS_CONNECT_BY_PATH function, Oracle provides other syntax elements for hierarchical queries as shown in Example 9.45. The CONNECT_BY_ROOT operator retrieves a column value from a root row. The CONNECT_BY_LEAF pseudo column provides a row's leaf status. A row is a leaf if it has no children.

Example 9.45

Hierarchical Query using CONNECT_BY_ROOT and CONNECT_BY_ISLEAF (Oracle)

The CONNECT_BY_ROOT operator uses a column name (*FacLastName*). The CONNECT_BY_ISLEAF pseudo column returns 1 if the row is a leaf and 0 otherwise.

```
SELECT SYS_CONNECT_BY_PATH(FacLastName,'/') AS Path,
       CONNECT_BY_ROOT FacLastName AS Root,
       CONNECT_BY_ISLEAF AS IsLeaf,
       FacHireDate, FacSalary, FacRank, LEVEL
 FROM Faculty2
 START WITH FacSupervisor IS NULL
 CONNECT BY PRIOR FacNo = FacSupervisor
 ORDER SIBLINGS BY FacLastName;
```

Path	Root	IsLeaf	FacHireDate	FacRank	FacSalary	LEVEL
/EMMANUEL	EMMANUEL	0	15-APR-05	PROF	120000	1
/EMMANUEL/FIBON	EMMANUEL	0	01- MAY-03	ASSC	70000	2
/EMMANUEL/FIBON/COLAN	EMMANUEL	1	01-MAR-08	ASST	90000	3
/EMMANUEL/FIBON/VINCE	EMMANUEL	1	10-APR-04	ASST	55000	3
/EMMANUEL/MILLSON	EMMANUEL	0	01-MAY-09	PROF	110000	2
/EMMANUEL/MILLSON/JONES	EMMANUEL	1	01-MAY-11	ASSC	125000	3
/EMMANUEL/MILLSON/SCOTT	EMMANUEL	1	01-MAY-10	ASST	90000	3
/MACON	MACON	0	11-APR-06	ASSC	105000	1
/MACON/MANNING	MACON	1	15-MAR-10	ASST	85000	2
/MACON/MILLS	MACON	0	15-MAR-09	ASSC	95000	2
/MACON/MILLS/BLOKE	MACON	1	15-APR-12	ASST	85000	3
/MACON/MILLS/SANCHEZ	MACON	1	10-MAY-13	PROF	107000	3

The CONNECT_BY_ROOT operator can be used indirectly for grouping so that summary totals can be calculated for paths in a hierarchy. To calculate summary totals, the root should not be restricted by the START WITH clause. Without a root restriction, every row will be considered a root so that summary totals are calculated for each row, not just the root rows. Example 9.46 shows summary salary totals and subordinate counts for each faculty member and subordinates.

Example 9.46

Summary Totals for a Hierarchical Query using the CONNECT_BY_ROOT function (Oracle)

A nested query in the FROM clause is necessary because the CONNECT_BY_ROOT operator cannot be used as a grouping column. The result is sorted by the number of faculty members in the group including the root row and subordinates (COUNT(*)).

```
SELECT Root, COUNT(*)-1 AS NumSubordinates,
       SUM(FacSalary) AS FacSalarySum
 FROM
  ( SELECT CONNECT_BY_ROOT FacLastName AS Root, FacSalary
    FROM Faculty2
    CONNECT BY PRIOR FacNo = FacSupervisor )
GROUP BY Root
ORDER BY COUNT(*) DESC;
```

Root	NumSubordinates	FacSalarySum
EMMANUEL	6	660000
MACON	4	477000
FIBON	2	215000
MILLS	2	287000
MILLSON	2	325000
SANCHEZ	0	107000

Root	NumSubordinates	FacSalarySum
BLOKE	0	85000
SCOTT	0	90000
MANNING	0	85000
VINCE	0	55000
JONES	0	125000
COLAN	0	90000

Path Exception Query
a hierarchical query listing violations of monotonicity in path relationships in hierarchical data. Monotonicity means that column values of subordinates change in the same direction (usually smaller) from ancestors.

The last problems involve path exceptions. Many hierarchies show consistency of values known as monotonicity on paths. Monotonicity means that column values of subordinates change in the same direction (usually smaller) from ancestors. For example in an organization hierarchy, monotonicity indicates that a manager has larger compensation than direct and indirect subordinates. **Path exception queries** involve violations about expected monotonicity of values. Typical path exception queries involve managers making less than subordinates, assemblies weighing less than constituent components, and parent accounts with smaller balances than related subaccounts.

The first step to formulate path exception queries involves the closure of the hierarchy. The closure of a hierarchy shows all pairs in which a child can be reached from a parent. In an organization chart, the closure shows a manager paired with each direct and indirect subordinate. In Figure 9.2, the closure for Macon contains Macon paired with each subordinate, Mills, Manning, Bloke, and Sanchez. Example 9.47 demonstrates the SELECT statement to derive the closure for the *Faculty2* table. Each row in the hierarchy is paired with each direct and indirect subordinate as shown in query result.

The last two examples retrieve the closure in the nested query and apply WHERE conditions in the outer query to retrieve path exceptions. Example 9.48 retrieves

Example 9.47

Closure of the Faculty2 Table (Oracle)

The WHERE clause removes redundant rows in which the faculty number matches the root's faculty number or the supervisor is null. The nested query in the FROM clause is necessary because the WHERE condition must be applied to the result without a starting row as shown in Example 9.41. The unary PRIOR operator retrieves the value of the column name argument from the parent row. Note that the PRIOR operator involves the immediate parent row while the CONNECT_BY_ROOT operator involves the root row. The SYS_CONNECT_BY_PATH function and sort order improve the readability of the result.

```
SELECT *
 FROM
  ( SELECT FacNo, PRIOR FacNo AS PriorFacNo,
      CONNECT_BY_ROOT FacNo AS FacSupNo, FacLastName,
      FacSalary, FacRank,
      SYS_CONNECT_BY_PATH(FacLastName,'/') AS Path
    FROM Faculty2
    CONNECT BY PRIOR FacNo = FacSupervisor )
WHERE FacSupNo <> FacNo
ORDER BY FacLastName;
```

FacNo	PriorFacNo	FacSupNo	FacLastName	FacRank	FacSalary	Path
888-33-1111	987-65-4321	987-65-4321	BLOKE	ASST	85000	/MILLS/BLOKE
888-33-1111	987-65-4321	765-43-2109	BLOKE	ASST	85000	/MACON/MILLS/BLOKE
876-54-3210	654-32-1098	543-21-0987	COLAN	ASST	90000	/EMMANUEL/FIBON/COLAN
876-54-3210	654-32-1098	654-32-1098	COLAN	ASST	90000	/FIBON/COLAN
654-32-1098	543-21-0987	543-21-0987	FIBON	ASSC	70000	/EMMANUEL/FIBON
555-66-7777	111-22-3333	111-22-3333	JONES	ASSC	125000	/MILLSON/JONES
555-66-7777	111-22-3333	543-21-0987	JONES	ASSC	125000	/EMMANUEL//MILLSON/JONES
777-11-4321	765-43-2109	765-43-2109	MANNING	ASST	85000	/MACON/MANNING
987-65-4321	765-43-2109	765-43-2109	MILLS	ASSC	95000	/MACON/MILLS
111-22-3333	543-21-0987	543-21-0987	MILLSON	PROF	110000	/EMMANUEL/MILLSON
789-12-3210	987-65-4321	987-65-4321	SANCHEZ	PROF	107000	/MILLS/SANCHEZ
789-12-3210	987-65-4321	765-43-2109	SANCHEZ	PROF	107000	/MACON/MILLS/SANCHEZ
333-22-4444	111-22-3333	111-22-3333	SCOTT	ASST	90000	/MILLSON/SCOTT
333-22-4444	111-22-3333	543-21-0987	SCOTT	ASST	90000	/EMMANUEL/MILLSON/SCOTT
098-76-5432	654-32-1098	543-21-0987	VINCE	ASST	55000	/EMMANUEL/FIBON/VINCE
098-76-5432	654-32-1098	654-32-1098	VINCE	ASST	55000	/FIBON/VINCE

subordinate faculty members with larger salaries than their ancestors (either direct or indirect supervisors). Example 9.49 retrieves subordinate faculty members with a higher rank than their supervisors. The rank order is PROF (full professor), ASSC (associate professor), and ASST (assistant professor). The results for both examples show the values for the subordinate and ancestor faculty member for ease of comparison.

The Oracle notation presented in this section has a number of syntax elements. To help you use the notation, Table 9-16 presents a convenient summary.

Example 9.48

Path Exception Query to Retrieve Faculty Earning more than a Supervisor, either Direct or Indirect (Oracle)

The path exception condition (comparison of faculty salary values) is added to the WHERE condition of the outer query. The nested query uses the CONNECT_BY_ROOT operator to retrieve values from the root row.

```
SELECT *
 FROM
 ( SELECT FacNo, CONNECT_BY_ROOT FacNo AS FacSupNo, FacLastName,
     CONNECT_BY_ROOT FacLastName AS FacSupLastName, FacSalary,
     CONNECT_BY_ROOT FacSalary AS FacSupSalary
  FROM Faculty2
  CONNECT BY PRIOR FacNo = FacSupervisor )
 WHERE FacSupNo <> FacNo AND FacSalary > FacSupSalary;
```

FacNo	FacSupNo	FacLastName	FacSupLastName	FacSalary	FacSupSalary
876-54-3210	654-32-1098	COLAN	FIBON	90000	70000
555-66-7777	543-21-0987	JONES	MILLSON	125000	110000
789-12-3210	987-65-4321	SANCHEZ	MILLS	107000	95000
555-66-7777	111-22-3333	JONES	EMMANUEL	125000	120000
789-12-3210	765-43-2109	SANCHEZ	MACON	107000	105000

Example 9.49

Path Exception Query to Retrieve Faculty with a Higher Rank than a Supervisor, either Direct or Indirect (Oracle)

The path exception condition (comparison of faculty rank values) in the outer query involves three combinations of inconsistent values for a subordinate (*FacRank*) and an ancestor (*FacSupRank*). The order among ranks is PROF > ASSC > ASST.

```
SELECT *
 FROM
 ( SELECT FacNo, CONNECT_BY_ROOT FacNo AS FacSupNo, FacLastName,
     CONNECT_BY_ROOT FacLastName AS FacSupLastName,
     FacRank, CONNECT_BY_ROOT FacRank AS FacSupRank
  FROM Faculty2
  CONNECT BY PRIOR FacNo = FacSupervisor )
WHERE FacSupNo <> FacNo
  AND ( (FacRank = 'PROF' AND FacSupRank = 'ASSC' )
   OR ( FacRank = 'PROF' AND FacSupRank = 'ASST' )
   OR ( FacRank = 'ASSC' AND FacSupRank = 'ASST' ) );
```

FacNo	FacSupNo	FacLastName	FacSupLastName	FacRank	FacSupRank
789-12-3210	987-65-4321	SANCHEZ	MILLS	PROF	ASSC
789-12-3210	765-43-2109	SANCHEZ	MACON	PROF	ASSC

TABLE 9-16
Summary of Proprietary Oracle Notation for Hierarchical Queries

Syntax Element	Meaning
CONNECT BY PRIOR	Clause to specify a condition that establishes a link between parent and child rows
START WITH	Clause to specify a condition to identify the root rows in a hierarchical query
SIBLINGS	Keyword to indicate a sort order for siblings, rows with the same parent. Used in the ORDER BY clause
LEVEL	Pseudo column to determine the hierarchical level of a row beginning with 1 for root rows
CONNECT_BY_ISLEAF	Pseudo column to determine leaf status of a row, 1 if a row has no related child rows, 0 otherwise
SYS_CONNECT_BY_PATH	Function to retrieve the path for a row using a column and separator character
CONNECT_BY_ROOT	Unary operator in the SELECT clause to retrieve the value of a specified column from a root row
PRIOR	Unary operator in the SELECT clause to reference the value of a specified column in the parent row

9.5.3 Extensions in the SQL Standard for Hierarchical Queries

The SQL standard, starting with SQL:1999, provided **recursive common table expressions (CTE)** to formulate hierarchical queries. CTEs can be used for other purposes besides hierarchical queries although they are only necessary for hierarchical queries so they were not previously introduced. Recursion means self-reference so a recursive CTE references itself. Recursive CTEs are supported by most major enterprise DBMSs including Oracle so recursive CTEs provide a reasonably portable notation compared to the proprietary nature of the Oracle CONNECT BY PRIOR clause.

Recursive Common Table Expression (CTE)
Recursive CTEs are the SQL standard notation for hierarchical queries. A recursive CTE involves two query blocks connected by a union operation and a second SELECT statement. The second query block references the CTE, a self-reference. The second SELECT statement uses the CTE to generate the results.

The recursive CTE notation is more verbose than the Oracle notation although the recursive CTE notation uses only one syntax element compared to many syntax elements in the Oracle notation. The CTE notation involves two query blocks connected by a union operation followed by a second SELECT statement. Example 9.50 shows the basic pattern for a hierarchical query using a recursive CTE. Note the WITH keyword begins the CTE.

Example 9.50

Query pattern for Hierarchical Query using a Recursive CTE

The WITH keyword indicates the CTE name and column names. The first SELECT block (<CTEQuery1>), known as the anchor member, references the table with hierarchical data. The second SELECT block (<CTEQuery2>), known as the recursive member, references the CTE name. The SELECT statement after the WITH clause uses the CTE to generate the result. The semicolon terminates the entire statement including the WITH clause and SELECT statement.

```
WITH CTEName ( ColumnName* )
AS
-- Anchor member (AM) referencing the hierarchical table.
 ( <CTEQuery1>
UNION ALL
-- Recursive member (RM) referencing the CTEName.
  <CTEQuery2> )
-- Statement using CTEName
SELECT * FROM CTEName;
```

Example 9.51 shows a SQL statement conforming to the pattern in Example 9.50. Example 9.51 begins with the definition of the CTE following the WITH keyword. *Faculty2CTE* contains the columns to identify a row as well as a column to identify the hierarchical level of a column. The CTE definition contains the query blocks specifying the anchor and recursive members following the AS keyword. The connection between the query blocks occurs in the join condition and the *LevelNo* calculation in the recursive member.

Example 9.51

Basic Hierarchical Query using a Recursive CTE (Oracle)

Example 9.51 generates a result equivalent to Example 9.42. The anchor member (AM) retrieves the two roots of the hierarchical table (`FacSupervisor IS NULL`). The recursive member (RM) repeatedly executes through each level below the roots. The *LevelNo* column in the CTE is 1 for the root rows in the anchor member. The *LevelNo* value is incremented by 1 for each level below the root in the recursive member. Note that *LevelNo* is a computed column, not the pseudo column used in the proprietary Oracle notation.

```
WITH Faculty2CTE ( FacNo, FacSupervisor, FacFirstName,
 FacLastName, FacHireDate, FacRank, FacSalary, LevelNo )
AS
-- RM referencing Faculty2CTE, the recursive CTE
( SELECT FacNo, FacSupervisor, FacFirstName, FacLastName,
         FacHireDate, FacRank, FacSalary, 1
  FROM Faculty2
  WHERE FacSupervisor IS NULL
UNION ALL
 SELECT F2.FacNo, F2.FacSupervisor, F2.FacFirstName,
        F2.FacLastName, F2.FacHireDate, F2.FacRank,
        F2.FacSalary, F2CTE.LevelNo + 1
 FROM Faculty2 F2 INNER JOIN Faculty2CTE F2CTE
   ON F2.FacSupervisor = F2CTE.FacNo
)
-- Statement using the CTE
SELECT * FROM Faculty2CTE
ORDER BY LevelNo, FacNo;
```

FacNo	FacSupervisor	FacFirstName	FacLastName	FacHireDate	FacRank	FacSalary	LevelNo
543-21-0987		VICTORIA	EMMANUEL	15-APR-05	PROF	120000	1
765-43-2109		NICKI	MACON	11-APR-06	ASSC	105000	1
111-22-3333	543-21-0987	JOHN	MILLSON	01-MAY-09	PROF	110000	2
654-32-1098	543-21-0987	LEONARD	FIBON	01-MAY-03	ASSC	70000	2
777-11-4321	765-43-2109	AIMEE	MANNING	15-MAR-10	ASST	85000	2
987-65-4321	765-43-2109	JULIA	MILLS	15-MAR-09	ASSC	95000	2
098-76-5432	654-32-1098	LEONARD	VINCE	10-APR-04	ASST	55000	3
333-22-4444	111-22-3333	SALLY	SCOTT	01-MAY-10	ASST	90000	3
555-66-7777	111-22-3333	SUSAN	JONES	01-MAY-11	ASSC	125000	3
789-12-3210	987-65-4321	JAIME	SANCHEZ	10-MAY-13	PROF	107000	3
876-54-3210	654-32-1098	CRISTOPHER	COLAN	01-MAR-08	ASST	90000	3
888-33-1111	987-65-4321	JAMES	BLOKE	15-APR-12	ASST	85000	3

This subsection finishes with two path exception examples to demonstrate recursive CTE notation on useful problems. Example 9.52 lists details about faculty earning more than a supervisor at any level. Example 9.53 lists details about faculty with a higher rank than a supervising faculty member. In both examples, you should note that the second SELECT statement using the CTE contains path exception conditions.

Example 9.52

Path Exception Query using the Recursive CTE Notation (Oracle)

Example 9.52 retrieves faculty earning more than a supervisor at any level, generating the same result as Example 9.48. The join operations in each query block retrieve columns from the supervisor's row.

```
WITH Faculty2CTE ( FacNo, FacSupNo, FacLastName,
 FacSupLastName, FacSalary, FacSupSalary )
AS
( SELECT F1.FacNo, F1.FacSupervisor, F1.FacLastName,
         F1Sup.FacLastName, F1.FacSalary, F1Sup.FacSalary
  FROM Faculty2 F1 INNER JOIN Faculty2 F1Sup
    ON F1.FacSupervisor = F1Sup.FacNo
```

```
UNION ALL
 SELECT F2.FacNo, F2CTE.FacSupNo, F2.FacLastName,
        F2CTE.FacSupLastName, F2.FacSalary, F2CTE.FacSupSalary
 FROM Faculty2 F2 INNER JOIN Faculty2CTE F2CTE
   ON F2.FacSupervisor = F2CTE.FacNo )
-- Statement using the CTE with path exception condition
SELECT *
 FROM Faculty2CTE
 WHERE FacSupNo <> FacNo AND FacSalary > FacSupSalary;
```

FacNo	FacSupNo	FacLastName	FacSupLastName	FacSalary	FacSupSalary
876-54-3210	654-32-1098	COLAN	FIBON	90000	70000
789-12-3210	987-65-4321	SANCHEZ	MILLS	107000	95000
555-66-7777	543-21-0987	JONES	MILLSON	125000	110000
555-66-7777	111-22-3333	JONES	EMMANUEL	125000	120000
789-12-3210	765-43-2109	SANCHEZ	MACON	107000	105000

Example 9.53

Path Exception Query using the Recursive CTE notation (Oracle)

Example 9.52 retrieves faculty with a higher rank than a supervisor at any level, generating the same result as Example 9.49. The join operations in each query block retrieve columns from the supervisor's row.

```
WITH Faculty2CTE ( FacNo, FacSupNo, FacLastName,
 FacSupLastName, FacRank, FacSupRank)
AS
( SELECT F1.FacNo, F1.FacSupervisor, F1.FacLastName,
         F1Sup.FacLastName, F1.FacRank, F1Sup.FacRank
  FROM Faculty2 F1 INNER JOIN Faculty2 F1Sup
    ON F1.FacSupervisor = F1Sup.FacNo
UNION ALL
 SELECT F2.FacNo, F2CTE.FacSupNo, F2.FacLastName,
        F2CTE.FacSupLastName, F2.FacRank, F2CTE.FacSupRank
 FROM Faculty2 F2 INNER JOIN Faculty2CTE F2CTE
   ON F2.FacSupervisor = F2CTE.FacNo )
-- Statement using the CTE with path exception condition
SELECT *
 FROM Faculty2CTE
 WHERE FacSupNo <> FacNo
   AND ( ( FacRank = 'PROF' AND FacSupRank = 'ASSC' )
    OR ( FacRank = 'PROF' AND FacSupRank = 'ASST' )
    OR ( FacRank = 'ASSC' AND FacSupRank = 'ASST' ) );
```

FacNo	FacSupNo	FacLastName	FacSupLastName	FacRank	FacSupRank
789-12-3210	987-65-4321	SANCHEZ	MILLS	PROF	ASSC
789-12-3210	765-43-2109	SANCHEZ	MACON	PROF	ASSC

CLOSING THOUGHTS

Chapter 9 presented advanced query formulation skills with an emphasis on complex matching problems and additional parts of the SQL SELECT statement. Complex matching problems involve the outer join operator with its variations (one-sided and full), the difference operator, and the division operator. In addition to more complex matching problems, this chapter explained the subtle effects of null values to provide a deeper understanding of query results and presented hierarchical queries that support retrieval from hierarchically-structured tables.

Two new parts of the SELECT statement were covered for complex matching problems. The keywords, LEFT, RIGHT, and FULL as part of the join operator style, support outer join operations. Nested queries are a query inside another query. To understand the effect of a nested query, you should look for tables used in both an outer and an inner query. If there are no common tables, the nested query executes one time (Type I nested query). Otherwise, the nested query executes one time for each row of the outer query (Type II nested query). Type I nested queries are typically used to formulate joins as part of the SELECT, UPDATE, and DELETE statements. Type I nested queries with the NOT IN operator and Type II nested queries with the NOT EXISTS operator are useful for problems involving the difference operator. Type I nested queries in the HAVING clause are useful for problems involving the division operator.

For hierarchical queries, two SQL extensions were covered. Oracle provides proprietary notation with the CONNECT BY PRIOR clause, START WITH clause, SIBLINGS sort specification, LEVEL pseudo column, and several functions. The SQL standard notation involves recursive common table expressions involving the WITH statement containing two query blocks connected by a union operation and a SELECT statement to generate the hierarchical query results. The proprietary Oracle notation is more succinct but not portable to other DBMSs. Somewhat surprisingly, Oracle supports both its proprietary notation and the SQL standard notation.

Although advanced query skills are not as widely applied as fundamental skills covered in Chapter 4, they are important when required. You can gain a competitive advantage by mastering these advanced query formulation skills.

Chapters 4 and 9 covered important query formulation skills and a large part of the SELECT statement of SQL. Despite this significant coverage, you still have much left to learn. You need lots of practice to confidently formulate complex matching problems and hierarchical queries. In addition, you have not learned to apply your query formulation skills to building applications. Chapter 10 applies your skills to building applications with views, while Chapter 11 applies your skills to stored procedures and triggers.

REVIEW CONCEPTS

- Formulating one-sided outer joins with Access and Oracle

```
SELECT OfferNo, CourseNo, Offering.FacNo, Faculty.FacNo,
       FacFirstName, FacLastName
 FROM Offering LEFT JOIN Faculty
   ON Offering.FacNo = Faculty.FacNo
 WHERE CourseNo = 'IS480'
```

- Formulating full outer joins using the FULL JOIN keyword (SQL:2016 and Oracle)

```
SELECT FacNo, FacFirstName, FacLastName, FacSalary,
       StdNo, StdFirstName, StdLastName, StdGPA
 FROM Faculty FULL JOIN Student
   ON Student.StdNo = Faculty.FacNo
```

- Formulating full outer joins by combining two one-sided outer joins in Access

```
SELECT FacNo, FacFirstName, FacLastName, FacSalary,
       StdNo, StdFirstName, StdLastName, StdGPA
 FROM Faculty RIGHT JOIN Student
   ON Student.StdNo = Faculty.FacNo
      UNION
SELECT FacNo, FacFirstName, FacLastName, FacSalary,
       StdNo, StdFirstName, StdLastName, StdGPA
 FROM Faculty LEFT JOIN Student
   ON Student.StdNo = Faculty.FacNo
```

- Mixing inner and outer joins (Access and Oracle)

```
SELECT OfferNo, Offering.CourseNo, OffTerm, CrsDesc,
       Faculty.FacNo, FacFirstName, FacLastName
 FROM ( Faculty RIGHT JOIN Offering
   ON Offering.FacNo = Faculty.FacNo )
  INNER JOIN Course
   ON Course.CourseNo = Offering.CourseNo
 WHERE OffYear = 2017
```

- Ambiguous query containing a non-preserved table (table with only matching rows in the result) in a one-sided outer join involved in another join or outer join operation
- Understanding that conditions in the WHERE or HAVING clause can use SELECT statements in addition to scalar (individual) values
- Identifying Type I nested queries by the IN keyword and the lack of a reference to a table used in an outer query
- Using a Type I nested query to formulate a join

```
SELECT DISTINCT StdNo, StdFirstName, StdLastName,
        StdMajor
 FROM Student
 WHERE Student.StdNo IN
 ( SELECT StdNo FROM Enrollment
     WHERE EnrGrade >= 3.5 )
```

- Using a Type I nested query inside a DELETE statement to test conditions on a related table

```
DELETE FROM Offering
 WHERE Offering.FacNo IN
 ( SELECT FacNo FROM Faculty
     WHERE FacFirstName = 'LEONARD'
       AND FacLastName = 'VINCE' )
```

- Using a Type I nested query inside an UPDATE statement to test conditions on a related table

```
UPDATE Offering SET OffLocation = 'BLM412'
 WHERE OffYear = 2017 AND FacNo IN
 ( SELECT FacNo FROM Faculty
     WHERE FacFirstName = 'LEONARD'
       AND FacLastName = 'FIBON')
```

- Not using a Type I nested query for a join when a column from the nested query is needed in the final query result
- Identifying problem statements involving the difference operator: the words *not* or *only* relating two nouns in a sentence
- Limited SQL formulations for difference problems: Type I nested queries with the NOT IN operator, one-sided outer join with an IS NULL condition, and difference operation using the EXCEPT or MINUS keywords

- Using a Type I nested query with the NOT IN operator for difference problems involving a comparison of a single column

```
SELECT FacNo, FacFirstName, FacLastName, FacDept, FacSalary
 FROM Faculty
 WHERE FacNo NOT IN
 ( SELECT StdNo FROM Student )
```

- Identifying Type II nested queries by a reference to a table used in an outer query
- Using Type II nested queries with the NOT EXISTS operator for complex difference problems

```
SELECT FacNo, FacFirstName, FacLastName, FacDept, FacSalary
 FROM Faculty
 WHERE NOT EXISTS
 ( SELECT * FROM Student
     WHERE Student.StdNo = Faculty.FacNo )
```

- Using a nested query in the FROM clause to compute nested aggregates or aggregates for more than one grouping

```
SELECT T.CourseNo, T.CrsDesc, COUNT(*) AS NumOfferings,
       AVG(T.EnrollCount) AS AvgEnroll
 FROM
 ( SELECT Course.CourseNo, CrsDesc,
          Offering.OfferNo, COUNT(*) AS EnrollCount
   FROM Offering, Enrollment, Course
   WHERE Offering.OfferNo = Enrollment.OfferNo
     AND Course.CourseNo = Offering.CourseNo
   GROUP BY Course.CourseNo, CrsDesc, Offering.OfferNo
   ) T
 GROUP BY T.CourseNo, T.CrsDesc
```

- Identifying problem statements involving the division operator: the word every or all connecting different parts of a sentence
- Using the count method to formulate division problems

```
SELECT StdNo
 FROM StdClub
 GROUP BY StdNo
 HAVING COUNT(*) = ( SELECT COUNT(*) FROM Club )
```

- Evaluating a simple condition containing a null value in a column expression
- Using three-valued logic and truth tables to evaluate compound conditions with null values
- Understanding the result of aggregate calculations with null values
- Understanding the result of grouping on a column with null values
- Recognizing the need to formulate hierarchical queries for tables with hierarchical data
- Using the CONNECT BY PRIOR and START WITH clauses to formulate basic hierarchical queries

```
SELECT FacNo, FacSupervisor, FacFirstName, FacLastName,
       FacHireDate, FacSalary, FacRank, LEVEL
 FROM Faculty2
 START WITH FacSupervisor IS NULL
 CONNECT BY PRIOR FacNo = FacSupervisor
 ORDER BY LEVEL;
```

- Applying the proprietary Oracle syntax elements including the LEVEL pseudo column, CONNECT_BY_ROOT operator, SYS_CONNECT_BY_PATH function, CONNECT_BY_ISLEAF pseudo column, and SIBLINGS keyword to formulate more complex hierarchical queries
- Formulating path exception queries listing violations of monotonicity in path relationships such as subordinates earning more than their direct or indirect supervisors. Path exception queries use the closure of the hierarchy.
- Recognizing recursive common table expressions, the SQL standard notation for formulating hierarchical queries on enterprise DBMSs

QUESTIONS

1. Explain a situation when a one-sided outer join is useful.
2. Explain a situation when a full outer join is useful.
3. How do you interpret the meaning of the LEFT and RIGHT JOIN keywords in the FROM clause?
4. What is the interpretation of the FULL JOIN keywords in the FROM clause?
5. How do you perform a full outer join in SQL implementations (such as Microsoft Access) that do not support the FULL JOIN keywords?
6. What is a nested query?
7. What is the distinguishing feature about the appearance of Type I nested queries?
8. What is the distinguishing feature about the appearance of Type II nested queries?
9. How many times is a Type I nested query executed as part of an outer query?
10. How is a Type I nested query like a procedure in a computer program?
11. How many times is a Type II nested query executed as part of an outer query?
12. How is a Type II nested query like a nested loop in a computer program?
13. What is the meaning of the IN comparison operator?
14. What is the meaning of the EXISTS comparison operator?
15. What is the meaning of the NOT EXISTS comparison operator?
16. When can you not use a Type I nested query to perform a join?
17. Why is a Type I nested query a good join method when you need a join in a DELETE or UPDATE statement?
18. Why does SQL:2016 permit nested queries in the FROM clause?
19. Identify two situations in which nested queries in the FROM clause are necessary.
20. How do you detect that a problem involves a division operation?
21. Explain the "count" method for formulating division problems.
22. Why is it sometimes necessary to use the DISTINCT keyword inside the COUNT function for division problems?
23. What is the result of a simple condition when a column expression in the condition evaluates to null?
24. What is a truth table?
25. How many values do truth tables have in the SQL:2016 standard?
26. How do you use truth tables to evaluate compound conditions?
27. How do null values affect aggregate calculations?

28. Explain why the following equation may not be true if Column1 or Column2 contains null values: `SUM(Column1) – SUM(Column2) = SUM(Column1 – Column2)`
29. How are null values handled in a grouping column?
30. In Access, how do you compensate for the lack of the DISTINCT keyword inside the COUNT function?
31. When can you use a Type I nested query with the NOT IN operator to formulate a difference operation in SQL?
32. When can you use a one-sided outer join with an IS NULL condition to formulate a difference operation in SQL?
33. When can you use a MINUS operation in SQL to formulate a difference operation in SQL?
34. What is the most general way to formulate difference operations in SQL statements?
35. Is the one-sided outer join operator associative?
36. What makes a query ambiguous?
37. What is the difference between Microsoft Access and Oracle in handling ambiguous queries?
38. What is a hierarchical query?
39. What is the difference between a self-join and a hierarchical query?
40. What is an important advantage for query language support for hierarchical queries?
41. What is a path exception query?
42. Explain the usage of the CONNECT BY PRIOR and START WITH clauses.
43. Explain the usage of the CONNECT_BY_ROOT operator and the SYS_CONNECT_BY_PATH function.
44. Explain the usage of the LEVEL pseudo column and the SIBLINGS keyword.
45. What is a recursive common table expression?
46. Briefly compare the proprietary Oracle notation for hierarchical queries to recursive common table expressions.
47. Explain the difference between the PRIOR operator and the CONNECT_BY_ROOT operator.
48. What is the limitation of using a one-sided outer is an IS NULL condition for a difference problem?
49. What is the limitation of using a Type I nested query with the NOT IN operator for a difference problem?
50. What is the limitation of using a Type I nested query with the NOT IN operator for a difference problem?

PROBLEMS

The problems use the tables of the Order Entry database introduced in the Problems section of Chapter 4. When formulating the problems, remember that the *EmpNo* foreign key in the *OrderTbl* table allows null values. An order does not have an associated employee if taken over the Internet.

1. Using a Type I nested query, list the customer number, name (first and last), and city of each customer who has a balance greater than $150 and placed an order in February 2017.

2. Using a Type II nested query, list the customer number, name (first and last), and city of each customer who has a balance greater than $150 and placed an order in February 2017.
3. Using two Type I nested queries, list the product number, the name, and the price of products with a price greater than $150 that were ordered on January 23, 2017.
4. Using two Type I nested queries and another join style, list the product number, name, and price of products with a price greater than $150 that were ordered in January 2017 by customers with balances greater than $400.
5. List the order number, order date, employee number, and employee name (first and last) of orders placed on January 23, 2017. List the order even if there is not an associated employee.
6. List the order number, order date, employee number, employee name (first and last), customer number, and customer name (first and last) of orders placed on January 23, 2017. List the order even if there is not an associated employee.
7. List all the people in the database. The resulting table should have all columns of the *Customer* and *Employee* tables. Match the *Customer* and *Employee* tables on first and last names. If a customer does not match any employees, the columns pertaining to the *Employee* table will be blank. Similarly for an employee who does not match any customers, the columns pertaining to the *Customer* table will be blank.
8. For each Ink Jet product ordered in January 2017, list the order number, order date, customer number, customer name (first and last), employee number (if present), employee name (first and last), quantity ordered, product number, and product name. Include products containing Ink Jet in the product name. Include both Internet (no employee) and phone orders (taken by an employee).
9. Using a Type II nested query, list the customer number and name of Colorado customers who have not placed orders in February 2017.
10. Repeat problem 9 using a Type I nested query with a NOT IN condition instead of a nested query. If the problem cannot be formulated in this manner, provide an explanation indicating the reason.
11. Repeat problem 9 using the MINUS keyword. Note that Access does not support the MINUS keyword. If the problem cannot be formulated in this manner, provide an explanation indicating the reason.
12. Repeat problem 9 using a one-sided outer join and an IS NULL condition. If the problem cannot be formulated in this manner, provide an explanation indicating the reason.
13. Using a Type II nested query, list the employee number, first name, and last name of employees in the (720) area code who have not taken orders. An employee is in the (720) area code if the employee phone number contains the string (720) in the beginning of the column value.
14. Repeat problem 13 using a Type I nested query with a NOT IN condition instead of a nested query. If the problem cannot be formulated in this manner, provide an explanation indicating the reason. (Hint: you need to think carefully about the effect of null values in the *OrderTbl.EmpNo* column.)
15. Repeat problem 9 using a one-sided outer join and an IS NULL condition. If the problem cannot be formulated in this manner, provide an explanation indicating the reason.
16. Repeat problem 9 using the MINUS keyword. Note that Access does not support the MINUS keyword. If the problem cannot be formulated in this manner, provide an explanation indicating the reason.

17. List the order number and order date of orders containing only one product with the words Ink Jet in the product description.
18. List the customer number and name (first and last) of customers who have ordered products only manufactured by Connex. Include only customers who have ordered at least one product manufactured by Connex. Remove duplicate rows from the result.
19. List the order number and order date of orders containing every product with the words Ink Jet in the product description.
20. List the product number and name of products contained on every order placed on January 7, 2017 through January 9, 2017.
21. List the customer number and name (first and last) of customers who have ordered every product manufactured by ColorMeg, Inc. in January 2017.
22. Using a Type I nested query, delete orders placed by customer Betty Wise in January 2017. The CASCADE DELETE action will delete related rows in the *OrdLine* table.
23. Using a Type I nested query, delete orders placed by Colorado customers that were taken by Landi Santos in January 2017. The CASCADE DELETE action will delete related rows in the *OrdLine* table.
24. List the order number and order date of orders in which any part of the shipping address (street, city, state, and zip) differs from the customer's address.
25. List the employee number and employee name (first and last) of employees who have taken orders in January 2017 from every Seattle customer.
26. For Colorado customers, compute the average amount of their orders. The average amount of a customer's orders is the sum of the amount (quantity ordered times the product price) on each order divided by the number of orders. The result should include the customer number, customer last name, and average order amount.
27. For Colorado customers, compute the average amount of their orders and the number of orders placed. The result should include the customer number, customer last name, average order amount, and number of orders placed. In Access, this problem is especially difficult to formulate.
28. For Colorado customers, compute the number of unique products ordered. If a product is purchased on multiple orders, it should be counted only one time. The result should include the customer number, customer last name, and number of unique products ordered.
29. For each employee with a commission less than 0.04, compute the number of orders taken and the average number of products per order. The result should include the employee number, employee last name, number of orders taken, and the average number of products per order. In Access, this problem is especially difficult to formulate as a single SELECT statement.
30. For each Connex product, compute the number of unique customers who ordered the product in January 2017. The result should include the product number, product name, and number of unique customers.
31. Please explain if the following SELECT statement is ambiguous. If it is ambiguous, provide a variation of this statement with possibly different results. The variation should be the identical SQL statement except for the order of operations in the FROM clause.

```
SELECT OrderTbl.OrdNo, OrdDate, Employee.EmpNo,
       EmpFirstName, EmpLastName, Customer.CustNo,
```

```
       CustFirstName, CustLastName, OrdLine.Qty,
       Product.ProdNo, ProdName
FROM ( ( ( OrderTbl LEFT JOIN Employee
       ON OrderTbl.EmpNo = Employee.EmpNo )
   INNER JOIN Customer
    ON Customer.CustNo = OrderTbl.CustNo )
   INNER JOIN OrdLine
    ON OrderTbl.OrdNo = OrdLine.OrdNo )
   INNER JOIN Product
    ON OrdLine.ProdNo = Product.ProdNo
```

32. Please explain if the following SELECT statement is ambiguous. If it is ambiguous, provide a variation of this statement with possibly different results. The variation should be the identical SQL statement except for the order of operations in the FROM clause.

```
SELECT OrderTbl.OrdNo, OrdDate, Employee.EmpNo,
       EmpFirstName, EmpLastName, Customer.CustNo,
       CustFirstName, CustLastName, OrdLine.Qty,
       Product.ProdNo, ProdName
FROM ( ( ( OrderTbl RIGHT JOIN Employee
       ON OrderTbl.EmpNo = Employee.EmpNo )
   INNER JOIN Customer
    ON Customer.CustNo = OrderTbl.CustNo )
   INNER JOIN OrdLine
    ON OrderTbl.OrdNo = OrdLine.OrdNo )
   INNER JOIN Product
    ON OrdLine.ProdNo = Product.ProdNo
```

33. Add 1 to the order quantity of each product ordered by customer number C9943201 on January 23, 2017. In the UPDATE statement, you should not use an order number constant. You need to reference the related tables in the UPDATE statement. Write the UPDATE statement in both Access and Oracle using a Type I nested query.
34. Revise problem 33 to add 1 to the order quantity of each product ordered by Harry Sanders on January 23, 2017. In the UPDATE statement, you should not use an order number or customer number constant. You need to reference the related tables in the UPDATE statement. Write the UPDATE statement in both Access and Oracle using a Type I nested query.
35. Revise problem 33 to add 1 to the order quantity of each product with product name containing the string "Color Inkjet" ordered by Harry Sanders on January 23, 2017. In the UPDATE statement, you should not use an order number, customer number, or product number constant. You need to reference the related tables in the UPDATE statement. Write the UPDATE statement in both Access and Oracle using a Type I nested query.

Null Value Problems

The following problems are based on the *Product* and *Employee* tables of the Order Entry database. The tables are repeated below for your convenience. The *ProdNextShipDate* column contains the next expected shipment date for the product. If the value is null, a new shipment has not been arranged. A shipment may not be scheduled for a variety of reasons, such as the large quantity on hand or unavailability of the product from the manufacturer. In the *Employee* table, the commission rate can be null indicating a commission rate has not been assigned. A null value for *SupEmpNo* indicates that the employee has no supervisor.

Product

ProdNo	ProdName	ProdMfg	ProdQOH	ProdPrice	ProdNextShipDate
P0036566	17 inch Color Monitor	ColorMeg, Inc.	12	$169.00	2/20/2017
P0036577	19 inch Color Monitor	ColorMeg, Inc.	10	$319.00	2/20/2017
P1114590	R3000 Color Laser Printer	Connex	5	$699.00	1/22/2017
P1412138	10 Foot Printer Cable	Ethlite	100	$12.00	
P1445671	8-Outlet Surge Protector	Intersafe	33	$14.99	
P1556678	CVP Ink Jet Color Printer	Connex	8	$99.00	1/22/2017
P3455443	Color Ink Jet Cartridge	Connex	24	$38.00	1/22/2017
P4200344	36-Bit Color Scanner	UV Components	16	$199.99	1/29/2017
P6677900	Black Ink Jet Cartridge	Connex	44	$25.69	
P9995676	Battery Back-up System	Cybercx	12	$89.00	2/1/2017

Employee

EmpNo	EmpFirstName	EmpLastName	EmpPhone	EmpEMail	SupEmpNo	EmpCommRate
E1329594	Landi	Santos	(303) 789-1234	LSantos@bigco.com	E8843211	0.02
E8544399	Joe	Jenkins	(303) 221-9875	JJenkins@bigco.com	E8843211	0.02
E8843211	Amy	Tang	(303) 556-4321	ATang@bigco.com	E9884325	0.04
E9345771	Colin	White	(303) 221-4453	CWhite@bigco.com	E9884325	0.04
E9884325	Thomas	Johnson	(303) 556-9987	TJohnson@bigco.com		0.05
E9954302	Mary	Hill	(303) 556-9871	MHill@bigco.com	E8843211	0.02
E9973110	Theresa	Beck	(720) 320-2234	TBeck@bigco.com	E9884325	

1. Identify the result rows in the following SELECT statement. Both Access and Oracle versions of the statement are shown.

   Access:
   ```
   SELECT *
    FROM Product
    WHERE ProdNextShipDate = #1/22/2017#
   ```
   Oracle:
   ```
   SELECT *
    FROM Product
    WHERE ProdNextShipDate = '22-Jan-2017';
   ```
2. Identify the result rows in the following SELECT statement:

   Access:
   ```
   SELECT *
    FROM Product
    WHERE ProdNextShipDate = #1/22/2017#
      AND ProdPrice < 100
   ```
   Oracle:
   ```
   SELECT *
    FROM Product
    WHERE ProdNextShipDate = '22-Jan-2017'
      AND ProdPrice < 100;
   ```

3. Identify the result rows in the following SELECT statement:

   Access:
   ```
   SELECT *
    FROM Product
    WHERE ProdNextShipDate = #1/22/2017#
       OR ProdPrice < 100
   ```
   Oracle:
   ```
   SELECT *
    FROM Product
    WHERE ProdNextShipDate = '22-Jan-2017'
       OR ProdPrice < 100;
   ```
4. Determine the result of the following SELECT statement:
   ```
   SELECT COUNT(*) AS NumRows,
          COUNT(ProdNextShipDate) AS NumShipDates
    FROM Product
   ```
5. Determine the result of the following SELECT statement:
   ```
   SELECT ProdNextShipDate, COUNT(*) AS NumRows
    FROM Product
    GROUP BY ProdNextShipDate
   ```
6. Determine the result of the following SELECT statement:
   ```
   SELECT ProdMfg, ProdNextShipDate, COUNT(*) AS NumRows
    FROM Product
    GROUP BY ProdMfg, ProdNextShipDate
   ```
7. Determine the result of the following SELECT statement:
   ```
   SELECT ProdNextShipDate, ProdMfg, COUNT(*) AS NumRows
    FROM Product
    GROUP BY ProdNextShipDate, ProdMfg
   ```
8. Identify the result rows in the following SELECT statement:
   ```
   SELECT EmpFirstName, EmpLastName
    FROM Employee
    WHERE EmpCommRate > 0.02
   ```
9. Determine the result of the following SELECT statement:
   ```
   SELECT SupEmpNo, AvG(EmpCommRate) AS AvgCommRate
    FROM Employee
    GROUP BY SupEmpNo
   ```
10. Compare the result of the following SELECT statement to the result of problem 9. What result row in problem 9 does not appear in problem 10's result? Explain why the row does not appear in problem 10's result. The SELECT statement for problem 10 computes the average commission rate of subordinate employees. The result includes the employee number, first name, and last name of the supervising employee as well as the average commission amount of the subordinate employees.
    ```
    SELECT Emp.SupEmpNo, Sup.EmpFirstName, Sup.EmpLastName,
           AVG(Emp.EmpCommRate) AS AvgCommRate
    FROM Employee Emp, Employee Sup
    WHERE Emp.SupEmpNo = Sup.EmpNo
    GROUP BY Emp.SupEmpNo, Sup.EmpFirstName, Sup.EmpLastName
    ```
11. Using your knowledge of null value evaluation, explain why these two SQL statements generate different results for the Order Entry Database. You should remember that null values are allowed for *OrderTbl.EmpNo*.
    ```
    SELECT EmpNo, EmpLastName, EmpFirstName
     FROM Employee
    ```

```
WHERE EmpNo NOT IN
( SELECT EmpNo FROM OrderTbl WHERE EmpNo IS NOT NULL )

SELECT EmpNo, EmpLastName, EmpFirstName
FROM Employee
WHERE EmpNo NOT IN
( SELECT EmpNo FROM OrderTbl )
```

12. Using problem 11 as an example, explain the impact of using a Type I nested query for a difference problem. (Hint: this explanation involves null values.)

Hierarchical Query Problems

The following problems use the *Employee2* table, extended from the *Employee* table of the Order Entry database. Some of the columns have been dropped and others added for these problems. Here are comments about the extensions to the *Employee* table.

- As in the *Employee* table, *EmpNo* is the primary key.
- As in the *Employee* table, the *SupEmpNo* column is a foreign key referencing the *EmpNo* column.
- The *EmpSalary* column should be smaller for subordinates than supervisors, both direct and indirect.
- The *EmpGrade* column should be larger for subordinates than supervisors, both direct and indirect.
- The *EmpCommRate* should be larger for subordinates than supervisors, both direct and indirect.

1. Draw organizational charts, similar to Figures 9.2 and 9.3, to depict the hierarchical organization among rows in the *Employee2* table.
2. Using the Oracle proprietary notation, write a SELECT statement to retrieve the closure (combinations of employee and supervisor, direct or indirect) starting with the root employees having null values for *SupEmpNo*. The result should contain *EmpNo*, *EmpLastName*, *EmpSalary*, *EmpGrade*, *SupEmpNo*, *EmpCommRate*, root employee number, and the LEVEL pseudo column. Order the result by *EmpLastName* and the LEVEL pseudo column. Note that you will need

Employee2

EmpNo	EmpFirstName	EmpLastName	EmpSalary	EmpGrade	SupEmpNo	EmpCommRate
E1329594	Landi	Santos	36000	2	E8843211	0.050
E8544399	Joe	Jenkins	30000	4	E8843211	0.040
E8843211	Amy	Tang	35000	3	E9884325	0.030
E9345771	Colin	White	40000	2	E9884325	0.040
E9884325	Thomas	Johnson	60000	2		0.035
E9954302	Mary	Hill	37000	3	E8843211	0.050
E9973110	Theresa	Beck	42000	1	E9884325	0.033
E1234567	Claire	Adams	50000	1		0.025
E7654321	Yanjuan	Pong	40000	3	E1234567	0.030
E4321098	Miguel	Sanchez	52000	2	E1234567	0.033
E6543210	Bradley	Smith	35000	3	E7654321	0.045
E5432109	Susan	Henry	41000	2	E7654321	0.050
E9876543	Michael	Roberts	55000	2	E4321098	0.040
E8765432	Melissa	Cole	42000	3	E4321098	0.033

to rename the LEVEL pseudo column in the output list to reference it in the ORDER BY clause.

3. Using the Oracle proprietary notation, write a SELECT statement to retrieve the closure (combinations of employee and supervisor, direct or indirect) starting with the root employees having null values for *SupEmpNo*. The result should contain *EmpNo*, *EmpLastName*, root employee number, and path using last name as the row identifier and / as the separator. Sort the siblings by employee last name.
4. Using the Oracle proprietary notation, write a SELECT statement to retrieve the closure (combinations of employee and supervisor, direct or indirect) starting with the root employees having null values for *SupEmpNo*. The result should contain *EmpLastName* arranged to depict the hierarchical structure using the LPAD function (See Example 9.43.), *EmpNo*, *EmpSalary*, *EmpGrade*, and *EmpCommRate*. Sort the siblings by employee last name.
5. Using the Oracle proprietary notation, summarize each supervisor (non-leaf row) on the number of subordinates (direct and indirect) and sum of the salary of the subordinates. The result should include the employee last name, sum of the salary, and number (count) of subordinates (both direct and indirect). Only include non-leaf nodes in the final result.
6. Using the Oracle proprietary notation, list details about employees with a larger salary than a supervisor, direct or indirect. The result should include the employee number, last name, and salary of both the employee and supervisor as well as the path from the supervisor to the employee using the last name to identify rows on the path and / as the separator character.
7. Using the Oracle proprietary notation, list details about employees with a smaller grade than a supervisor, direct or indirect. The result should include the employee number, last name, and grade of both the employee and supervisor as well as the path from the supervisor to the employee using the last name to identify rows on the path and / as the separator character.
8. Using the Oracle proprietary notation, list details about employees with a smaller commission rate than a supervisor, direct or indirect. The result should include the employee number, last name, and commission rates of the employee and supervisor as well as the path from the supervisor to the employee using the last name to identify rows on the path and / as the separator character.
9. Using the Oracle proprietary notation, summarize the commission amounts earned on January 2017 sales for each employee supervised by Johnson either directly or indirectly. The earned commission is calculated by the employee's commission rate times the amount of sales on orders taken by the employee. The amount of sales on an order is calculated by summing the quantity ordered times price for each product on an order. You should combine the *Employee2* and *OrderTbl* tables on employee number to link employees with orders. The result should include the employee number, employee last name, last name of the employee's direct supervisor, hierarchical level, and sum of the earned commission.
10. Using the recursive CTE notation, list details about employees with a larger salary than a supervisor, direct or indirect. The result should include the employee number, last name, and salary of both the employee and supervisor.
11. Using the recursive CTE notation list details about employees with a smaller grade than a supervisor, direct or indirect. The result should include the employee number, last name, and grade of both the employee and supervisor.
12. Using the recursive CTE notation, list details about employees with a smaller commission than a supervisor, direct or indirect. The result should include the employee number, last name, and commission of both the employee and supervisor.

REFERENCES FOR FURTHER STUDY

Most textbooks for the business student do not cover query formulation and SQL in as much detail as here. For resources about the SQL standard, you should consult the SQL standards page (www.jcc.com/resources/sql-standards) in JCC Consulting website. For new features in the latest SQL standard, you should consult modern SQL (modern-sql.com). For product-specific SQL advice, the sqlblog.com site features forums about a number of DBMSs including Microsoft SQL Server and open source products. The Database Journal (www.databasejournal.com) provides articles, tutorials, and resources about many DBMS products. Oracle documentation can be found at the Oracle Technet site (www.oracle.com/technetwork). The Mimer Developer website has validators (developer.mimer.se/validator) for the SQL standards as aids to writing portable SQL statements.

chapter

10

Application Development with Views

Learning Objectives

This chapter describes underlying concepts for views and demonstrates usage of views in forms and reports. After this chapter, the student should have acquired the following knowledge and skills:

- Write CREATE VIEW statements
- Write queries that use views
- Explain basic ideas about the modification and materialization approaches for processing queries using views
- Apply rules to determine if single-table and multiple-table views are updatable
- Determine data requirements for hierarchical forms
- Write queries that provide data for hierarchical forms
- Write queries that provide data for hierarchical reports

OVERVIEW

Chapters 3, 4, and 9 provided the foundation for understanding relational databases and formulating queries in SQL. Most importantly, you gained practice with a large number of examples, acquired problem-solving skills for query formulation, and learned different parts of the SELECT statement. This chapter extends your query formulation skills to building applications with views.

This chapter emphasizes views as the foundation for building database applications. Before discussing the link between views and database applications, essential background is provided. You will learn the motivation for views, the CREATE VIEW statement, and usage of views in the SELECT statement and data manipulation (INSERT, UPDATE, and DELETE) statements. Most view examples in Sections 10.2 and 10.3 are supported by both Microsoft Access and Oracle. After this background, you will learn to use views for hierarchical forms and reports. You will learn the steps for analyzing data requirements culminating in views to support the data requirements.

The presentation in Sections 10.1 and 10.2 covers core features in SQL:2016 that were part of SQL-92. Some rules about updatable views in Sections 10.3 and 10.4 are specific to Microsoft Access due to the varying support among DBMSs and the strong support available in Access. Appendix 10.B provides an alternative perspective with updatability rules in Oracle.

10.1 BACKGROUND

View
a table derived from base or physical tables using a query.

A **view** is a virtual or derived table. Virtual means that a view behaves like a base table but no physical table exists. A view can be used in a query like a base table. However, the rows of a view do not exist until they are derived from base tables. This section describes the importance of views and demonstrates view definition in SQL.

10.1.1 Motivation

Views provide the external level of the Three Schema Architecture described in Chapter 1. The Three Schema Architecture promotes data independence to reduce the impact of database definition changes on applications that use a database. Because database definition changes are common, reducing the impact of database definition changes is important to control the cost of software maintenance. Views provide compartmentalization of database requirements so that database definition changes do not affect applications using a view. If an application accesses a database through a view, most changes to the conceptual schema will not affect the application. For example, if you change a table name used in a view changes, you must change the view definition, but applications using the view do not change.

Simplification of tasks is another important benefit of views. Many queries can be easier to formulate if a view is used rather than base tables. Without a view, a SELECT statement may involve two, three, or more tables and require grouping if summary data are needed. With a view, the SELECT statement can just reference a view without joins or grouping. Training users to write single table queries is much easier than training them to write multiple table queries with grouping.

Views provide simplification, similar to macros in programming languages and spreadsheets. A macro is a named collection of commands. Using a macro removes the burden of specifying the commands. In a similar manner, using a view removes the burden of writing the underlying query.

Views also provide a flexible level of security. Restricting access by views is more flexible than restrictions for columns and tables because a view is any derived part of a database. Data not in the view are hidden from a user. For example, you can restrict a user to selected departments, products, or geographic regions in a view. Security using tables and columns cannot specify conditions and computations, which can be done in a view. A view even can include aggregate calculations to restrict users to row summaries rather than individual rows.

The only drawback to views can be performance. For most views, using the views instead of base tables directly will not involve a noticeable performance penalty. For some complex views, using the views can involve a significant performance penalty as opposed to using the base tables directly. The performance penalty can vary by DBMS. Before using complex views, you are encouraged to compare performance to usage of the base tables directly.

10.1.2 View Definition

Defining a view is only slightly more difficult than writing a query. SQL provides the CREATE VIEW statement in which a view name and a SELECT statement must be specified, as shown in Examples 10.1 and 10.2. In Oracle, the CREATE VIEW statement executes directly. In Microsoft Access, the CREATE VIEW statement can be used in SQL-92 query mode[1]. In SQL-89 query mode, the SELECT statement part of the examples can be saved as a stored query to achieve the same effect as a view. You create a stored query simply by writing it and then supplying a name when saving it.

[1] SQL-89 is the default query mode in Microsoft Access. The query mode can be changed using the Options window.

Example 10.1

Define a Single Table View

Define a view named *IS_View* consisting of students majoring in IS.

```
CREATE VIEW IS_View AS
 SELECT * FROM Student
  WHERE StdMajor = 'IS'
```

StdNo	StdFirstName	StdLastName	StdCity	StdState	StdZip	StdMajor	StdClass	StdGPA
123-45-6789	HOMER	WELLS	SEATTLE	WA	98121-1111	IS	FR	3.00
345-67-8901	WALLY	KENDALL	SEATTLE	WA	98123-1141	IS	SR	2.80
567-89-0123	MARIAH	DODGE	SEATTLE	WA	98114-0021	IS	JR	3.60
876-54-3210	CRISTOPHER	COLAN	SEATTLE	WA	98114-1332	IS	SR	4.00
890-12-3456	LUKE	BRAZZI	SEATTLE	WA	98116-0021	IS	SR	2.20
901-23-4567	WILLIAM	PILGRIM	BOTHELL	WA	98113-1885	IS	SO	3.80

Example 10.2

Define a Multiple Table View

Define a view named *MS_View* consisting of offerings taught by faculty in the Management Science department.

```
CREATE VIEW MS_View AS
 SELECT OfferNo, Offering.CourseNo, CrsUnits, OffTerm,
        OffYear, Offering.FacNo, FacFirstName,
        FacLastName, OffTime, OffDays
  FROM Faculty, Course, Offering
  WHERE FacDept = 'MS'
    AND Faculty.FacNo = Offering.FacNo
    AND Offering.CourseNo = Course.CourseNo
```

OfferNo	CourseNo	CrsUnits	OffTerm	OffYear	FacNo	FacFirstName	FacLastName	OffTime	OffDays
1234	IS320	4	FALL	2016	098-76-5432	LEONARD	VINCE	10:30 AM	MW
3333	IS320	4	SPRING	2017	098-76-5432	LEONARD	VINCE	8:30 AM	MW
4321	IS320	4	FALL	2016	098-76-5432	LEONARD	VINCE	3:30 PM	TTH
4444	IS320	4	WINTER	2017	543-21-0987	VICTORIA	EMMANUEL	3:30 PM	TTH
8888	IS320	4	SUMMER	2017	654-32-1098	LEONARD	FIBON	1:30 PM	MW
9876	IS460	4	SPRING	2017	654-32-1098	LEONARD	FIBON	1:30 PM	TTH
5679	IS480	4	SPRING	2017	876-54-3210	CRISTOPHER	COLAN	3:30 PM	TTH

In the CREATE VIEW statement, a list of column names, enclosed in parentheses, can follow the view name. A list of column names is required when you want to rename one or more columns from the names used in the SELECT clause. The column list is omitted in *MS_View* because there are no renamed columns. The column list is required in Example 10.3a to rename the aggregate calculation (COUNT(*)) column. If one column is renamed, the entire list of column names must be given. Alternatively, renaming can be selectively done in the result list as shown in Example 10.3b.

Example 10.3a

Define a View with Renamed Columns

Define a view named *Enrollment_View* consisting of offering data and the number of students enrolled.

```
CREATE VIEW  Enrollment_View
(OfferNo, CourseNo, Term, Year, Instructor, NumStudents)
 AS
 SELECT Offering.OfferNo, CourseNo, OffTerm, OffYear,
        FacLastName, COUNT(*)
  FROM Offering, Faculty, Enrollment
  WHERE Offering.FacNo  =  Faculty.FacNo
    AND Offering.OfferNo  =  Enrollment.OfferNo
  GROUP BY Offering.OfferNo, CourseNo, OffTerm, OffYear,
           FacLastName
```

OfferNo	CourseNo	Term	Year	Instructor	NumStudents
1234	IS320	FALL	2016	VINCE	6
4321	IS320	FALL	2016	VINCE	6
5555	FIN300	WINTER	2017	MACON	2
5678	IS480	WINTER	2017	MILLS	5
5679	IS480	SPRING	2017	COLAN	6
6666	FIN450	WINTER	2017	MILLS	2
7777	FIN480	SPRING	2017	MACON	3
9876	IS460	SPRING	2017	FIBON	7

Example 10.3b

Define a View with Renamed Columns in the SELECT Clause

Define a view named *Enrollment_View1* consisting of offering data and the number of students enrolled. The result is identical to Example 10.3a.

```
CREATE VIEW  Enrollment_View1 AS
 SELECT Offering.OfferNo, CourseNo, OffTerm, OffYear,
        FacLastName AS Instructor,
        COUNT(*) AS NumStudents
  FROM Offering, Faculty, Enrollment
  WHERE Offering.FacNo  =  Faculty.FacNo
    AND Offering.OfferNo  =  Enrollment.OfferNo
  GROUP BY Offering.OfferNo, CourseNo, OffTerm, OffYear,
           FacLastName
```

10.2 USING VIEWS FOR RETRIEVAL

This section shows examples of queries that use views and explains processing of queries with views. After showing examples in Section 10.2.1, two methods to process queries with views are described in Section 10.2.2.

10.2.1 Using Views in SELECT Statements

Once a view is defined, it can be used in SELECT statements. You simply use the view name in the FROM clause and the view columns in other parts of the statement. You can add other conditions and select a subset of the columns as demonstrated in Examples 10.4 and 10.5.

Example 10.4 (Oracle)

Query Using a Multiple Table View

List the spring 2017 courses in *MS_View*.

```
SELECT OfferNo, CourseNo, FacFirstName, FacLastName,
       OffTime, OffDays
 FROM MS_View
 WHERE OffTerm = 'SPRING' AND OffYear = 2017
```

OfferNo	CourseNo	FacFirstName	FacLastName	OffTime	OffDays
3333	IS320	LEONARD	VINCE	8:30 AM	MW
9876	IS460	LEONARD	FIBON	1:30 PM	TTH
5679	IS480	CRISTOPHER	COLAN	3:30 PM	TTH

Example 10.5 (Oracle)

Query Using a Grouping View

List the spring 2017 offerings of IS courses in the Enrollment_View. In Access, you need to substitute the * for % as the wildcard symbol.

```
SELECT OfferNo, CourseNo, Instructor, NumStudents
 FROM Enrollment_View
 WHERE Term = 'SPRING' AND Year = 2017
   AND CourseNo LIKE 'IS%'
```

OfferNo	CourseNo	Instructor	NumStudents
5679	IS480	COLAN	6
9876	IS460	FIBON	7

Both queries are much easier to write than the original queries. A novice user can probably write both queries with just a little training. In contrast, it may take many hours of training for a novice user to write queries with multiple tables and grouping.

According to SQL:2016, a view can be used in any query. In practice, most DBMSs have some limitations on view usage in queries. For example, some DBMSs do not support the queries[2] shown in Examples 10.6 and 10.7.

View Materialization
a method to process a query on a view by executing the query directly on the stored view. The stored view can be materialized on demand (when the view query is submitted) or periodically rebuilt from the base tables. For databases with a mixture of retrieval and update activity, materialization usually is not an efficient way to process a query on a view.

10.2.2 Processing Queries with View References

To process queries that reference a view, a DBMS can use either a materialization or modification strategy. **View materialization** requires the storage of view rows. The

[2] Microsoft Access 97 through 2016 and Oracle 8i through 12c all support Examples 10.6 and 10.7.

Example 10.6

Grouping Query Using a View Derived from a Grouping Query

List the average number of students by instructor name using *Enrollment_View*.

```
SELECT Instructor, AVG(NumStudents) AS AvgStdCount
 FROM Enrollment_View
 GROUP BY Instructor
```

Instructor	AvgStdCount
COLAN	6
FIBON	7
MACON	2.5
MILLS	3.5
VINCE	6

Example 10.7

Joining a Base Table with a View Derived from a Grouping Query

List the offering number, instructor, number of students, and course units using the *Enrollment_View* view and the *Course* table.

```
SELECT OfferNo, Instructor, NumStudents, CrsUnits
 FROM Enrollment_View, Course
 WHERE Enrollment_View.CourseNo = Course.CourseNo
   AND NumStudents < 5
```

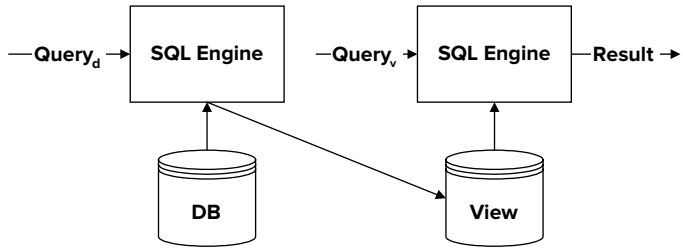
OfferNo	Instructor	NumStudents	CrsUnits
5555	MACON	2	4
6666	MILLS	2	4
7777	MACON	3	4

View Modification
a method to process a query on a view involving the execution of only one query. A query using a view is translated into a query using base tables by replacing references to the view with its definition. For databases with a mixture of retrieval and update activity, modification provides an efficient way to process a query on a view.

simplest way to store a view is to build the view from the base tables on demand (when the view query is submitted). Processing a query with a view reference requires that a DBMS execute two queries, as depicted in Figure 10.1. A user submits a query using a view ($Query_v$). The query defining the view ($Query_d$) is executed and a temporary view table is created. Figure 10.1 depicts this action by the arrow into the view. Then, the query using the view is executed using the temporary view table.

View materialization is usually not the preferred strategy because it requires a DBMS to execute two queries. However, on certain queries such as Examples 10.6 and 10.7, materialization may be necessary. In addition, materialization is preferred in data warehouses in which retrievals dominate. In a data warehouse environment, views are periodically refreshed from base tables rather than materialized on demand. Chapter 15 discusses materialized views used in data warehouses.

In an environment with a mix of update and retrieval operations, **view modification** usually provides better performance than materialization because the



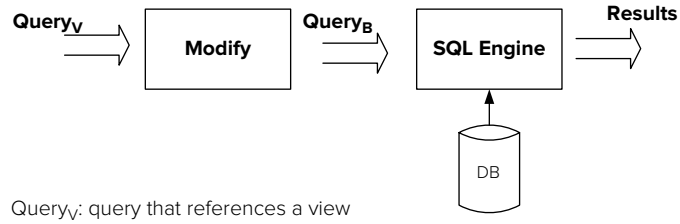
Query$_d$: Query that defines a view

Query$_v$: Query that references a view

FIGURE 10.1
Process Flow of View Materialization

DBMS only executes one query. Figure 10.2 shows that a query using a view is modified or rewritten as a query using base tables only; then the modified query is executed using the base tables. The modification process happens automatically without any user knowledge or action. In most DBMSs, the modified query cannot be seen even if you want to review it.

As a view modification example, consider the transformation shown from Example 10.8 to Example 10.9. When you submit a query using a view, the reference to the view is replaced with the definition of the view. The view name in the FROM clause is replaced by base tables. In addition, the conditions in the WHERE clause are combined using the Boolean AND operator with the conditions in the query defining the view. The underlined parts in Example 10.9 indicate substitutions made in the modification process.



Query$_V$: query that references a view

Query$_B$: modification of Query$_V$ such that references to the view are replaced by references to base tables.

FIGURE10.2
Process Flow of View Modification

Example 10.8

Query Using MS_View

```
SELECT OfferNo, CourseNo, FacFirstName, FacLastName,
       OffTime, OffDays
 FROM MS_View
 WHERE OffTerm = 'SPRING' AND OffYear = 2017
```

OfferNo	CourseNo	FacFirstName	FacLastName	OffTime	OffDays
3333	IS320	LEONARD	VINCE	8:30 AM	MW
9876	IS460	LEONARD	FIBON	1:30 PM	TTH
5679	IS480	CRISTOPHER	COLAN	3:30 PM	TTH

Example 10.9

Modification of Example 10.8

Example 10.8 is modified by replacing references to *MS_View* with base table references.

```
SELECT OfferNo, Course.CourseNo, FacFirstName,
       FacLastName, OffTime, OffDays
 FROM Faculty, Course, Offering
 WHERE FacDept = 'MS'
   AND Faculty.FacNo = Offering.FacNo
   AND Offering.CourseNo = Course.CourseNo
   AND OffTerm = 'SPRING' AND OffYear = 2017
```

Some DBMSs perform additional simplification of modified queries to remove unnecessary joins. For example, the *Course* table is not needed because there are no conditions and columns from the *Course* table in Example 10.9. In addition, the join between the *Offering* and the *Course* tables is not necessary because every *Offering* row is related to a *Course* row (null is not allowed). As a result the modified query can be simplified by removing the *Course* table. Simplification will result in a faster execution time, as the most important factor in execution time is the number of tables.

Example 10.10

Further Simplification of Example 10.9

Simplify by removing the *Course* table because it is not needed in Example 10.9.

```
SELECT OfferNo, CourseNo, FacFirstName, FacLastName,
       OffTime, OffDays
 FROM Faculty, Offering
 WHERE FacDept = 'MS'
   AND Faculty.FacNo = Offering.FacNo
   AND OffTerm = 'SPRING' AND OffYear = 2017
```

10.3 UPDATING USING VIEWS

Updatable View
a view that can be used in SELECT statements as well as UPDATE, INSERT, and DELETE statements. Views that can be used only with SELECT statements are known as read-only views. Updatable views provide a foundation for simplified specification of data requirements for data entry forms.

Depending on its definition, a view can be read-only or updatable. A read-only view can be used in SELECT statements as demonstrated in Section 10.2. All views are at least read-only. A read-only view cannot be used in queries involving INSERT, UPDATE, and DELETE statements. A view that can be used in modification statements as well as SELECT statements is known as an **updatable view**. Updatable views support data requirements for data entry forms, commonly used for database input by users. Instead of tedious coding, data requirements for data entry forms can be specified in a simplified manner using views. This section describes rules for defining both single-table and multiple-table updatable views.

10.3.1 Single-Table Updatable Views

An updatable view allows you to insert, update, or delete rows in the underlying base tables by performing the corresponding operation on the view. Whenever a modification is made to a view row, a corresponding operation is performed on the base table. Intuitively, this means that the rows of an updatable view correspond in a one-to-one manner with rows from the underlying base tables. If a view contains the primary

key of the base table, then each view row matches a base table row. A single-table view is updatable if it satisfies the following three rules that include the primary key requirement.

Rules for Single-Table Updatable Views

- The view contains the primary key of the base table.
- The view contains all required columns (NOT NULL) of the base table except required columns with default values.
- The view's query does not have the GROUP BY or DISTINCT keywords.

Following these rules, *Fac_View1* (Example 10.11) is updatable while *Fac_View2* (Example 10.12) and *Fac_View3* (Example 10.13) are read-only. *Fac_View1* is updatable assuming the missing *Faculty* columns are not required. *Fac_View2* violates Rules 1 and 2 while *Fac_View3* violates all three rules making both views read-only.

Example 10.11

Single-Table Updatable View

Create a row and column subset view with the primary key.

```
CREATE VIEW Fac_View1 AS
 SELECT FacNo, FacFirstName, FacLastName, FacRank,
        FacSalary, FacDept, FacCity, FacState, FacZipCode
  FROM Faculty
  WHERE FacDept = 'MS'
```

FacNo	FacFirstName	FacLastName	FacRank	FacSalary	FacDept	FacCity	FacState	FacZipCode
098-76-5432	LEONARD	VINCE	ASST	35000.00	MS	SEATTLE	WA	98111-9921
543-21-0987	VICTORIA	EMMANUEL	PROF	120000.00	MS	BOTHELL	WA	98011-2242
654-32-1098	LEONARD	FIBON	ASSC	70000.00	MS	SEATTLE	WA	98121-0094
876-54-3210	CRISTOPHER	COLAN	ASST	40000.00	MS	SEATTLE	WA	98114-1332

Example 10.12

Single Table Read-Only View

Create a row and column subset without the primary key.

```
CREATE VIEW Fac_View2 AS
 SELECT FacDept, FacRank, FacSalary
  FROM Faculty
  WHERE FacSalary > 50000
```

FacDept	FacRank	FacSalary
MS	PROF	120000.00
MS	ASSC	70000.00
FIN	PROF	65000.00
FIN	ASSC	75000.00

Example 10.13

Single table read-only view

Create a grouping view with faculty department and average salary.

```
CREATE View Fac_View3 (FacDept, AvgSalary) AS
 SELECT FacDept, AVG(FacSalary)
  FROM Faculty
  WHERE FacRank = 'PROF'
  GROUP BY FacDept
```

FacDept	AvgSalary
FIN	65000
MS	120000

Because *Fac_View1* is updatable, it can be used in INSERT, UPDATE, and DELETE statements to change the *Faculty* table. In Chapter 4, you used these statements to change rows in base tables. Examples 10.14 through 10.16 demonstrate that these statements can be applied to views to change rows of the underlying base tables. Note that modifications to views are subject to the integrity rules of the underlying base tables. For example, the insertion in Example 10.14 is rejected if another *Faculty* row has 999-99-8888 as the faculty number. When deleting rows in a view or changing the primary key column, the rules on referenced rows apply (Section 3.4). For example, the deletion in Example 10.16 is rejected if the *Faculty* row with *FacNo* 098-76-5432 has related rows in the *Offering* table and the delete action for the *Faculty-Offering* relationship is set to RESTRICT.

Example 10.14

Insert Operation on Updatable View

Insert a new faculty row into the MS department.

```
INSERT INTO Fac_View1
 (FacNo, FacFirstName, FacLastName, FacRank, FacSalary,
  FacDept, FacCity, FacState, FacZipCode)
 VALUES ('999-99-8888', 'JOE', 'SMITH', 'PROF', 80000,
          'MS', 'SEATTLE', 'WA', '98011-011')
```

Example 10.15

Update Operation on Updatable View

Give assistant professors in Fac_View1 a 10 percent raise.

```
UPDATE Fac_View1
 SET FacSalary = FacSalary * 1.1
 WHERE FacRank = 'ASST'
```

Example 10.16

Delete Operation on Updatable View

Delete a specific faculty member from Fac_View1.

```
DELETE FROM Fac_View1
 WHERE FacNo = '999-99-8888'
```

View Updates with Side Effects Some modifications to updatable views can be problematic, as demonstrated in Example 10.17 and Tables 10-1 and 10-2. The update statement in Example 10.17 changes the department of the second row (Victoria Emmanuel) in the view and the corresponding row in the base table. Upon regenerating the view, however, the changed row disappears (Table 10-2). The update has the side effect of causing the row to disappear from the view. This kind of side effect can occur whenever a column in the WHERE clause of a view definition is changed by an UPDATE statement. Example 10.17 updates the *FacDept* column, the column used in the WHERE clause of the definition of the *Fac_View1* view.

Example 10.17

Update Operation on Updatable View with a Side Effect

Change the department of highly paid faculty members to the finance department.

```
UPDATE Fac_View1
 SET FacDept = 'FIN'
 WHERE FacSalary > 100000
```

Because this side effect can be confusing to a user, the WITH CHECK OPTION clause can be used to prevent updates with side effects. If the WITH CHECK OPTION is specified in the CREATE VIEW statement (Example 10.18), INSERT or UPDATE statements that do not satisfy the WHERE clause are rejected. The update in Example 10.17 would be rejected if *Fac_View1* contained a WITH CHECK OPTION clause because changing *FacDept* to 'FIN' contradicts the WHERE condition with *FacDept* = 'MS'.

TABLE 10-1
Fac_View1 before Update

FacNo	FacFirstName	FacLastName	FacRank	FacSalary	FacDept	FacCity	FacState	FacZipCode
098-76-5432	LEONARD	VINCE	ASST	35000.00	MS	SEATTLE	WA	98111-9921
543-21-0987	VICTORIA	EMMANUEL	PROF	120000.00	MS	BOTHELL	WA	98011-2242
654-32-1098	LEONARD	FIBON	ASSC	70000.00	MS	SEATTLE	WA	98121-0094
876-54-3210	CRISTOPHER	COLAN	ASST	40000.00	MS	SEATTLE	WA	98114-1332

TABLE 10-2
Fac_View1 after Example 10.17 Update

FacNo	FacFirstName	FacLastName	FacRank	FacSalary	FacDept	FacCity	FacState	FacZipCode
098-76-5432	LEONARD	VINCE	ASST	35000.00	MS	SEATTLE	WA	98111-9921
654-32-1098	LEONARD	FIBON	ASSC	70000.00	MS	SEATTLE	WA	98121-0094
876-54-3210	CRISTOPHER	COLAN	ASST	40000.00	MS	SEATTLE	WA	98114-1332

Example 10.18 (Oracle)

Single-Table Updatable View Using the WITH CHECK OPTION clause

Create a row and column subset view with the primary key. The WITH CHECK OPTION clause is not supported in Access.

```
CREATE VIEW Fac_View1_Revised AS
 SELECT FacNo, FacFirstName, FacLastName, FacRank,
        FacSalary, FacDept, FacCity, FacState, FacZipCode
  FROM Faculty
  WHERE FacDept = 'MS'
 WITH CHECK OPTION
```

WITH CHECK OPTION: a clause in the CREATE VIEW statement that prevents side effects when updating a view. The WITH CHECK OPTION clause prevents UPDATE and INSERT statements that do not satisfy a view's WHERE clause.

10.3.2 Multiple-Table Updatable Views

It may be surprising but some multiple-table views are also updatable. A multiple-table view may correspond in a one-to-one manner with rows from one or more tables if the view contains the primary key of at least one table. Because multiple-table views are more complex than single-table views, there is not wide agreement on updatability rules for multiple-table views.

Some DBMSs provide no updatability support for multiple-table views. Other DBMSs support updatability for a large number of multiple-table views. In this section, the updatability rules in Microsoft Access are described as they support a wide range of multiple-table views. To demonstrate the importance of updatable views, the rules for updatable views in Access are linked to the presentation of hierarchical forms in Section 10.4.

To complement the presentation of the Access updatability rules, Appendix 10.B describes rules for updatable join views in Oracle. The rules for updatable join views in Oracle are similar to Microsoft Access although Oracle is more restrictive on allowable manipulation operations and conditions required for updatable tables.

Rules for 1-M Updatable Queries in Microsoft Access In Microsoft Access, multiple-table queries that support updates are known as 1-M updatable queries. A 1-M updatable query involves two or more tables with one table playing the role of the parent (or 1) table and another table playing the role of the child (or M) table. For example, in a query involving the *Course* and the *Offering* tables, *Course* plays the role of the parent table and *Offering* the child table. A 1-M updatable query can support manipulation operations on one table (typically the child table) or both tables. To simplify the presentation, updatability rules for the child table are presented first.

Rules for Insert Operations on 1-M Updatable Queries for a Child Table

- The query contains the primary key of the child table.
- The query contains required foreign key column(s) of the child table.
- The query contains other required columns (besides foreign keys) without default values of the child table.
- The query does not have a GROUP BY clause or the DISTINCT keyword.
- The join column of the parent table should be unique (either a primary key or a unique constraint).
- The query uses the join operator style for all join operations in the FROM clause.

A 1-M updatable query for a child table supports update operations as well as insert operations. Here are two extensions for update operations on a 1-M updatable query.

- The query supports update operations on the child table even if missing a required column (foreign key or other column besides the primary key) of the child table as long as other rules are satisfied.
- The query supports update operations on the parent table even if the query does not contain the primary key of the parent table as long as other rules are satisfied.

Using these rules, Course_Offering_View1 (Example 10.19) demonstrates a typical updatable 1-M updatable query supporting insert operations on the child table (Offering) and update operations for columns of the parent table (Course). If any required columns of the child table are missing, the query does not support inserts on the child table. Example 10.20 shows a multiple table query that does not support insert operations on

Example 10.19 (Access)

1-M Updatable Query

Create a 1-M updatable query (saved as *Course_Offering_View1*) with a join between the *Course* and the *Offering* tables. This query supports insert operations on the *Offering* table as well as update operations on *Course* columns (*CrsDesc* and *CrsUnits*).

Course_Offering_View1:

```
SELECT CrsDesc, CrsUnits,
       Offering.OfferNo, OffTerm, OffYear,
       Offering.CourseNo, OffLocation, OffTime, FacNo,
       OffDays
  FROM Course INNER JOIN Offering
    ON Course.CourseNo = Offering.CourseNo
```

Example 10.20 (Access)

Multiple-Table Query not Supporting Insert Operations

This query (saved as Course_Offering_View2) does not support insert operations on the *Offering* table because the result lacks *Offering.CourseNo*, a required foreign key. The query supports update operations on both parent (*Course*) and child (*Offering*) tables, however. You should consider Example 10.20 as a negative example, depicting an uncommon situation.

Course_Offering_View2:

```
SELECT CrsDesc, CrsUnits, Offering.OfferNo,
       OffTerm, OffYear, OffLocation,
       OffTime, FacNo, OffDays
  FROM Course INNER JOIN Offering
    ON Course.CourseNo = Offering.CourseNo
```

the Offering table because the result does not contain a required foreign key (Offering.CourseNo). In both examples, you should note usage of the join operator style (INNER JOIN keywords), required by Microsoft Access for 1-M updatable queries.

Using a 1-M updatable query to support insert operations on a parent table is not common as the presentation in Section 10.4 about hierarchical forms demonstrates. To understand 1-M updatable queries completely, however, you should know the rules for parent table operations in a 1-M updatable query.

Rules for Insert Operations on 1-M Updatable Queries for a Parent Table

- The query contains the primary key of the parent table.
- The query contains required foreign key column(s) of the parent table.
- The query contains other required columns (besides foreign keys) without default values of the parent table.
- The query does not have a GROUP BY clause or the DISTINCT keyword.
- The join column of the parent table should be unique (either a primary key or a unique constraint).
- The query uses the join operator style for all join operations in the FROM clause.

A 1-M updatable query for a parent table supports update operations as well as insert operations. Here are two extensions for update operations on a 1-M updatable query.

- The query supports update operations on the parent table even if missing a required column (foreign key or other column besides the primary key) of the parent table as long as other rules are satisfied.
- The query supports update operations on the parent table even if the query does not contain the primary key of the parent table as long as other rules are satisfied.

Example 10.21 (Access)

1-M Updatable Query for Both Parent and Child Table

Create a 1-M updatable query (saved as *Course_Offering_View3*) with a join between the *Course* and the *Offering* tables. This query supports insert operations on both the *Course* and *Offering* tables as the primary key and required columns of the parent and child tables are in the result. This query depicts an uncommon situation.

Course_Offering_View3:

```
SELECT Course.CourseNo, CrsDesc, CrsUnits,
       Offering.OfferNo, OffTerm, OffYear,
       Offering.CourseNo, OffLocation, OffTime, FacNo, OffDays
  FROM Course INNER JOIN Offering
    ON Course.CourseNo = Offering.CourseNo
```

Example 10.22 (Access)

1-M Updatable Query Supporting Insert Operations on the Parent Table Only

Create a 1-M updatable query (saved as *Course_Offering_View4*) with a join between the *Course* and the *Offering* tables. This query supports insert operations on the *Course* table but only update operations on the *Offering* tables as the primary key (*Offering.OfferNo*) is not in the result. This query depicts an uncommon situation.

Course_Offering_View4:

```
SELECT Course.CourseNo, CrsDesc, CrsUnits,
       Offering.CourseNo, OffTerm, OffYear,
       OffLocation, OffTime, FacNo,  OffDays
  FROM Course INNER JOIN Offering
    ON Course.CourseNo = Offering.CourseNo
```

Example 10.21 demonstrates a query supporting insert operations on both parent and child tables. Example 10.22 depicts a query supporting insert operations on the parent table but not on the child table. The requirement for insert operations on a parent table is not common as indicated in Section 10.4 so both examples are unusual in practice.

Example 10.23 depicts another common example of a 1-M updatable query involving the *Faculty* and *Offering* tables. Although *Faculty.OfferNo* is not required (optional foreign key), it is included in the query result to support updates on this column.

Example 10.23 (Access)

1-M Updatable Query

Create a 1-M updatable query (saved as *Faculty_Offering_View1*) with a join between the *Faculty* and the *Offering* tables. This query supports update operations on both tables (*Offering* and *Faculty*) but only insert operations on the *Offering* table.

Faculty_Offering_View1:

```
SELECT Offering.OfferNo, Offering.FacNo, CourseNo,
       OffTerm, OffYear, OffLocation, OffTime,
       OffDays, FacFirstName, FacLastName, FacDept
  FROM Faculty INNER JOIN Offering
    ON Faculty.FacNo = Offering.FacNo
```

Inserting Rows in 1-M Updatable Queries Inserting a new row in a 1-M updatable query is more involved than inserting a row in a single-table view. This complication occurs because there is a choice about the tables that support insert operations. Rows from the child table only or both the child and parent tables can be inserted as a result of a view update. To insert a row into the child table, you need only supply values for the child table as demonstrated in Example 10.24. Note that values for *Offering.CourseNo* and *Offering.FacNo* must match existing rows in the *Course* and the *Faculty* tables, respectively.

Example 10.24 (Access)

Inserting a Row into the Child Table as a Result of a View Update

Insert a new row into *Offering* as a result of using *Course_Offering_View1*.

```
INSERT INTO Course_Offering_View1
 ( Offering.OfferNo, Offering.CourseNo, OffTerm, OffYear,
   OffLocation, OffTime, FacNo, OffDays )
 VALUES ( 7799, 'IS480', 'SPRING', 2017, 'BLM201',
          #1:30PM#, '098-76-5432', 'MW' )
```

To insert a row into both tables (parent and child tables), the view must include the primary key and the required columns of the parent table. If the view includes these columns, supplying values for all columns inserts a row into both tables as demonstrated in Example 10.25. Supplying values for just the parent table inserts a row only into the parent table as demonstrated in Example 10.26. In both examples, the value for *Course.CourseNo* must not match an existing *Course* row. If the value for *Course.CourseNo* matches an existing row, the insert operation will fail with a message about inserting a duplicate row.

Example 10.25 (Access)

Inserting a Row into Both Tables as a result of a View Update

Insert a new row into *Course* and *Offering* as a result of using *Course_Offering_View3*.

```
INSERT INTO Course_Offering_View3
 ( Course.CourseNo, CrsUnits, CrsDesc, Offering.OfferNo,
   OffTerm, OffYear, OffLocation, OffTime, FacNo,
   OffDays )
 VALUES ( 'IS423', 4, 'OBJECT ORIENTED COMPUTING', 8877,
          'SPRING', 2017, 'BLM201', #3:30PM#,
          '123-45-6789', 'MW' )
```

Example 10.26 (Access)

Inserting a Row into the Parent Table as a Result of a View Update

Insert a new row into the *Course* table as a result of using the *Course_Offering_View3*.

```
INSERT INTO Course_Offering_View3
 ( Course.CourseNo, CrsUnits, CrsDesc )
 VALUES ( 'IS481', 4, 'ADVANCED DATABASE' )
```

1-M Updatable Queries with More than Two Tables Queries involving more than two tables also can be updatable. The same rules apply to 1-M updatable queries with more than two tables. However, you should apply the rules to each join in the query. For example, if a query has three tables (two joins) then apply the rules to both joins. In *Faculty_Offering_Course_View1* (Example 10.27), *Offering* is the child table in both joins. Thus, the foreign keys (*Offering.CourseNo* and *Offering.FacNo*) must be in the query result. In the *Faculty_Offering_Course_Enrollment_View1* (Example 10.28), *Enrollment* is the child table in one join and *Offering* is the child table in the other two joins. The primary key of the *Offering* table is not needed in the result unless *Offering* rows should be inserted using the view. The query in Example 10.28 supports insertions on the *Enrollment* table and updates on the other tables.

Example 10.27 (Access)

1-M Updatable Query with Three Tables

Faculty_Offering_Course_View1:

```
SELECT CrsDesc, CrsUnits, Offering.OfferNo,
       Offering.CourseNo, OffTerm, OffYear,
       OffLocation, OffTime, Offering.FacNo, OffDays,
       FacFirstName, FacLastName
     FROM ( Course INNER JOIN Offering
            ON Course.CourseNo = Offering.CourseNo )
           INNER JOIN Faculty
            ON Offering.FacNo = Faculty.FacNo
```

Example 10.28 (Access)

1-M Updatable Query with Four Tables

Faculty_Offering_Course_Enrollment_View1:

```
SELECT CrsDesc, CrsUnits, Offering.CourseNo,
       Offering.FacNo, FacFirstName, FacLastName,
       OffTerm, OffYear, OffLocation, OffTime, OffDays,
       Enrollment.OfferNo, Enrollment.StdNo,
       Enrollment.EnrGrade
     FROM ( ( Course INNER JOIN Offering
              ON Course.CourseNo = Offering.CourseNo )
            INNER JOIN Faculty
              ON Offering.FacNo = Faculty.FacNo )
            INNER JOIN Enrollment
              ON Enrollment.OfferNo = Offering.OfferNo
```

The specific rules about supported insert, update, and delete operations on 1-M updatable queries are somewhat more complex than the presentation here. The purpose here is to demonstrate that multiple-table views can be updatable and the rules can be complex. The Microsoft Access documentation provides a more detailed description of the rules.

The choices about updatable tables in a 1-M updatable query can be confusing especially when the query includes more than two tables. Typically, only the child table should support insert operations, so the considerations in Examples 10.25 and 10.26 do not apply. The choices are usually dictated by the needs of data entry forms, presented in the next section.

10.4 USING VIEWS IN HIERARCHICAL FORMS

One of the most important benefits of views is that they are the building blocks for applications. Data entry forms, a cornerstone of most database applications, support retrieval and modification of tables. Data entry forms are formatted so that they are visually appealing and easy to use. In contrast, the standard formatting of query results may not appeal to most users. This section describes the hierarchical form, a powerful kind of data entry form, and the relationships between updatable views and hierarchical forms.

10.4.1 Hierarchical Forms

A form is a document used in a business process. A form is designed to support a business task such as processing an order, registering for a class, or making an airline reservation. **Hierarchical forms**[3] support business tasks with a fixed and a variable part. The fixed part of a hierarchical form is known as the main form, while the variable (repeating) part is known as the subform. For example, a hierarchical form for course offerings (Figure 10.3) shows course data in the main form and offering data in the subform. A hierarchical form for class registration (Figure 10.4) shows registration and student data in the main form and enrollment in course offerings in the subform. The billing calculation fields below the subform are part of the main form. In each form, the subform can display multiple records while the main form shows only one record.

Hierarchical Form
a formatted window for data entry and display using a fixed (main form) and a variable (subform) part. One record is shown in the main form and multiple, related records are shown in the subform.

[3] The web version of a hierarchical form is a shopping cart with the cart page containing the variable part and the checkout page containing the fixed part.

FIGURE 10.3
Example Course Offering Form

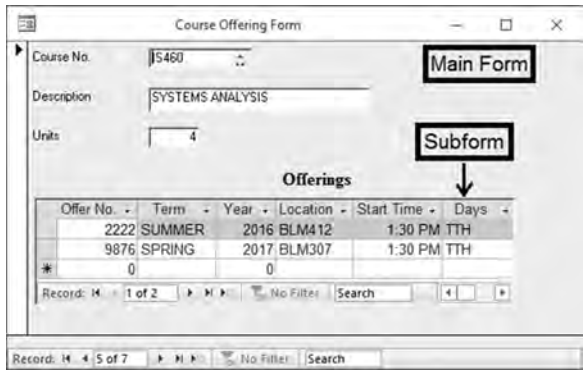
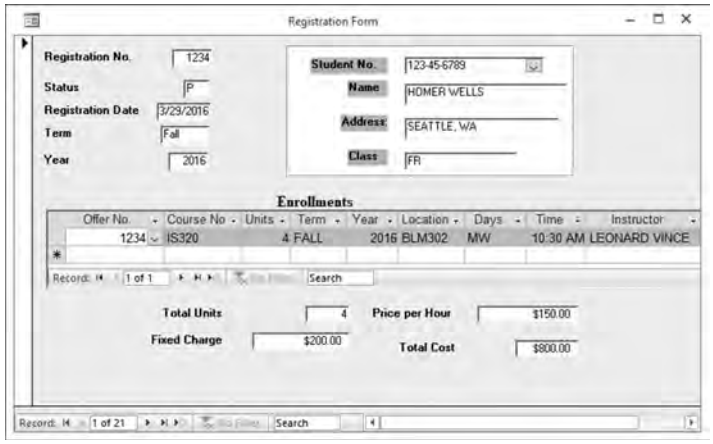


FIGURE 10.4
Example Registration Form



Hierarchical forms can be part of a system of related forms. For example, a student information system may have forms for student admissions, grade recording, course approval, course scheduling, and faculty assignments to courses. These forms may be related indirectly through updates to the database or directly by sending data between forms. For example, updates to a database made by processing a registration form are used at the end of a term by a grade recording form. This chapter emphasizes specification of data requirements for individual forms, an important skill of application development. This skill complements other application development skills such as user interface design and workflow design.

10.4.2 Relationship between Hierarchical Forms and Tables

Hierarchical forms support operations on 1-M relationships. A hierarchy or tree is a structure with 1-M relationships. Each 1-M relationship has a parent (the 1 table) and child (the M table). The 1-M relationship connects the main form to the subform. A hierarchical form allows the user to insert, update, delete, and retrieve rows in both tables of a 1-M relationship. In a typical design, a hierarchical form supports manipulation (display, insert, update, and delete) of the parent table in the main form and the child table in the subform. In essence, a hierarchical form provides a convenient interface for operations on a 1-M relationship.

As examples, let us consider the hierarchical forms shown in Figures 10.3 and 10.4. In the Course Offering Form (Figure 10.3), the relationship between the *Course* and *Offering* tables enables the form to display a *Course* row in the main form and related Offering rows in the subform. The Registration Form (Figure 10.4) operates on the *Registration* and *Enrollment* tables as well as the 1-M relationship between these tables. The *Registration* table is a new table in the university database. Figure 10.5 shows a revised relationship diagram.

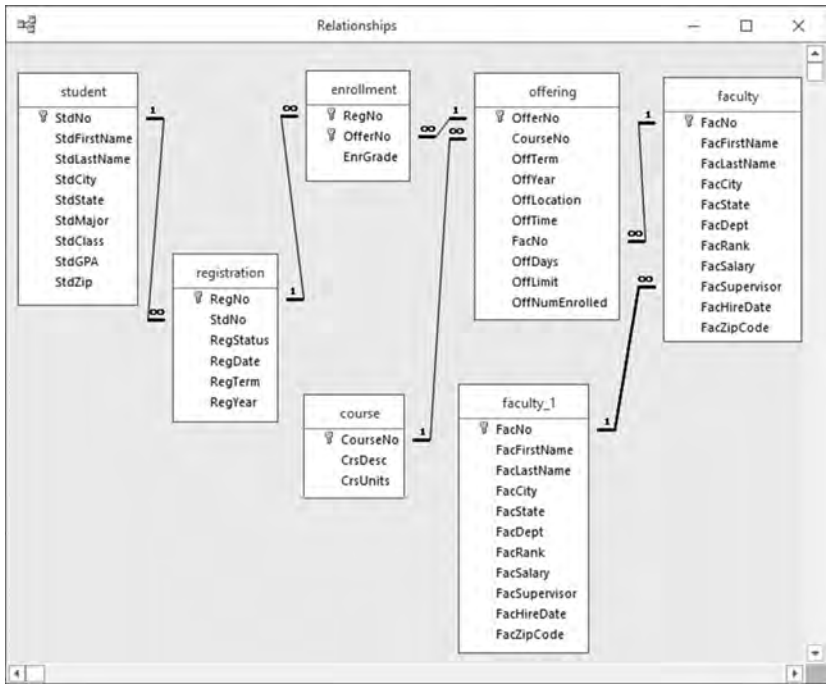


FIGURE 10.5
Relationships in the Revised University Database

To better support a business process, it is often useful to display related details in the main form and the subform. Other information (outside of the parent and the child tables) is usually for display purposes. Although it is possible to design a form to allow columns from other tables to be changed, requirements of most business processes do not support it. For example, the Registration Form (Figure 10.4) contains columns from the *Student* table in the main form so that a user can be authenticated. Likewise, columns from the *Offering*, *Faculty*, and *Course* tables are shown in the subform so that a user can make an informed enrollment choice. If a business process permits columns from other tables to be changed, this task is usually done using another form.

The Registration Form also contains summary fields (*Total Units* and *Total Cost*). Both columns are computed from data on the subform. *Total Units* is the sum of the *Units* field in the subform while *Total Cost* is calculated as the total units times the price per hour plus the fixed charge.

10.4.3 Data Requirements for Hierarchical Forms

Data requirements for hierarchical forms involve decisions for each step listed below. These steps clarify the relationship between a form and database tables referenced and changed using the form. In addition, these steps can be used directly to implement the form in some DBMSs such as Microsoft Access.

1. Identify the 1-M relationship that connects the main form to the subform.
2. Identify the join or linking columns for the 1-M relationship in step 1.
3. Identify the other tables appearing in the main form and the subform.
4. Determine the updatability of the tables in the hierarchical form.
5. Write queries for the main form and the subform.

Step 1: Identify the 1-M Relationship The most important decision is matching the form to a 1-M relationship in the database. If you are starting from a layout of a form (such as Figure 10.3 or 10.4), look for a relationship that has columns from the parent table in the main form and columns from the child table in the subform. If you are performing the form design and layout yourself, decide on the 1-M relationship before you sketch the form layout. The 1-M relationship connects the main form to the subform.

You should focus on identification of the parent table. The parent table contains the primary key of the main form. Typically, the main form contains its primary key in the top left of the form. In Figure 10.3, the Course No field is the primary key of the main form so *Course* is the parent table. In Figure 10.4, the Registration No. field is the primary key of the main form, so *Registration* is the parent table. After the parent table is identified, the child table should be easy. Typically, the table design only contains one child table related to the parent table associated with the main form. For example, *Registration* is related to only one other child table (*Enrollment*), and *Course* is related to only one other child table (*Offering*). If the parent table is related to more than one child table, choose the child table with columns in the subform.

One point of confusion involves 1-M relationships between tables represented in the same part of the form. The 1-M relationship for the form involves a parent table contained in the main form and child table contained in the subform. You should ignore 1-M relationships involving tables contained in the same part of the form. In Figure 10.4, the main form contains columns from the *Student* and *Registration* tables. You should ignore this 1-M relationship in the first two data requirement steps because this 1-M relationship involves tables in the same part of the form. You need to find a 1-M relationship that links the main form and subform.

Step 2: Identify the Linking Columns If you can identify the 1-M relationship, identifying the linking columns is usually easy. The linking columns are simply the join columns from both tables (parent and child) in the relationship. In the Course Offering Form, the linking columns are *Course.CourseNo* and *Offering.CourseNo*. In the Registration Form, the linking columns are *Registration.RegNo* and *Enrollment.RegNo*. It is important to remember that the linking columns connect the main form to the subform. With this connection, the subform only shows rows that match the linking column value of the main form. Without this connection, the subform displays all rows, not just the rows related to the record displayed in the main form.

Step 3: Determine Other Tables In addition to the 1-M relationship, other tables can be shown in the main form and the subform to provide a context to a user. If you see columns from other tables, you should note those tables so that you can use them in step 5 when writing queries for the form. For example, the Registration Form includes columns from the *Student* table in the main form. The subform includes columns from the *Offering*, *Faculty*, and *Course* tables. Computed columns, such as *Total Units* and *Total Cost*, are not a concern until the form is implemented.

Step 4: Determine Updatable Tables The fourth step requires that you understand the tables that can be changed when using the form. As part of designing a hierarchical form, you should clearly understand the requirements of the underlying business process. These requirements should be transformed into decisions about the tables affected by user actions in the form such as updating a field or inserting a new record.

Typically, there is only one table in the main form and one table in the subform that should be changed as the user enters data. In the Registration Form, the *Registration* table is changed when the user manipulates the main form and the *Enrollment* table is changed when the user manipulates the subform. Usually, other tables identified in step 3 are read-only due to business requirements. The *Student*, *Offering*, *Faculty*, and *Course* tables are read-only in the Registration subform. For some form fields that are not updatable in a hierarchical form, buttons can be used to transfer to another form to change the data. For example, a button can be added to the main form to allow a user to change student data in another form.

Sometimes the main form does not support updates to any tables. In the Course Offering Form, the *Course* table is not changed when using the main form. The reason for making the main form read-only is to support the course approval process. Most universities require a separate approval process for new courses using a separate form. The Course Offering Form is designed only for adding offerings to existing

courses and changing details of offerings of a course. If a university does not have this constraint, the main form can be used to change the *Course* table.

Step 5: Write Form Queries The last step integrates decisions made in the other steps. You should write a query for the main form and a query for the subform. These queries must support updates to the tables you identified in step 4. You should follow the rules for formulating updatable views (both single-table and multiple-table) given in Section 10.3. Some DBMSs may require that you use a CREATE VIEW statement for these queries while other DBMSs may allow you to type the SELECT statements directly.

Tables 10-3 and 10-4 summarize the responses for steps 1 to 4 for the Course Offering and Registration forms. For step 5, examples 10.29 to 10.32 show queries for the main forms and subforms of Figures 10.3 and 10.4. In Example 10.31, the *Address* form field (Figure 10.4) is derived from the *StdCity* and *StdState* columns. In Example 10.32, the primary key of the *Offering* table is not needed because the query does not support insert operations on the *Offering* table. The query only supports insert operations on the *Enrollment* table. Note that all examples conform to the Microsoft Access rules for 1-M updatable queries.

Example 10.29 (Access)

Query for the Main Form of the Course Offering Form

```
SELECT CourseNo, CrsDesc, CrsUnits FROM Course
```

Example 10.30 (Access)

Query for the Subform of the Course Offering Form

```
SELECT * FROM Offering
```

Step	Response
1	*Course* (parent table), *Offering* (child table)
2	*Course.CourseNo, Offering.CourseNo*
3	Only data from the *Course* and *Offering* tables
4	Insert, update, and delete operations on the *Offering* table in the subform

TABLE 10-3
Summary of Query Formulation Steps for the Course Offering Form

Step	Response
1	*Registration* (parent table), *Enrollment* (child table)
2	*Registration.RegNo, Enrollment. RegNo*
3	Data from the *Student* table in the main form and the *Offering*, *Course*, and *Faculty* tables in the subform
4	Insert, update, and delete operations on the *Registration* table in the main form and the *Enrollment* table in the subform

TABLE 10-4
Summary of Query Formulation Steps for the Registration Form

Example 10.31 (Access)

Query for the Main Form of the Registration Form

```
SELECT RegNo, RegTerm, RegYear, RegDate,
       Registration.StdNo, RegStatus, StdFirstName,
       StdLastName, StdClass, StdCity, StdState
 FROM Registration INNER JOIN Student
   ON Registration.StdNo = Student.StdNo
```

Example 10.32 (Access)

Query for the Subform of the Registration Form

```
SELECT RegNo, Enrollment.OfferNo, Offering.CourseNo,
       OffTime, OffLocation, OffTerm, OffYear,
       Offering.FacNo, FacFirstName, FacLastName,
       CrsDesc, CrsUnits
 FROM ( ( Enrollment INNER JOIN Offering
          ON Enrollment.OfferNo = Offering.OfferNo )
       INNER JOIN Faculty
          ON Faculty.FacNo = Offering.FacNo )
       INNER JOIN Course
          ON Course.CourseNo = Offering.CourseNo
```

Summary Field Calculation
summary fields involving aggregate functions are never calculated by the queries for the main form and subform. The calculations are specified in the form design tool, not in form queries. Thus, form queries should never have a GROUP BY clause.

You should note that neither query calculates summary fields in the main form. The Registration form contains the summary fields, *Total Units* and *Total Cost*. **Summary field calculations** are always specified in the form, not in the query. Your form queries (main form and subform) should never contain a GROUP BY clause to calculate summary fields because GROUP BY eliminates updatability. The details of summary field calculation depend on the form design tool. You should consult documentation about your DBMS and form design tool for calculating summary fields.

In the subform query for the Registration Form (Example 10.32), there is one other issue. The subform query will display an *Offering* row only if there is an associated *Faculty* row. If you want the subform to display *Offering* rows regardless of whether there is an associated *Faculty* row, a one-sided outer join should be used, as shown in Example 10.33. You can tell if an outer join is needed by looking at example copies of the form. If you can find offerings listed without an assigned faculty, then you need a one-sided outer join in the query.

As another example, Table 10-5 summarizes responses to the query formulation steps for the Faculty Assignment Form shown in Figure 10.6. The goal of this form is to support administrators in assigning faculty to course offerings. The 1-M relationship for the form is the relationship from the *Faculty* table to the *Offering* table. This form cannot be used to insert new *Faculty* rows or change data about *Faculty*. In addition, this form cannot be used to insert new *Offering* rows. The only update operation supported by this form is to change the *Faculty* assigned to teach an existing *Offering*. To update the assigned faculty in the subform, the linking column (*Offering.FacNo*) must

Example 10.33 (Access)

Revised Subform Query with a One-Sided Outer Join

```
SELECT RegNo, Enrollment.OfferNo, Offering.CourseNo,
       OffTime, OffLocation, OffTerm, OffYear,
       Offering.FacNo, FacFirstName, FacLastName,
       CrsDesc, CrsUnits
  FROM ( ( Enrollment INNER JOIN Offering
           ON Enrollment.OfferNo = Offering.OfferNo)
       INNER JOIN Course
           ON Offering.CourseNo = Course.CourseNo )
       LEFT JOIN Faculty
           ON Faculty.FacNo = Offering.FacNo
```

Step	Response
1	*Faculty* (parent table), *Offering* (child table)
2	*Faculty.FacNo, Offering.FacNo*
3	Data from the *Course* table in the subform
4	Update *Offering.FacNo*

TABLE 10-5
Summary of Query Formulation Steps for the Faculty Assignment Form

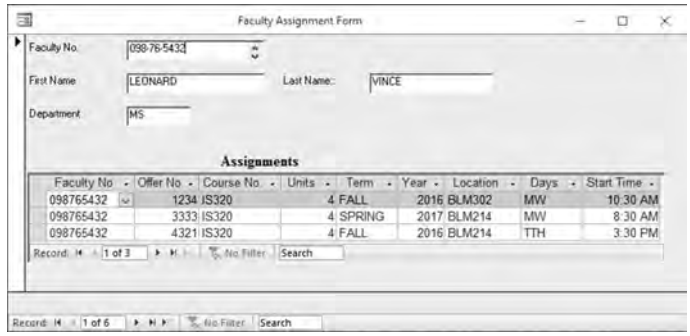


FIGURE 10.6
Example Faculty Assignment Form

Example 10.34 (Access)

Main Form Query for the Faculty Assignment Form

```
SELECT FacNo, FacFirstName, FacLastName, FacDept
 FROM Faculty
```

Example 10.35 (Access)

Subform Query for the Faculty Assignment Form

```
SELECT OfferNo, Offering.CourseNo, FacNo, OffTime,
       OffDays, OffLocation, CrsUnits
 FROM Offering INNER JOIN COURSE
      ON Offering.CourseNo = Course.CourseNo
```

appear in the subform. Examples 10.34 and 10.35 show the main form and the subform queries.

Common Query Formulation Errors Despite careful planning of data requirements, you can still make errors in query formulation especially for the initial form queries that you write. Table 10-6 provides a convenient summary of formulation errors for form queries. Note that the third error (omitting the linking column) is specific to query formulation for hierarchical forms in Microsoft Access. Students omit the linking column because it does not typically appear in the subform as shown in the form examples (Registration Form, Faculty Assignments Form, and Course Offering Form) in this section. Form design tools need the linking column to make the association between data shown on the main form and subform. Using the main form's parent table in the subform is also specific to hierarchical forms in Microsoft Access.

Examples 10.36 (revised Example 10.31) and 10.37 (revised Example 10.32) inject errors into the queries for the Registration Form. After you attempt to detect and correct errors, you can see an explanation about the errors in Appendix 10.C.

Example 10.36 (Access)

Query for the Main Form of the Registration Form

The SELECT statement contains some formulation errors covered in Table 10-6.

```
SELECT RegNo, RegTerm, RegYear, RegDate,
       Student.StdNo, RegStatus, StdFirstName,
       StdLastName, StdClass, StdCity, StdState
 FROM Registration INNER JOIN Student
   ON Registration.StdNo = Student.StdNo
 GROUP BY RegNo, RegTerm, RegYear, RegDate,
       Student.StdNo, RegStatus, StdFirstName,
       StdLastName, StdClass, StdCity, StdState
```

TABLE 10-6
Common Errors in Form Queries

Item	Description	Remedy
Foreign key	Using a primary key from the parent table instead of a foreign key from the child table	Include all required foreign keys in updatable tables in the query result
GROUP BY clause	Using a GROUP BY clause to calculate summary fields	Eliminate the GROUP BY clause and aggregate functions; specify summary fields using the form design tool
Linking column	Omitting linking column in the subform query	Include the linking column in the result of the subform query even when the linking column is not displayed in the subform
Required columns	Missing required columns in a table in which the query should support inserts	Include all required columns without default values in the query result
Duplicate parent table	Subform query contains the parent table of the main form	Remove the main form's parent table in the subform. The subform should contain the child table for the form, not both parent and child tables.

Example 10.37 (Access)

Query for the Subform of the Registration Form

The SELECT statement contains some formulation errors covered in Table 10-6.

```
SELECT Offering.OfferNo, Offering.CourseNo,
       OffTime, OffLocation, OffTerm, OffYear,
       Offering.FacNo, FacFirstName, FacLastName,
       CrsDesc, CrsUnits
 FROM ( ( ( Enrollment INNER JOIN Offering
          ON Enrollment.OfferNo = Offering.OfferNo )
      INNER JOIN Faculty
          ON Faculty.FacNo = Offering.FacNo )
      INNER JOIN Course
          ON Course.CourseNo = Offering.CourseNo )
      INNER JOIN Registration
          ON Registration.RegNo = Enrollment.RegNo
```

10.5 USING VIEWS IN REPORTS

Besides being the building blocks of data entry forms, views are also the building blocks of reports. A report is a stylized presentation of data appropriate to a selected audience. A report is similar to a form in that both use views and present the data much differently than they appear in the base tables. A report differs from a form in that a report does not change the base tables while a form can make changes to the base tables. This section describes the hierarchical report, a powerful kind of report, and the relationship between views and hierarchical reports.

10.5.1 Hierarchical Reports

Hierarchical reports (also known as control break reports) use nesting or indentation to provide a visually appealing format. The Faculty Schedule Report (Figure 10.7) shows data arranged by department, faculty name, and term. Each indented field is known as a group. The nesting of the groups indicates the sorting order of the report. The innermost line in a report is known as the detail line. In the Faculty Schedule

Hierarchical Report
a formatted display of a query using indentation to show grouping and sorting.

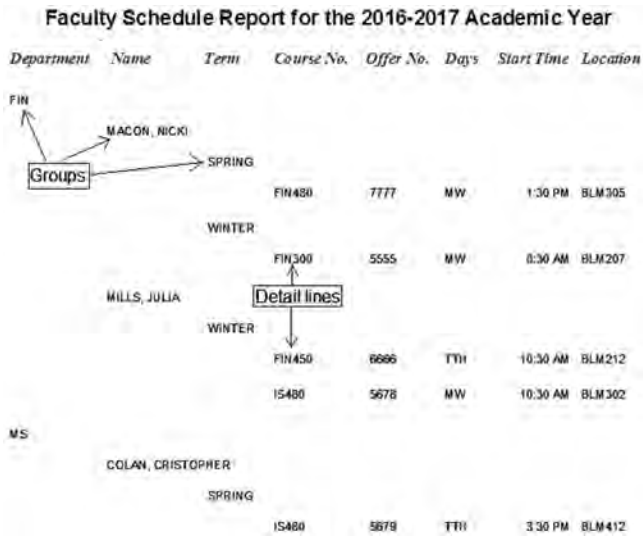


FIGURE 10.7
Faculty Schedule Report

Report, detail lines show the course number, offering number, and other details of the assigned course. The detail lines also can be sorted. In the Faculty Schedule Report, the detail lines are sorted by course number.

The major advantage of hierarchical reports is that users can more readily grasp the meaning of data that are sorted and arranged in an indented manner. The standard output of a query (a datasheet) is difficult to inspect when data from multiple tables are in the result. For example, the datasheet (Figure 10.8) shows the same values as the Faculty Schedule Report but the relationships among fields is difficult to detect. It is distracting to see the department, faculty name, and term repeated.

To improve appearance, hierarchical reports can show summary data in detail lines, computed columns, and calculations after groups. The detail lines in Figure 10.9 show the enrollment count (number of students enrolled) in each course offering taught by a professor. In SQL, the number of students is computed with the COUNT function. The columns Percent Full `((Enrollment/Limit) * 100)` and Low Enrollment (a true/false value) are computed. A check box is a visually appealing way to display true/false columns. Many reports show summary calculations after each group. In the Faculty Work Load Report, summary calculations show the total units and students as well as average percentage full of course offerings.

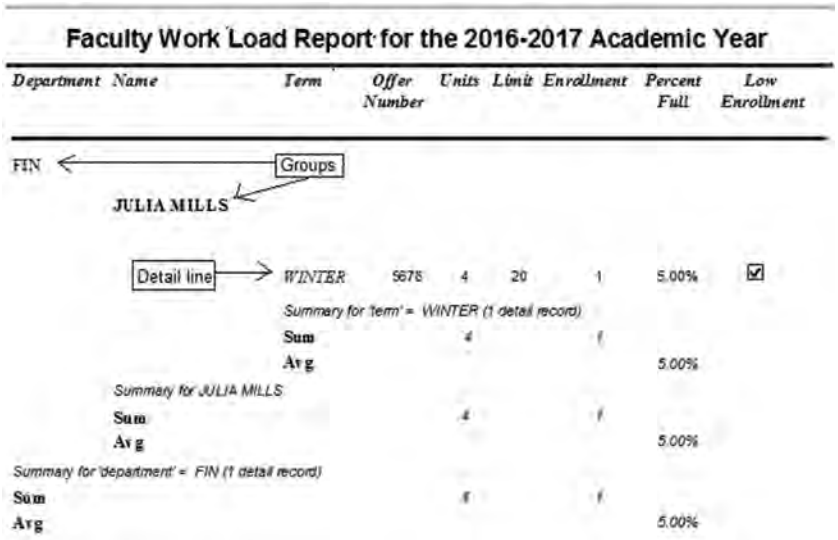
10.5.2 Data Requirements for Hierarchical Reports

Data requirements for reports are simpler than hierarchical forms because updatability requirements do not exist. Report queries are usually read-only without updatability

FIGURE 10.8
Datasheet Showing the Contents of the Faculty Schedule Report

FacDept	FacLastName	FacFirstName	OffTerm	CourseNo	OfferNo	OffLocation	OffTime	OffDays
FIN	MACON	NICKI	SPRING	FIN480	7777	BLM305	1:30 PM	MW
FIN	MACON	NICKI	WINTER	FIN300	5555	BLM207	8:30 AM	MW
FIN	MILLS	JULIA	WINTER	FIN450	6666	BLM212	10:30 AM	TTH
FIN	MILLS	JULIA	WINTER	IS480	5678	BLM302	10:30 AM	MW
MS	COLAN	CRISTOPHER	SPRING	IS480	5679	BLM412	3:30 PM	TTH
MS	EMMANUEL	VICTORIA	WINTER	IS320	4444	BLM302	3:30 PM	TTH
MS	FIBON	LEONARD	SPRING	IS460	9876	BLM307	1:30 PM	TTH
MS	VINCE	LEONARD	FALL	IS320	4321	BLM214	3:30 PM	TTH
MS	VINCE	LEONARD	FALL	IS320	1234	BLM302	10:30 AM	MW
MS	VINCE	LEONARD	SPRING	IS320	3333	BLM214	8:30 AM	MW

FIGURE 10.9
Faculty Work Load Report



support. In addition, a report only contains one query as opposed to two or more queries for a hierarchical form.

In formulating a query for a report, you should (1) match fields in the report to database columns, (2) determine necessary tables, and (3) identify the join conditions. Most report queries will involve joins and possibly one-sided outer joins. More difficult queries involving difference and division operations are not common. You can follow these steps to formulate the query, shown in Example 10.38, for the Faculty Schedule Report (Figure 10.7).

Example 10.38

Query for the Faculty Scheduling Report

```
SELECT Faculty.FacNo, Faculty.FacFirstName, FacLastName,
       Faculty.FacDept, Offering.OfferNo,
       Offering.CourseNo, Offering.OffTerm,
       Offering.OffYear, Offering.OffLocation,
       Offering.OffTime, Offering.OffDays
 FROM Faculty, Offering
 WHERE Faculty.FacNo = Offering.FacNo
   AND ( ( Offering.OffTerm = 'FALL'
   AND Offering.OffYear = 2016 )
    OR ( Offering.OffTerm = 'WINTER'
   AND Offering.OffYear = 2017 )
    OR ( Offering.OffTerm = 'SPRING'
   AND Offering.OffYear = 2017 ) )
```

The major query formulation issue for hierarchical reports is the level of the output. Sometimes you have a choice between individual rows or groups of rows in the query result. A rule of thumb is that the query should produce data for detail lines on the report. The query for the Faculty Work Load Report (Example 10.39) groups data and counts the number of students enrolled. The query directly produces data for detail lines on the report. If the query produced one row per student enrolled in a course (a finer level of detail), then the report must calculate the number of students enrolled. With most reporting tools, it is easier to perform aggregate calculations in the query when the detail line of the report shows only summary data.

Query Formulation Tip for Hierarchical Reports: the query for a report should produce data for detail lines of the report. If detail lines in a report contain summary data, the query should usually contain summary data.

The other calculations (*PercentFull* and *LowEnrollment*) in Example 10.39 can be performed in the query or report with about the same effort. Note that *OffLimit* is a new column in the *Offering* table. It shows the maximum number of students that can enroll in a course offering. The Access formulation uses two notational shortcuts not recognized by the Oracle SQL compiler. Access SQL allows the usage of a renamed column (*NumStds*) in other expressions in the SELECT list. In addition, Access SQL allows usage of a comparison operator to return a true/false value. Oracle SQL does not allow the renamed column (*NumStds*) to appear in other expressions. In addition, the Oracle formulation uses a CASE expression, a proprietary extension of Oracle SQL instead of the comparison operator. The keyword CASE begins a CASE expression.

Example 10.39 (Access)

Query for the Faculty Work Load Report with summary data in detail lines

```
SELECT Offering.OfferNo, FacFirstName, FacLastName,
       FacDept, OffTerm, CrsUnits, OffLimit,
       Count(Enrollment.RegNo) AS NumStds,
       NumStds/OffLimit AS PercentFull,
      (NumStds/OffLimit) < 0.25 AS LowEnrollment
 FROM Faculty, Offering, Course, Enrollment
 WHERE Faculty.FacNo = Offering.FacNo
   AND Course.CourseNo = Offering.CourseNo
   AND Offering.OfferNo = Enrollment.OfferNo
   AND ( ( Offering.OffTerm = 'FALL'
   AND Offering.OffYear = 2016 )
    OR ( Offering.OffTerm = 'WINTER'
   AND Offering.OffYear = 2017 )
    OR ( Offering.OffTerm = 'SPRING'
   AND Offering.OffYear = 2017 ) )
 GROUP BY Offering.OfferNo, FacFirstName, FacLastName,
          FacDept, OffTerm, CrsUnits, OffLimit
```

Example 10.39 (Oracle)

Query for the Faculty Work Load Report with summary data in detail lines

```
SELECT Offering.OfferNo, FacFirstName, FacLastName,
       FacDept, OffTerm, CrsUnits, OffLimit,
       Count(Enrollment.RegNo) AS NumStds,
       Count(Enrollment.RegNo)/OffLimit AS PercentFull,
       CASE WHEN
          Count(Enrollment.RegNo)/OffLimit < 0.25
           THEN 'T'
           ELSE 'F' END AS LowEnrollment
 FROM Faculty, Offering, Course, Enrollment
 WHERE Faculty.FacNo = Offering.FacNo
   AND Course.CourseNo = Offering.CourseNo
   AND Offering.OfferNo = Enrollment.OfferNo
   AND ( ( Offering.OffTerm = 'FALL'
   AND Offering.OffYear = 2016 )
    OR ( Offering.OffTerm = 'WINTER'
   AND Offering.OffYear = 2017 )
    OR ( Offering.OffTerm = 'SPRING'
   AND Offering.OffYear = 2017 ) )
 GROUP BY Offering.OfferNo, FacFirstName, FacLastName,
          FacDept, OffTerm, CrsUnits, OffLimit
```

CLOSING THOUGHTS

This chapter presented views, virtual tables derived from base tables with queries. The important concepts about views are the motivation for views and the usage of views in database application development. The original motivation for view usage is data independence. Changes to base table definitions usually do not affect applications that

use views. The current motivations for view usage are simplification of query formulation and flexible specification for security control. To effectively use views, you need to understand the difference between read-only and updatable views. A read-only view can be used in a query just like a base table. All views are at least read-only, but only some views are updatable. With an updatable view, changes to rows in a view are propagated to the underlying base tables. Both single-table and multiple-table views can be updatable. The most important determinant of updatability is that a view contains primary keys of the underlying base tables.

Views have become the building blocks of database applications because form and report tools use views. Data entry forms support retrieval and changes to a database. Hierarchical forms manipulate 1-M relationships in a database. To define a hierarchical form, you need to identify the 1-M relationship and define updatable views for the fixed (main form) and variable (subform) parts of the form. Hierarchical reports provide a visually appealing presentation of data. To define a hierarchical report, you need to identify grouping levels and formulate a query to produce data for the detail lines of the report.

This chapter continues Part 5 with emphasis on application development with relational databases. In Chapter 9, you extended your query formulation skills and understanding of relational databases begun in the Part 2 chapters. This chapter stressed the application of query formulation skills in building applications based on views. Chapter 11 demonstrates the usage of queries in stored procedures and triggers to customize and extend database applications. To cement your understanding of application development with views, you need to use a relational DBMS especially to build forms and reports. It is only by applying the concepts to an actual database application that you will really learn the concepts.

REVIEW CONCEPTS

- Benefits of views: data independence, simplified query formulation, security
- View definition in SQL:

```
CREATE VIEW IS_Students AS
 SELECT * FROM Student WHERE StdMajor = 'IS'
```

- Using a view in a query:

```
SELECT StdFirstName, StdLastName, StdCity, StdGPA
 FROM IS_Students
 WHERE StdGPA >= 3.7
```

- Using an updatable view in INSERT, UPDATE, and DELETE statements:

```
UPDATE IS_Students
 SET StdGPA = 3.5
 WHERE StdClass = 'SR'
```

- View modification: DBMS service to process a query on a view involving the execution of only one query. A query using a view is translated into a query using base tables by replacing references to the view with its definition.
- View materialization: DBMS service to process a query on a view by executing the query directly on the stored view. The stored view can be materialized on demand (when the view query is submitted) or periodically rebuilt from its base tables.
- Typical usage of view modification for databases that have a mix of update and retrieval operations
- Updatable view: a view that can be used in SELECT statements as well as UPDATE, INSERT, and DELETE statements.
- Rules for defining single-table updatable views: primary key and required columns

- WITH CHECK OPTION clause to prevent view updates with side effects
- Rules for defining multiple-table updatable views: primary key and required columns of each updatable table along with foreign keys of the child tables
- 1-M updatable queries for developing data entry forms in Microsoft Access
- Components of a hierarchical form: main form and subform
- Hierarchical forms providing a convenient interface for manipulating 1-M relationships and associated tables
- Data requirement steps for hierarchical forms: identify the 1-M relationship, identify the linking columns, identify other tables on the form, determine updatability of tables, write the queries for the main form and subform
- Writing updatable queries for the main form and the subform
- Form queries (main form and subform) not using the GROUP BY clause
- Common errors in updatable queries for forms: using primary key of parent table instead of foreign key of child table, GROUP BY to calculate summary fields, omitting the linking column in a subform query, missing required columns from tables with supported insert operations, and using the parent table of the main form in the subform query
- Summary calculations specified using a form design tool, not in a form query (main form or subform query)
- Hierarchical report: a formatted display of a query using indentation to show grouping and sorting
- Components of hierarchical reports: grouping fields, detail lines, and group summary calculations
- Writing queries for hierarchical reports: provide data for detail lines

QUESTIONS

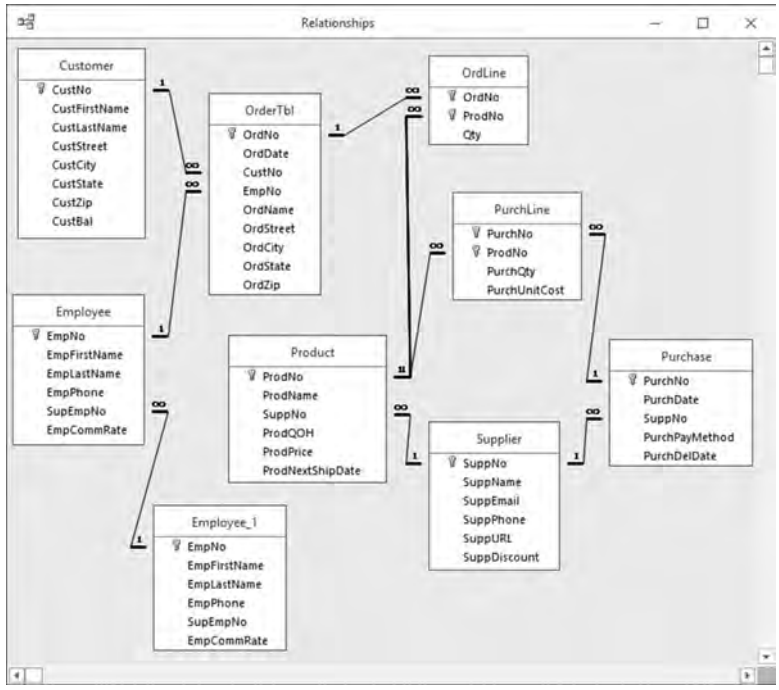
1. How do views provide data independence?
2. How can views simplify queries written by users?
3. How is a view like a macro in a spreadsheet?
4. What is view materialization?
5. What is view modification?
6. When is modification preferred to materialization for processing view queries?
7. What is an updatable view?
8. Why are some views read-only?
9. What are the rules for single-table updatable views?
10. What are the rules for a 1-M updatable query to support insert operations for a child table in Microsoft Access?
11. What is the purpose of the WITH CHECK clause?
12. What is a hierarchical form?
13. Briefly describe how a hierarchical form can be used in a business process that you know about. For example, if you know something about order processing, describe how a hierarchical form can support this process.
14. What is the difference between a main form and a subform?
15. What is the purpose of linking columns in hierarchical forms?
16. Why should you write updatable queries for a main form and a subform?
17. Why are tables used in a hierarchical form even when the tables cannot be changed as a result of using the form?

18. What is the first step of specifying data requirements for hierarchical forms?
19. What is the second step of specifying data requirements for hierarchical forms?
20. What is the third step of specifying data requirements for hierarchical forms?
21. What is the fourth step of specifying data requirements for hierarchical forms?
22. What is the fifth step of specifying data requirements for hierarchical forms?
23. Provide an example of a hierarchical form in which the main form is not updatable. Explain the business reason that determines the read-only status of the main form.
24. What is a hierarchical report?
25. What is a grouping column in a hierarchical report?
26. How do you identify grouping columns in a report?
27. What is a detail line in a hierarchical report?
28. What is the relationship of grouping columns in a report to sorting columns?
29. Why is it often easier to write a query for a hierarchical report than for a hierarchical form?
30. What does it mean that a query should produce data for the detail line of a hierarchical report?
31. Do commercial DBMSs agree on the rules for updatable multiple-table views? If no, briefly comment on the level of agreement about rules for updatable multiple-table views.
32. What side effects can occur when a user changes the row of an updatable view? What is the cause of such side effects?
33. Describe a scenario in which materialization is preferred to modification for view processing.
34. Why should it be easy to identify the child table in the 1-M relationship for a hierarchical form?
35. How do you determine the updatable tables in a hierarchical form?
36. Identify calculated summary fields on the example forms in this chapter. Are the summary calculations specified in a form design tool or in form queries?
37. Briefly explain common query formulation errors for main form and subform queries.
38. What are the rules for a 1-M updatable query to support insert operations for a parent table in Microsoft Access?
39. Identify keywords that eliminate updatability for a query.
40. What join style must be used in Microsoft Access for 1-M updatable queries?
41. Under what conditions does a 1-M updatable query support update operations on a child table?
42. Under what conditions does a 1-M updatable query support update operations on a parent table?
43. Can a 1-M updatable query support update operations on a parent table if the query result does not contain the primary key of the parent table?

PROBLEMS

The problems use the extended order entry database depicted in Figure 10.P1 and Table 10-P1. Oracle CREATE TABLE statements for the new tables and the revised *Product* table follow Table 10-P1. This database extends the order entry database used in the problems of Chapters 4 and 9 with three tables: (1) *Supplier*, containing the list of suppliers for products carried in inventory; (2) *Purchase*, recording the general details of purchases to

FIGURE 10.P1
Relationship Diagram for the Revised Order Entry Database



replenish inventory; and (3) *PurchLine*, containing the products requested on a purchase. In addition, the extended order entry database contains a new 1-M relationship (*Supplier* to *Product*) that replaces the *Product.ProdMfg* column in the original database.

In addition to the revisions noted in the previous paragraph, you should be aware of several assumptions made in the design of the Extended Order Entry Database:

- The design makes the simplifying assumption that there is only one supplier for each product. This assumption is appropriate for a single retail store that orders directly from manufacturers.
- The 1-M relationship from *Supplier* to *Purchase* supports the purchasing process. In this process, a user designates the supplier before selecting items to order from the supplier. Without this relationship, the business process and associated data entry forms would be more difficult to implement.

TABLE 10-P1
Explanations of Selected Columns in the Revised Order Entry Database

Column Name	Description
PurchDate	Date of making the purchase
PurchPayMethod	Payment method for the purchase (Credit, PO, or Cash)
PurchDelDate	Expected delivery date of the purchase
SuppDiscount	Discount provided by the supplier
PurchQty	Quantity of product purchased
PurchUnitCost	Unit cost of the product purchased

```
CREATE TABLE Product
(     ProdNo            CHAR(8),
      ProdName          VARCHAR2(50) CONSTRAINT ProdNameRequired NOT NULL,
      SuppNo            CHAR(8) CONSTRAINT SuppNo1Required NOT NULL,
      ProdQOH           INTEGER DEFAULT 0,
      ProdPrice         DECIMAL(12,2) DEFAULT 0,
      ProdNextShipDate  DATE,
```

```
 CONSTRAINT PKProduct PRIMARY KEY (ProdNo),
 CONSTRAINT SuppNoFK1 FOREIGN KEY (SuppNo) REFERENCES Supplier
    ON DELETE CASCADE )
```

```
CREATE TABLE Supplier
(     SuppNo            CHAR(8),
      SuppName          VARCHAR2(30) CONSTRAINT SuppNameRequired NOT NULL,
      SuppEmail         VARCHAR2(50),
      SuppPhone         CHAR(13),
      SuppURL           VARCHAR2(100),
      SuppDiscount      DECIMAL(3,3),
 CONSTRAINT PKSupplier PRIMARY KEY (SuppNo) )
```

```
CREATE TABLE Purchase
(     PurchNo           CHAR(8),
      PurchDate         DATE CONSTRAINT PurchDateRequired NOT NULL,
      SuppNo            CHAR(8) CONSTRAINT SuppNo2Required NOT NULL,
      PurchPayMethod    CHAR(6) DEFAULT 'PO',
      PurchDelDate      DATE,
 CONSTRAINT PKPurchase PRIMARY KEY (PurchNo) ,
 CONSTRAINT SuppNoFK2 FOREIGN KEY (SuppNo) REFERENCES Supplier )
```

```
CREATE TABLE PurchLine
(     ProdNo         CHAR(8),
      PurchNo        CHAR(8),
      PurchQty       INTEGER DEFAULT 1 CONSTRAINT PurchQtyRequired NOT NULL,
      PurchUnitCost  DECIMAL(12,2),
CONSTRAINT PKPurchLine PRIMARY KEY (PurchNo, ProdNo),
CONSTRAINT FKPurchNo FOREIGN KEY (PurchNo) REFERENCES Purchase
    ON DELETE CASCADE,
CONSTRAINT FKProdNo2 FOREIGN KEY (ProdNo) REFERENCES Product )
```

1. Define a view containing products from supplier number S3399214. Include all *Product* columns in the view.
2. Define a view containing the details of orders placed in January 2017. Include all *OrderTbl* columns, *OrdLine.Qty*, and the product name in the view.
3. Define a view containing the product number, name, price, and quantity on hand along with the number of orders in which the product appears.
4. Using the view defined in problem 1, write a query to list the products with a price greater than $300. Include all view columns in the result.
5. Using the view defined in problem 2, write a query to list the rows containing the words "Ink Jet" in the product name. Include all view columns in the result.

6. Using the view defined in problem 3, write a query to list the products in which more than five orders have been placed. Include the product name and the number of orders in the result.
7. For the query in problem 4, modify the query so that it uses base tables only.
8. For the query in problem 5, modify the query so that it uses base tables only.
9. For the query in problem 6, modify the query so that it uses base tables only.
10. Is the view in problem 1 updatable? Explain why or why not.
11. Is the view in problem 2 updatable? Explain why or why not. What database tables can be changed by modifying rows in the view?
12. Is the view in problem 3 updatable? Explain why or why not.
13. For the view in problem 1, write an INSERT statement that references the view. The effect of the INSERT statement should add a new row to the *Product* table.
14. For the view in problem 1, write an UPDATE statement that references the view. The effect of the UPDATE statement should modify the *ProdQOH* column of the row added in problem 13.
15. For the view in problem 1, write a DELETE statement that references the view. The effect of the DELETE statement should remove the row added in problem 13.
16. Modify the view definition of problem 1 to prevent side effects. Use a different name for the view than the name used in problem 1. Note that the WITH CHECK OPTION clause cannot be specified in Microsoft Access using the SQL window.
17. Write an UPDATE statement for the view in problem 1 to modify the *SuppNo* of the row with *ProdNo* of P6677900 to S4420948. The UPDATE statement should be rejected by the revised view definition in problem 16 but accepted by the original view definition in problem 1. This problem cannot be done in Access using the SQL window.
18. Define a 1-M updatable query involving the *Customer* and the *OrderTbl* tables. The query should support insert operations to the *OrderTbl* table. The query should include all columns of the *OrderTbl* table and the name (first and last), street, city, state, and zip of the *Customer* table. Note that this problem is specific to Microsoft Access.
19. Define a 1-M updatable query involving the *Customer* table, the *OrderTbl* table, and the *Employee* table. The query should support insert operations to the *OrderTbl* table. Include all rows in the *OrderTbl* table even if there is a null employee number. The query should include all columns of the *OrderTbl* table, the name (first and last), street, city, state, and zip of the *Customer* table, and the name (first and last) and phone of the *Employee* table. Note that this problem is specific to Microsoft Access.
20. Define a 1-M updatable query involving the *OrdLine* and the *Product* tables. The query should support insert operations to the *OrdLine* table. The query should include all columns of the *OrdLine* table and the name, the quantity on hand, and the price of the *Product* table. Note that this problem is specific to Microsoft Access.
21. Define a 1-M updatable query involving the *Purchase* and the *Supplier* tables. The query should support updates and inserts to the *Product* and the *Supplier* tables. Include the necessary columns so that both tables are updatable. Note that this problem is specific to Microsoft Access.
22. For the sample Simple Order Form shown in Figure 10.P2, answer the five data requirement questions discussed in Section 10.4.3. The form supports manipulation of the heading and the details of orders.
23. For the sample Order Form shown in Figure 10.P3, answer the five data requirement questions discussed in Section 10.4.3. Like the Simple Order

Form depicted in Figure 10.P2, the Order Form supports manipulation of the heading and the details of orders. In addition, the Order Form displays data from other tables to provide a context for the user when completing an order. The Order Form supports both phone (an employee taking the order) and Web (without an employee taking the order) orders. The subform query should compute the Amount field as `Qty*ProdPrice`. Do not compute the Total Amount field in either the main form query or the subform query. It is computed in the form.

24. Modify your answer to problem 23 assuming that the Order Form supports only phone orders, not Web orders.
25. For the sample Simple Purchase Form shown in Figure 10.P4, answer the five data requirement questions discussed in Section 10.4.3. The form supports manipulation of the heading and the details of purchases.
26. For the sample Purchase Form shown in Figure 10.P5, answer the five data requirement questions presented in Section 10.4.3. Like the Simple Purchase Form depicted in Figure 10.P4, the Purchase Form supports manipulation of the heading and the details of purchases. In addition, the Purchase Form displays data from other tables to provide a context for the user when completing a purchase. The subform query should compute the Amount field as `PurchQty*PurchUnitCost`. The Amount field is to the right of the Unit Cost Field in the subform. Do not compute the Total Amount field in either the main form query or the subform query. It is computed in the form.

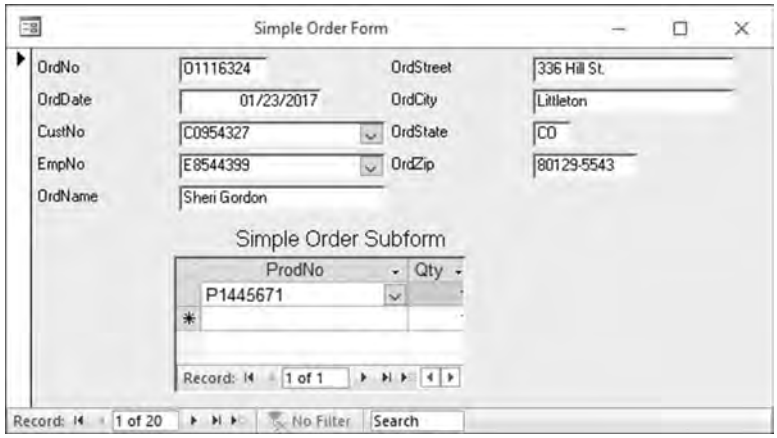


FIGURE 10.P2
Simple Order Form

FIGURE 10.P3
Order Form

FIGURE 10.P4
Simple Purchase Form

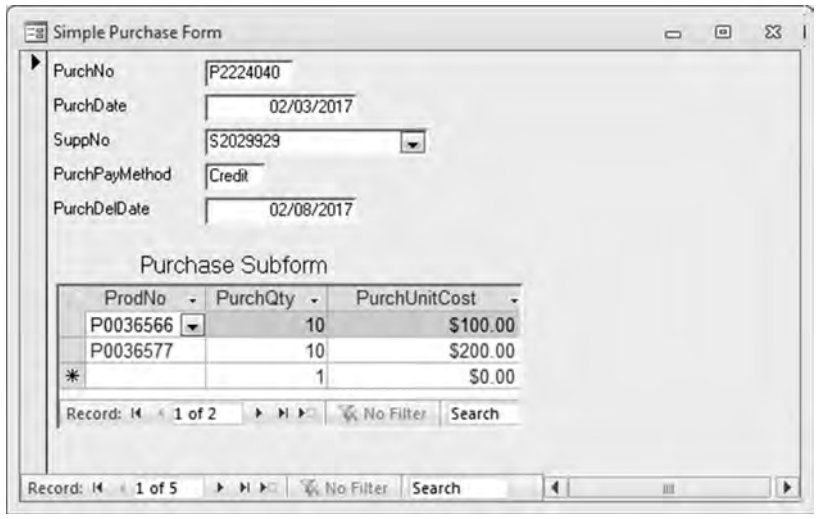
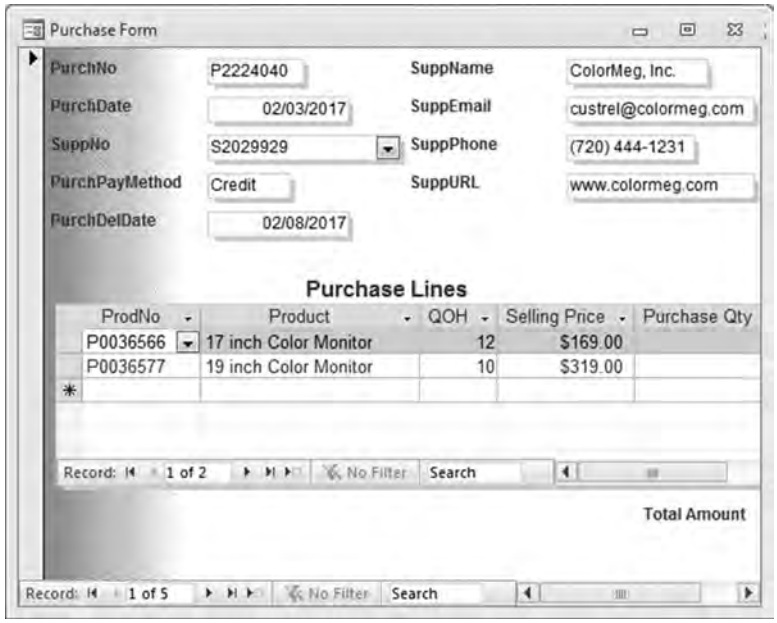
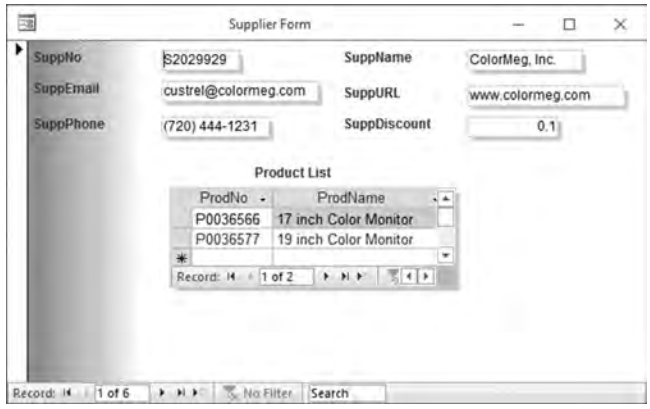


FIGURE 10.P5
Purchase Form



27. For the sample Supplier Form shown in Figure 10.P6, answer the five data requirement questions presented in Section 10.4.3. The main form supports the manipulation of supplier data while the subform supports the manipulation of only the product number and the product name of the products provided by the supplier in the main form.

FIGURE 10.P6
Supplier Form



28. For the Order Detail Report, write a SELECT statement to produce the data for the detail lines. The grouping column in the report is *OrdNo*. The report should list the orders for customer number O2233457 in January 2017.

Order Detail Report

Order Number	Order Date	Product No	Qty	Price	Amount
O2233457	1/12/2017	P1441567	1	$14.99	$14.99
		P0036577	2	$319.00	$638.00
Total Order Amount					$652.99
O4714645	1/11/2017	P9995676	2	$89.00	$178.00
		P0036566	1	$369.00	$369.00
Total Order Amount					$547.00

29. For the sample Order Summary Report, write a SELECT statement to produce the data for the detail lines. The Zip Code report field is the first five characters of the *CustZip* column. The grouping field in the report is the first five characters of the *CustZip* column. The Order Amount Sum report field is the sum of the quantity times the product price. Limit the report to year 2017 orders. You should also include the month number in the SELECT statement so that the report can be sorted by the month number instead of the Month report field. Use the following expressions to derive computed columns used in the report:

    - In Microsoft Access, the expression `left(CustZip, 5)` generates the Zip Code report field. In Oracle, the expression `substr(CustZip, 1, 5)` generates the Zip Code report field.
    - In Microsoft Access, the expression `format(OrdDate, "mmmm yyyy")` generates the Year report field. In Oracle, the expression `to_char(OrdDate, 'MONTH YYYY')` generates the Year report field.
    - In Microsoft Access, the expression `month(OrdDate)` generates the Month report field. In Oracle, the expression `to_number(to_char(OrdDate, 'MM'))`generates the Month report field.

Order Summary Report

Zip Code	Month	Order Line Count	Order Amount Sum
80111	January 2017	10	$1,149
	February 2017	21	$2,050
Summary of 80111		31	$3,199
80113	January 2017	15	$1,541
	February 2017	11	$1,450
Summary of 80113		31	$2,191

30. Revise the Order Summary Report to list the number of orders and the average order amount instead of the Order Line Count and Order Amount Sum. The revised report appears below. You will need to use a SELECT statement in the FROM clause or write two statements to produce the data for the detail lines.

Order Summary Report

Zip Code	Month	Order Count	Average Order Amount
80111	January 2017	5	$287.25
	February 2017	10	$205.00
Summary of 80111		15	$213.27
80113	January 2017	5	$308.20
	February 2017	4	$362.50
Summary of 80113		9	$243.44

31. For the Purchase Detail Report, write a SELECT statement to produce the data for the detail lines. The grouping column in the report is *PurchNo*. The report should list the orders for supplier number S5095332 in February 2017.

Purchase Detail Report

Purch Number	Purch Date	Product No	Qty	Cost	Amount
P2345877	2/11/2017	P1441567	1	$11.99	$11.99
		P0036577	2	$229.00	$458.00
Total Purchase Amount					$469.99
P4714645	2/10/2017	P9995676	2	$69.00	$138.00
		P0036566	1	$309.00	$309.00
Total Purchase Amount					$447.00

32. For the sample Purchase Summary Report, write a SELECT statement to produce the data for the detail lines. The Area Code report field is the second through fourth characters of the *SuppPhone* column. The grouping field in the report is the second through fourth characters of the *SuppPhone* column. The Purchase Amount Sum report field is the sum of the quantity times the product price. Limit the report to year 2017 orders. You should also include the month number in the SELECT statement so that the report can be sorted by the month number instead of the Month report field. Use the following expressions to derive computed columns used in the report:
    - In Microsoft Access, the expression `mid(SuppPhone, 2, 3)` generates the Area Code report field. In Oracle, the expression `substr(SuppPhone, 2, 3)` generates the Area Code report field.
    - In Microsoft Access, the expression `format(PurchDate, "mmmm yyyy")` generates the Year report field. In Oracle, the expression `to_char(PurchDate, 'MONTH YYYY')` generates the Year report field.
    - In Microsoft Access, the expression `month(PurchDate)` generates the Month report field. In Oracle, the expression `to_number(to_char(PurchDate, 'MM'))`generates the Month report field.

Purchase Summary Report

Area Code	Month	Purch Line Count	Purch Amount Sum
303	January 2017	20	$1,149
	February 2017	11	$2,050
Summary of 303		31	$3,199
720	January 2017	19	$1,541
	February 2017	11	$1,450
Summary of 720		30	$2,191

33. Revise the Purchase Summary Report to list the number of purchases and the average purchase amount instead of the Purchase Line Count and Purchase Amount Sum. The revised report appears below. You will need to use a SELECT statement in the FROM clause or write two statements to produce the data for the detail lines.

Purchase Summary Report

Area Code	Month	Purchase Count	Average Purchase Amount
303	January 2017	8	$300.00
	February 2017	12	$506.50
Summary of 303		20	$403.25
720	January 2017	6	$308.20
	February 2017	3	$362.50
Summary of 720		9	$243.44

34. Define a view containing purchases from supplier names Connex or Cybercx. Include all *Purchase* columns in the view.
35. Define a view containing the details of purchases placed in February 2017. Include all *Purchase* columns, *PurchLine.PurchQty*, *PurchLine.PurchUnitCost*, and the product name in the view.
36. Define a view containing the product number, name, price, and quantity on hand along with the sum of the quantity purchased and the sum of the purchase cost (unit cost times quantity purchased).
37. Using the view defined in problem 34, write a query to list the purchases made with payment method PO. Include all view columns in the result.
38. Using the view defined in problem 35, write a query to list the rows containing the words Printer in the product name. Include all view columns in the result.
39. Using the view defined in problem 36, write a query to list the products in which the total purchase cost is greater than $1,000. Include the product name and the total purchase cost in the result.
40. For the query in problem 37, modify the query so that it uses base tables only.
41. For the query in problem 38, modify the query so that it uses base tables only.
42. For the query in problem 39, modify the query so that it uses base tables only.
43. Write a CREATE VIEW statement containing a join of the *Customer* and *OrderTbl* tables. The view should include all *Customer* columns and all *OrderTbl* columns except *OrderTbl.CustNo*. The view should only contain customers with a balance greater than $200.
44. Is the view in problem (43) an updatable join view in Oracle? If yes, identify the key preserving table. You should consult Appendix 10.B and the Oracle Database Administrators Guide for details about updatable join views and key preserving tables.
45. Indicate the result of the following manipulation statements using the view in problem (43).
    - Insert statement using *Customer* columns.
    - Update statement increasing the balance of all Seattle customers by $100.
    - Insert statement using the *OrderTbl* columns.
    - Update statement changing the date of an order with a specified order number.
    - Delete statement to remove all view rows associated with a particular customer number.
46. Using the view defined in problem 35, write a query to list the purchase orders in February 2017 in which the total purchase cost is greater than $100. Include the purchase date and the total purchase cost in the result.
47. For the query in problem 46, modify the query so that it uses base tables only. Use the view modification process without additional simplification.
48. For the query in problem 47, modify the query to remove unnecessary joins.

REFERENCES FOR FURTHER STUDY

Date (2003) provides additional details of view updatability issues especially related to multiple-table views. Melton and Simon (2001) describe updatable query specifications in SQL:1999. For product-specific SQL advice, For product-specific SQL advice, the sqlblog.com site features forums about a number of DBMSs including Microsoft SQL Server and open source products. The Database Journal (www.databasejournal.com) provides articles, tutorials, and resources about many DBMS products. Oracle documentation can be found at the Oracle Technet site (www.oracle.com/technetwork).

chapter

11

Stored Procedures and Triggers

Learning Objectives

This chapter explains the motivation and design issues for stored procedures and triggers and provides practice writing them using PL/SQL, the database programming language of Oracle. After this chapter, the student should have acquired the following knowledge and skills:

- Explain the reasons for writing stored procedures and triggers
- Understand the design issues of language style, binding, database connection, and result processing for database programming languages
- Write PL/SQL procedures
- Understand the classification of triggers
- Write PL/SQL triggers
- Understand trigger execution procedures

OVERVIEW

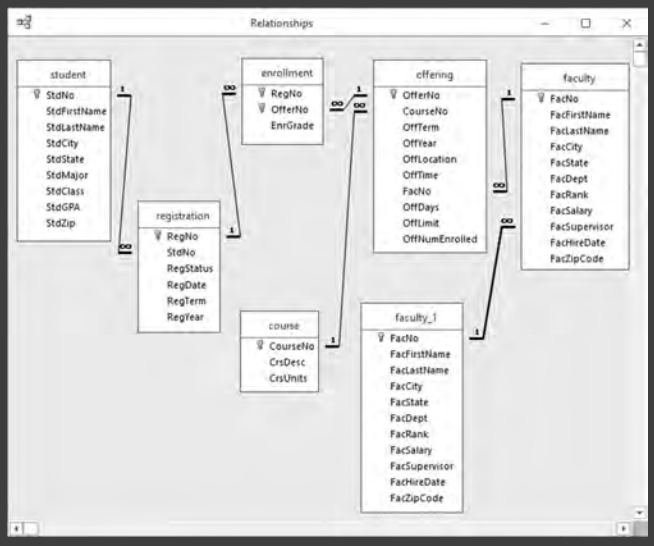
Chapter 10 provided details about application development with views. You learned about defining user views, updating base tables with views, and using views in forms and reports. This chapter augments your database application development skills with stored procedures and triggers. Stored procedures provide reuse of common code, while triggers provide rule processing for common tasks. Together, stored procedures and triggers support customization of database applications and improved productivity in developing database applications.

To become skilled in database application development as well as in database administration, you need to understand stored procedures and triggers. Since both stored procedures and triggers are coded in a database programming language, this chapter first provides background about the motivation and design issues for database programming languages as well as specific details about PL/SQL, the proprietary database programming language of Oracle.

After the background about database programming languages and PL/SQL, this chapter then presents stored procedures and triggers. For stored procedures, you will learn about the motivation and coding practices for simple and more advanced procedures. For triggers, you will learn about the classification of triggers, trigger execution procedures, and coding practices for triggers.

The presentation of PL/SQL in Sections 11.1 to 11.3 assumes that you have had a previous course in computer programming using a business programming language such as Visual Basic, Java, or JavaScript. If you would like a broader treatment of the material without computer programming details, you should read Sections 11.1.1, 11.1.2, 11.3.1, and the introductory material

FIGURE 11.1
Relationship Window for the Revised University Database



in Section 11.2 before the beginning of Section 11.2.1. In addition, the trigger examples in Section 11.3.2 mostly involve SQL statements so that you can understand the trigger examples without detailed knowledge of programming statements in PL/SQL.

For continuity, all examples about stored procedures and triggers use the revised university database of Chapter 10. Figure 11.1 shows the Access relationship window of the revised university database for convenient reference.

11.1 DATABASE PROGRAMMING LANGUAGES AND PL/SQL

After learning about the power of nonprocedural access and application development tools, you might think that procedural languages are not necessary for database application development. However, these tools, despite their power, are not complete solutions for commercial database application development. This section presents the motivation for database programming languages, design issues for database programming languages, and details about PL/SQL, the database programming language of Oracle.

11.1.1 Motivation for Database Programming Languages

Database Programming Language
a procedural language with an interface to one or more DBMSs. The interface allows a program to combine procedural statements with nonprocedural database access.

A **database programming language** is a procedural language with an interface to one or more DBMSs. The interface allows a program to combine procedural statements with database access, usually nonprocedural database access. This subsection discusses three primary motivations (customization, batch processing, and data intensive web applications) for using a database programming language and two secondary motivations (efficiency and portability).

Customization Most database application development tools support customization. Customization is necessary because no tool provides a complete solution for the development of complex database applications. Customization allows an organization to use the built-in power of a tool along with customized code to change the tool's default actions and to add new actions beyond those supported by the tool.

To support customized code, most database application development tools use event-driven coding. In this coding style, an event triggers execution of a procedure. An event model includes events for user actions such as clicking a button, as well as internal events such as before a database record is updated. An event procedure

may access the values of controls on forms and reports as well as retrieve and update database rows. Event procedures are coded using a database programming language, often a proprietary language provided by a DBMS vendor. For commercial application development, event coding is common.

Batch Processing Despite the growth of online database processing, batch processing continues to be an important way to process database work. For example, check processing typically is a batch process in which a clearinghouse bank processes large groups or batches of checks during nonpeak hours. Batch processing usually involves a delay from the occurrence of an event to its capture in a database. In the check processing case, checks are presented for payment to a merchant but not processed by a clearinghouse bank until later. Some batch processing applications such as billing statement preparation involve a cutoff time, not a time delay. Batch processing in situations involving time delays and time cutoffs can provide significant economies of scale to offset the drawback of less timely data. Even with the continued growth of commercial web commerce, batch processing remains an important method of processing database work.

Application development for batch processing involves writing computer programs in a database programming language. Since few development tools support batch processing, coding can be detailed and labor intensive. A programmer typically must write code to read input data, perform database manipulation, and create output records to show processing results.

Data Intensive Web Applications The prominence of data intensive web applications has invigorated the need for nonprocedural database access inside procedural language code. Shopping carts for electronic commerce are the most common data intensive web application. As part of the shopping experience, consumers place goods in an electronic shopping cart and then checkout to complete the purchase. Data intensive web applications in other areas have become widely used including portfolio management with online trading, health insurance exchanges, online banking, and digital mapping services.

In some ways, web development for data intensive web applications is back to the future. In the early years of personal computers and client-server processing, markup languages for page layout dominated. The later development of graphical user interfaces for personal computers replaced markup languages with office software suites and application development tools for DBMS products reduced the need for tedious coding in form and report development.

Web development has changed software development needs sharply, however. Web development is dominated by markup languages such as the Hypertext Markup Language (HTML), eXtensible Markup Language (XML), and Cascading Style Sheets (CSS). For data intensive web applications, markup language code is typically created inside code pages using procedural languages such as JavaScript. Database access is necessary to populate controls in data intensive web pages for shopping cart and other consumer web applications. Application development tools have achieved only limited usage for web development in contrast to extensive usage for non-web applications. Thus, nonprocedural database access inside procedural language code is necessary to develop data intensive web applications for both traditional and mobile users.

Other Motivations Efficiency and portability are two additional reasons for using a database programming language. When distrust in optimizing database compilers was high (until the mid-1990s), efficiency was a primary motivation for using a database programming language. To avoid inefficiencies of optimizing compilers, some DBMS vendors supported record-at-a-time access with a programmer determining the access plan for complex queries. As confidence has grown in optimizing database compilers, efficiency has become less important. However, demands of high-volume, mission-critical web applications such as online shopping and financial trading have

made custom database coding necessary to achieve acceptable performance in these applications.

Portability can be important in some environments. Most application development tools and database programming languages are proprietary. If an organization wants to remain vendor neutral, an application can be built using a nonproprietary programming language (such as Java) along with a standard database interface. If just DBMS neutrality is desired (not neutrality from an application development tool), some application development tools allow connection with a variety of DBMSs through standard database interfaces such as the Open Database Connectivity (ODBC), the Java Database Connectivity (JDBC), and the Entity Framework. Portability is a particular concern for web database access in which an application must be compatible with many types of servers and browsers.

11.1.2 Design Issues

Before undertaking the study of any database programming language, you should understand design issues about integrating a procedural language with a nonprocedural language. Understanding these issues will help you differentiate among the many languages in the marketplace and understand features of a specific language. Most DBMSs provide several alternatives for database programming languages. This section discusses the design issues of language style, binding, database connection, and result processing with an emphasis on the design choices first specified in SQL:1999 and refined through SQL:2016. Many DBMS vendors are adapting to the specifications in SQL:2016.

Statement-Level Interface
a language style for integrating a programming language with a nonprocedural language such as SQL. A statement-level interface involves changes to the syntax of a host programming language to accommodate embedded SQL statements.

Language Style SQL:2016 provides two language styles for integrating a procedural language with SQL. A **statement-level interface** involves changes to the syntax of a host programming language to accommodate embedded SQL statements. The host language contains additional statements to establish database connections, execute SQL statements, use the results of an SQL statement, associate programming variables with database columns, handle exceptions in SQL statements, and manipulate database descriptors. Statement-level interfaces are available for standard and proprietary languages. For standard languages such as Java and Visual Basic, some DBMSs provide a precompiler to process the statements before invoking the programming language compiler. Most DBMSs also provide proprietary languages such as the Oracle language PL/SQL with a statement-level interface to support embedded SQL.

The SQL:2016 specification defines the Persistent Stored Modules (SQL/PSM) language as a database programming language. Because SQL/PSM was defined after many DBMS vendors already had widely used proprietary languages, most DBMS vendors do not conform to the SQL/PSM standard. However, the SQL/PSM standard has influenced the evolution of proprietary database programming languages such as Oracle PL/SQL.

Call-Level Interface (CLI)
a language style for integrating a programming language with a nonprocedural language such as SQL. A CLI includes a set of procedures and a set of type definitions for manipulating the results of SQL statements in computer programs.

The second language style provided by SQL:2016 is known as a **call-level interface (CLI)**. The SQL:2016 CLI contains a set of procedures and a set of type definitions for SQL data types. The procedures provide similar functionality to the additional statements in a statement-level interface. The SQL:2016 CLI is more difficult to learn and use than a statement-level interface. However, the SQL:2016 CLI is portable across host languages, whereas the statement-level interface is not portable and not supported for all programming languages.

The most widely used call-level interfaces are the Open Database Connectivity (ODBC) supported by Microsoft and the Java Database Connectivity (JDBC) supported by Oracle. Because both Microsoft and Oracle have cooperated with the SQL standards efforts, the most recent versions of these proprietary CLIs are compatible to the SQL:2016 CLI. Because of the established user bases, these interfaces probably will continue to be more widely used than the SQL:2016 CLI.

Binding Binding for a database programming language involves the association of an SQL statement with its access plan. Recall from Chapter 8 that an SQL compiler determines the best access plan for an SQL statement after a detailed search of possible access plans. Static binding involves the determination of the access plan at compile time. Because the optimization process can consume considerable computing resources, it is desirable to determine the access plan at compile time and then reuse the access plan for repetitively executed statements. However, in some applications, data to retrieve cannot be predetermined. These situations require dynamic binding in which the access plan for a statement is determined when the statement is executed during run-time of the application. Even in these dynamic situations, it is useful to reuse the access plan for a statement if the statement is repetitively executed by the application.

SQL:2016 specifies both static and dynamic binding to support a range of database applications. A statement-level interface can support both static and dynamic binding. Embedded SQL statements have static binding. Dynamic SQL statements are supported by the SQL:2016 EXECUTE statement that contains an SQL statement as an input parameter. If a dynamic statement is repetitively executed by an application, the SQL:2016 PREPARE statement supports reuse of the access plan. The SQL:2016 CLI supports only dynamic binding. If a dynamic statement is repetitively executed by an application, the SQL:2016 CLI provides the Prepare() procedure to reuse the access plan.

Database Connection A database connection identifies the database used by an application. A database connection can be implicit or explicit. For procedures and triggers stored in a database, the connection is implicit. The SQL statements in triggers and procedures implicitly access the database that contains the triggers and procedures unless a different connection is specified.

In programs external to a database, the connection is explicit. SQL:2016 contains the CONNECT statement and other related statements for statement-level interfaces and the Connect() procedure and related procedures in the CLI. A database is identified by a web address or a database identifier that contains a web address. Using a database identifier relieves a database programmer from knowing the specific web address for a database as well as providing the server administrator more flexibility to relocate a database to a different location on a server.

Result Processing To process the results of SQL statements, database programming languages must resolve differences in data types and processing orientation. The data types in a programming language may not correspond exactly to the standard SQL data types. To resolve this mismatch, the database interface provides statements or procedures to map between the programming language data types and the SQL data types.

The result of a SELECT statement can be one row or a collection of rows. For SELECT statements that return at most one row (for example, retrieval by primary key), the SQL:2016 specification allows the result values to be stored in program variables. In the statement-level interface, SQL:2016 provides the USING clause to store result values in program variables. The SQL:2016 CLI provides for implicit storage of result values using predefined descriptor records that can be accessed in a program.

For SELECT statements that return more than one row, a **cursor** must be used. A cursor allows storage and iteration of a set of records returned by a SELECT statement. A cursor is similar to a dynamic array in which the array size is determined by the size of the query result. For statement-level interfaces, SQL:2016 provides statements to declare cursors, open and close cursors, position cursors, and retrieve values from cursors. The SQL:2016 CLI provides procedures with similar functionality to the statement-level interface. Section 11.2.3 presents details about cursors for PL/SQL.

Cursor
a construct in a database programming language that allows storage and iteration of a set of records returned by a SELECT statement. A cursor is similar to a dynamic array in which the array size is determined by the size of the query result.

11.1.3 PL/SQL Statements

Programming Language/Structured Query Language (PL/SQL) is a proprietary database programming language for the Oracle DBMS. Since its introduction in 1992, Oracle has steadily added features to PL/SQL so that it has the features of a modern programming language as well as a statement-level interface for SQL. Because PL/SQL is a widely used language among Oracle developers and Oracle is a widely used enterprise DBMS, this chapter uses PL/SQL to depict stored procedures and triggers.

To prepare you to read and code stored procedures and triggers, this section presents examples of PL/SQL statements. After reading this section, you should understand the structure of PL/SQL statements and be able to write PL/SQL statements using the example statements as guidelines. This section shows enough PL/SQL statement examples to allow you to read and write stored procedures and triggers of modest complexity after you complete the chapter. However, this section depicts neither all PL/SQL statements nor all statement variations.

This section is not a tutorial about computer programming. To follow the remainder of this chapter, you should have taken a previous course in computer programming or have equivalent experience. You will find that PL/SQL statements are similar to statements in other modern programming languages such as Java and Visual Basic.

Basics of PL/SQL PL/SQL statements contain reserved words and symbols, user identifiers, and constant values. Reserved words in PL/SQL are not case sensitive. Reserved symbols include the semicolon (;) for terminating statements as well as operators such as + and –. User identifiers provide names for variables, constants, and other PL/SQL constructs. User identifiers like reserved words are not case sensitive. The following list defines restrictions on user identifiers:

- Must have a maximum of 30 characters.
- Must begin with a letter.
- Allowable characters are letters (upper- and lower-case), numbers, the dollar sign, the pound symbol (#), and the underscore.
- Must not be identical to any reserved word or symbol.
- Must not be identical to other identifiers, table names, or column names.

A PL/SQL statement may contain constant values for numbers and character strings along with certain reserved words. The following list provides background about PL/SQL constants:

- Numeric constants can be whole numbers (100), numbers with a decimal point (1.67), negative numbers (–150.15), and numbers in scientific notation (3.14E7).
- String constants are surrounded in single quotation marks such as ‘this is a string’. Do not use single quotation marks to surround numeric or Boolean constants. String constants are case sensitive so that ‘This is a string’ is a different value than ‘this is a string’. To use a single quotation mark in a string constant, you should use two single quotation marks as ‘today’’s date’.
- Boolean constants are the TRUE and FALSE reserved words.
- The reserved word NULL can be used as a number, string, or Boolean constant. For strings, two single quotation marks ‘’ without anything inside denote the NULL value.
- PL/SQL does not provide date constants. You should use the **To_Date** function to convert a string constant to a date value.

Variable Declaration and Assignment Statements A variable declaration contains a variable name (a user identifier), a data type, and an optional default value. Table 11-1 lists common PL/SQL data types. Besides using the predefined types, a variable’s type can be a user defined-type created with a TYPE statement. A default value can be indicated with the DEFAULT keyword or the assignment (:=) symbol. The DECLARE keyword should precede the first variable declaration as shown in Example 11.1.

Category	Data Types	Comments
String	CHAR(L), VARCHAR2(L)	CHAR for fixed length strings, VARCHAR2 for variable length strings; L for the maximum length
Numeric	INTEGER, SMALLINT, POSITIVE, NUMBER(W,D), DECIMAL(W,D), FLOAT, REAL	W for the width; D for the number of digits to the right of the decimal point
Logical	BOOLEAN	TRUE, FALSE values
Date	DATE	Stores both date and time information including the century, the year, the month, the day, the hour, the minute, and the second. A date occupies 7 bytes.

TABLE 11-1
Summary of Common PL/SQL Data Types

Example 11.1

PL/SQL Variable Declarations

Lines beginning with double hyphens are comments.

```
DECLARE
 aFixedLengthString         CHAR(6) DEFAULT 'ABCDEF';
 aVariableLengthString      VARCHAR2(30);
 anIntegerVariable          INTEGER := 0;
 aFixedPrecisionVariable    DECIMAL(10,2);
 -- Uses the SysDate function for the default value
 aDateVariable              DATE DEFAULT SysDate;
```

For variables associated with columns of a database table, PL/SQL provides anchored declarations. Anchored declarations relieve the programmer from knowing the data types of database columns. An anchored declaration includes a fully-qualified column name followed by the keyword %TYPE. Example 11.2 demonstrates anchored variable declarations using columns from the revised university database of Chapter 10. The last anchored declaration involves a variable using the type associated with a previously defined variable.

Example 11.2

PL/SQL Anchored Variable Declarations

```
DECLARE
 anOffTerm Offering.OffTerm%TYPE;
 anOffYear Offering.OffYear%TYPE;
 aCrsUnits Course.CrsUnits%TYPE;
 aSalary1 DECIMAL(10,2);
 aSalary2 aSalary1%TYPE;
```

Oracle also provides structured data types for combining primitive data types. Oracle supports variable length arrays (VARRAY), tables (TABLE), and records (RECORD) for combining data types. For information about the structured data types, you should consult the online Oracle documentation such as the *PL/SQL User's Guide*.

Assignment statements involve a variable, the assignment symbol (:=), and an expression on the right. Expressions can include combinations of constants, variables, functions, and operators. When evaluated, an expression produces a value. Example 11.3 demonstrates assignment statements with various expression elements.

Example 11.3

PL/SQL Assignment Examples

It is assumed that variables used in the examples have been previously declared. Lines beginning with double hyphens are comments.

```
aFixedLengthString := 'XYZABC';
-- || is the string concatenation function
aVariableLengthString := aFixedLengthString || 'ABCDEF';
anIntegerVariable := anAge + 1;
aFixedPrecisionVariable := aSalary * 0.10;
-- To_Date is the date conversion function
aDateVariable := To_Date('30-Jun-2017');
```

Conditional Statements PL/SQL provides the IF and CASE statements for conditional decision making. In an IF statement, a logical expression or condition evaluating to TRUE, FALSE, or NULL, follows the IF keyword. Conditions contain comparison expressions using the comparison operators (Table 11-2) connected using the logical operators AND, OR, and NOT. Parentheses can be used to clarify the order of evaluation in complex conditions. When mixing the AND and OR operators, you should use parentheses to clarify the order of evaluation. Conditions are evaluated using the three-valued logic described in Chapter 9 (Section 9.4).

Similar to other languages, the PL/SQL IF statement has multiple variations. Example 11-4 depicts the first variation known as the IF-THEN statement. Any number of statements can be used between the THEN and END IF keywords. Example 11-5 depicts the second variation known as the IF-THEN-ELSE statement. This statement allows a set of alternative statements if the condition is false. The third variation (IF-THEN-ELSIF) depicted in Example 11-6 allows a condition for each ELSIF clause along with a final ELSE clause if all conditions are false.

IF-THEN Statement

```
IF condition THEN
   sequence of statements
END IF;
```

IF-THEN-ELSE Statement

```
IF condition THEN
   sequence of statements 1
ELSE
   sequence of statements 2
END IF;
```

TABLE 11-2
List of PL/SQL Comparison Operators

Operator	Meaning
=	Equal to
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

IF-THEN-ELSIF Statement

```
IF condition1 THEN
   sequence of statements 1
ELSIF condition2 THEN
   sequence of statements 2
ELSIF conditionN THEN
   sequence of statements N
ELSE
   sequence of statements N+1
END IF;
```

Example 11.4

IF-THEN Statement Examples

It is assumed that variables used in the examples have been previously declared.

```
IF aCrsUnits > 3 THEN
   CourseFee := BaseFee + aCrsUnits * VarFee;
END IF;

IF anOffLimit > NumEnrolled OR CourseOverRide = TRUE THEN
   NumEnrolled := NumEnrolled + 1;
   EnrDate := SysDate;
END IF;
```

Example 11.5

IF-THEN-ELSE Statement Examples

It is assumed that variables used in the examples have been previously declared.

```
IF aCrsUnits > 3 THEN
   CourseFee := BaseFee + ((aCrsUnits – 3) * VarFee);
ELSE
   CourseFee := BaseFee;
END IF;

IF anOffLimit > NumEnrolled OR CourseOverRide = TRUE THEN
   NumEnrolled := NumEnrolled + 1;
   EnrDate := SysDate;
ELSE
   Enrolled := FALSE;
END IF;
```

Example 11.6

IF-THEN-ELSIF Statement Examples

It is assumed that variables used in the examples have been previously declared.

```
IF anOffTerm = 'Fall' AND Enrolled := TRUE THEN
   FallEnrolled := FallEnrolled + 1;
```

```
ELSIF anOffTerm = 'Spring' AND Enrolled := TRUE THEN
   SpringEnrolled := SpringEnrolled + 1;
ELSE
   SummerEnrolled := SummerEnrolled + 1;
END IF;

IF aStdClass = 'FR' THEN
   NumFR := NumFR + 1;
   NumStudents := NumStudents + 1;
ELSIF aStdClass = 'SO' THEN
   NumSO := NumSO + 1;
   NumStudents := NumStudents + 1;
ELSIF aStdClass = 'JR' THEN
   NumJR := NumJR + 1;
   NumStudents := NumStudents + 1;
ELSIF aStdClass = 'SR' THEN
   NumSR := NumSR + 1;
   NumStudents := NumStudents + 1;
END IF;
```

The CASE statement uses a selector instead of condition. A selector is an expression whose value determines a decision. Example 11.7 shows a CASE statement corresponding to the second part of Example 11.6. The CASE statement was first introduced in PL/SQL for Oracle 9i. Previous Oracle versions give a syntax error for Example 11.7.

Example 11.7

CASE Statement Example Corresponding to the Second Part of Example 11-6

It is assumed that variables used in the example have been previously declared.

```
CASE aStdClass
 WHEN 'FR' THEN
   NumFR := NumFR + 1;
   NumStudents := NumStudents + 1;
 WHEN 'SO' THEN
   NumSO := NumSO + 1;
   NumStudents := NumStudents + 1;
 WHEN 'JR' THEN
   NumJR := NumJR + 1;
   NumStudents := NumStudents + 1;
 WHEN 'SR' THEN
   NumSR := NumSR + 1;
   NumStudents := NumStudents + 1;
END CASE;
```

CASE Statement

```
CASE selector
 WHEN expression1 THEN sequence of statements 1
 WHEN expression2 THEN sequence of statements 2
 WHEN expressionN THEN sequence of statements N
 [ ELSE sequence of statements N+1 ]
END CASE;
```

Iteration Statements PL/SQL contains three iteration statements along with a statement to terminate a loop. The FOR LOOP statement iterates over a range of integer values, as shown in Example 11.8. The WHILE LOOP statement iterates until a stopping condition is false, as shown in Example 11.9. The LOOP statement iterates until an EXIT statement ceases termination, as shown in Example 11.10. Note that the EXIT statement can also be used in the FOR LOOP and the WHILE LOOP statements to cause early termination of a loop.

Example 11.8

FOR LOOP Statement Example

It is assumed that variables used in the example have been previously declared.

```
FOR Idx IN 1 .. NumStudents LOOP
   TotalUnits := TotalUnits + (Idx * aCrsUnits);
END LOOP;
```

Example 11.9

WHILE LOOP statement corresponding to Example 11.8

```
Idx := 1;
WHILE Idx <= NumStudents LOOP
   TotalUnits := TotalUnits + (Idx * aCrsUnits);
   Idx := Idx + 1;
END LOOP;
```

Example 11.10

LOOP statement corresponding to Example 11.8

```
Idx := 1;
LOOP
   TotalUnits := TotalUnits + (Idx * aCrsUnits);
   Idx := Idx + 1;
   EXIT WHEN Idx > NumStudents;
END LOOP;
```

FOR LOOP Statement

```
FOR variable IN BeginExpr .. EndExpr LOOP
  sequence of statements
END LOOP;
```

WHILE LOOP Statement

```
WHILE condition LOOP
  sequence of statements
END LOOP;
```

LOOP Statement

```
LOOP
  sequence of statements containing an EXIT statement
END LOOP;
```

11.1.4 Executing PL/SQL Statements in Anonymous Blocks

PL/SQL is a block structured language. You will learn about named blocks in Section 11.2. This section introduces anonymous blocks so that you can execute statement examples in SQL Developer, a visual tool for creating database objects, building data models, monitoring database activity, and testing database procedures. SQL Developer replaced SQL *Plus, the traditional command-oriented tool for submitting SQL and PL-SQL statements. Anonymous blocks also are useful for testing procedures and triggers. Before presenting anonymous blocks, a brief introduction to the SQL Developer is provided.

SQL Developer provides the SQL Worksheet tool for executing SQL, PL/SQL, and SQL *Plus commands. You can use the Run Statement button to execute SQL statements, and the Run Script button to execute a collection of SQL, PL/SQL, and SQL *Plus statements. Table 11-3 lists common SQL *Plus commands useful in the SQL Developer. The SQL Developer documentation provides details about the SQL *Plus commands supported in the SQL Worksheet tool. The anonymous procedures and related SQL *Plus commands in Examples 11.11 and 11.12 can be executed in the SQL Worksheet using the Run Script button.

A PL/SQL block contains an optional declaration section (DECLARE keyword), an executable section (BEGIN keyword), and an optional exception section (EXCEPTION keyword). The presentation here depicts anonymous blocks containing declaration and executable sections. Section 11.2 depicts the exception section.

Block Structure

```
[ DECLARE
  sequence of declarations ]
BEGIN
  sequence of statements
[ EXCEPTION
  sequence of statements to respond to exceptions ]
END;
```

To demonstrate anonymous blocks, Example 11.11 computes the sum and product of integers 1 to 10. The *Dbms_Output.Put_Line* procedure displays the results. The *Dbms_Output* package contains procedures and functions to read and write lines in a buffer. Example 11.12 modifies Example 11.11 to compute the sum of the odd numbers and the product of the even numbers.

TABLE 11-3
List of SQL *Plus Commands useful in the SQL Developer

Command	Example and Meaning
SET	SET SERVEROUTPUT ON causes the results of PL/SQL statements to be displayed.
SHOW	SHOW ERRORS causes compilation errors to be displayed.
SPOOL	SPOOL FileName causes the output to be written to FileName. SPOOL OFF stops spooling to a file.
/	Use on a line by itself to terminate a collection of statements or SQL *Plus commands

Example 11.11

Anonymous Block to Compute the Sum and the Product

The first line (SET command) and the last line (/) are not part of the anonymous block.

```
-- SQL *Plus command
SET SERVEROUTPUT ON;
-- Anonymous block
DECLARE
 TmpSum      INTEGER;
 TmpProd     INTEGER;
 Idx         INTEGER;
BEGIN
-- Initialize temporary variables
 TmpSum := 0;
 TmpProd := 1;
 -- Use a loop to compute the sum and product
 FOR Idx IN 1 .. 10 LOOP
    TmpSum := TmpSum + Idx;
    TmpProd := TmpProd * Idx;
 END LOOP;
 -- Display the results
 Dbms_Output.Put_Line('Sum is ' || To_Char(TmpSum));
 Dbms_Output.Put_Line('Product is ' || To_Char(TmpProd));
END;
/
```

Example 11.12

Anonymous Block to Compute the Sum of the Even Numbers and the Product of the Odd Numbers

The SET command is not necessary if it was used for Example 11.11 in the same session of SQL *Plus.

```
SET SERVEROUTPUT ON;
DECLARE
 TmpSum      INTEGER;
 TmpProd     INTEGER;
 Idx         INTEGER;
BEGIN
 -- Initialize temporary variables
 TmpSum := 0;
 TmpProd := 1;
 -- Use a loop to compute the sum of the even numbers and
 -- the product of the odd numbers.
 -- Mod(X,Y) returns the integer remainder of X/Y.
 FOR Idx IN 1 .. 10 LOOP
    IF Mod(Idx,2) = 0 THEN -- even number
      TmpSum := TmpSum + Idx;
    ELSE
      TmpProd := TmpProd * Idx;
    END IF;
 END LOOP;
 -- Display the results
 Dbms_Output.Put_Line('Even sum is ' || To_Char(TmpSum));
 Dbms_Output.Put_Line('Odd product is ' || To_Char(TmpProd));
END;
/
```

11.2 STORED PROCEDURES

With background about database programming languages and PL/SQL, you are now ready to learn about stored procedures. Programming languages have supported procedures since the early days of business computing. Procedures support the management of complexity by allowing computing tasks to be divided into manageable chunks. A database procedure is like a programming language procedure except that it is managed by the DBMS, not the programming environment. The following list explains the reasons for a DBMS to manage procedures:

- A DBMS can compile the programming language code along with the SQL statements in a stored procedure. In addition, a DBMS can detect when the SQL statements in a procedure need to be recompiled due to changes in database definitions.
- Stored procedures allow flexibility for client-server development. The stored procedures are saved on a server and not replicated on each client. In the early days of client-server computing, the ability to store procedures on a server was a major motivation for stored procedures. With the development of distributed objects on the Web, this motivation is not as important now because there are other technologies for managing stored procedures on remote servers.
- Stored procedures allow for the development of more complex operators and functions than supported by SQL. Chapter 19 describes the importance of specialized procedures and functions in object-oriented databases.
- Database administrators can manage stored procedures with the same tools for managing other parts of a database application. Most importantly, stored procedures are managed by the security system of the DBMS.

This section covers PL/SQL procedures, functions, and packages. Some additional parts of PL/SQL (cursors and exceptions) are shown to demonstrate the utility of stored procedures. Testing scripts assume that the university tables are populated according to the data on the textbook's website.

11.2.1 PL/SQL Procedures

In PL/SQL, a procedure is a named block with an optional set of parameters. Each parameter contains a parameter name, a usage (IN, OUT, IN OUT), and a data type. An input parameter (IN) should not be changed inside a procedure. An output parameter (OUT) is given a value inside a procedure. An input-output parameter (IN OUT) should have a value provided outside a procedure but can be changed inside a procedure. The data type specification should not include any constraints such as length. For example, you should use the data type VARCHAR2 without a length specification for a parameter containing text.

Procedure Structure

```
CREATE [OR REPLACE] PROCEDURE ProcedureName
  [ ( Parameter1, …, ParameterN ) ]
IS
  [ sequence of declarations ]
BEGIN
  sequence of statements
[ EXCEPTION
  sequence of statements to respond to exceptions ]
END;
```

As a simple example, the procedure *pr_InsertRegistration* in Example 11.13 inserts a row into the Registration table of the university database. The input parameters, defined using anchored data types, provide the values to insert. The *Dbms_Output.*

Put_Line procedure call displays a message that the insert was successful. In the testing code that follows the CREATE PROCEDURE statement, the ROLLBACK statement eliminates changes made by all SQL statements. ROLLBACK statements are useful in testing code when database changes should not be permanent.

Example 11.13

Procedure to Insert a Row into the *Registration* Table along with Code to Test the Procedure

```
CREATE OR REPLACE PROCEDURE pr_InsertRegistration
(aRegNo IN Registration.RegNo%TYPE,
 aStdNo IN Registration.StdNo%TYPE,
 aRegStatus IN Registration.RegStatus%TYPE,
 aRegDate IN Registration.RegDate%TYPE,
 aRegTerm IN Registration.RegTerm%TYPE,
 aRegYear IN Registration.RegYear%TYPE) IS
-- Insert a new registration using parameter values
BEGIN
INSERT INTO Registration
       (RegNo, StdNo, RegStatus, RegDate, RegTerm, RegYear)
VALUES (aRegNo, aStdNo, aRegStatus, aRegDate, aRegTerm, aRegYear);

dbms_output.put_line('Row added to Registration table');
END;
/

-- Testing code
SET SERVEROUTPUT ON;
-- Number of rows before the procedure execution
SELECT COUNT(*) FROM Registration;

BEGIN
pr_InsertRegistration
 (1275,'901-23-4567','F',To_Date('27-Feb-2017'),'Spring',2017);
END;
/
-- Number of rows after the procedure execution
SELECT COUNT(*) FROM Registration;
-- Delete the inserted row using the ROLLBACK statement
ROLLBACK;
```

To enable reuse of *pr_InsertRegistration* by other procedures, you should replace the output display with an output parameter indicating the success or failure of the insertion. Example 11.14 modifies Example 11.13 to use an output parameter. The OTHERS exception catches a variety of errors such as a violation of a primary key constraint or a foreign key constraint. You should use the OTHERS exception when you do not need specialized code for each kind of exception. To catch a specific error, you should use a predefined exception (Table 11-4) or create a user-defined exception. Example 11.26 (in the trigger section) contains an example of a user-defined exception. After the procedure, the script includes test cases for a successful insert as well as a primary key constraint violation.

Example 11.14

Procedure to Insert a Row into the *Registration* Table Along with Code to Test the Procedure

```
CREATE OR REPLACE PROCEDURE pr_InsertRegistration
(aRegNo IN Registration.RegNo%TYPE,
 aStdNo IN Registration.StdNo%TYPE,
 aRegStatus IN Registration.RegStatus%TYPE,
 aRegDate IN Registration.RegDate%TYPE,
 aRegTerm IN Registration.RegTerm%TYPE,
 aRegYear IN Registration.RegYear%TYPE,
 aResult OUT BOOLEAN ) IS
-- Create a new registration
-- aResult is TRUE if successful, false otherwise.
BEGIN
aResult := TRUE;
INSERT INTO Registration
       (RegNo, StdNo, RegStatus, RegDate, RegTerm, RegYear)
VALUES (aRegNo, aStdNo, aRegStatus, aRegDate, aRegTerm, aRegYear);

EXCEPTION
WHEN OTHERS THEN aResult := FALSE;
END;
/

-- Testing code
SET SERVEROUTPUT ON;
-- Number of rows before the procedure execution
SELECT COUNT(*) FROM Registration;
DECLARE
 -- Output parameter declared in the calling block
 Result BOOLEAN;
BEGIN
-- This test should succeed.
-- Assign value to the output parameter (Result).
pr_InsertRegistration
(1275,'901-23-4567','F',To_Date('27-Feb-2017'),'Spring',2017,Result);
IF Result THEN
     dbms_output.put_line('Registration row added');
ELSE
     dbms_output.put_line('Registration row not added');
END IF;

-- This test should fail because of the duplicate primary key.
pr_InsertRegistration
(1275,'901-23-4567','F',To_Date('27-Feb-2017'),'Spring',2017,Result);
IF Result THEN
     dbms_output.put_line('Registration row added');
ELSE
     dbms_output.put_line('Registration row not added');
END IF;
END;
/

-- Number of rows after the procedure executions
SELECT COUNT(*) FROM Registration;
-- Delete inserted row
ROLLBACK;
```

TABLE 11-4
List of Common Predefined PL/SQL Exceptions

Exception	When Raised
Cursor_Already_Open	Attempt to open a cursor that has been previously opened
Dup_Val_On_Index	Attempt to store a duplicate value in a unique index
Invalid_Cursor	Attempt to perform an invalid operation on a cursor such as closing a cursor that was not previously opened
No_Data_Found	SELECT INTO statement returns no rows
Rowtype_Mismatch	Attempt to assign values with incompatible data types between a cursor and a variable
Timeout_On_Resource	Timeout[1] occurs such as when waiting for an exclusive lock
Too_Many_Rows	SELECT INTO statement returns more than one row

11.2.2 PL/SQL Functions

Functions should return values instead of manipulating output variables and having side effects such as inserting rows into a table. You should always use a procedure if you want to have more than one result and/or have a side effect. Functions should be usable in expressions, meaning that a function call can be replaced by the value it returns. A PL/SQL function is similar to a procedure in that both contain a parameter list. However, a function should use only input parameters. After the parameter list, the return data type is defined without any constraints such as length. In the function body, the sequence of statements should include a RETURN statement to generate the function's output value.

Function Structure

```
CREATE [OR REPLACE] FUNCTION FunctionName
  [ (Parameter1, …, ParameterN) ]
RETURN DataType
IS
  [ sequence of declarations ]
BEGIN
  sequence of statements including a RETURN statement
[ EXCEPTION
  sequence of statements to respond to exceptions ]
END;
```

Procedures versus Functions: use a procedure if the code should have more than one result or a side effect. Functions should be usable in expressions, meaning that a function call can be replaced by the value it returns. To enable functions to be used in expressions, functions should only use input parameters.

As a simple example, the function *fn_RetrieveStdName* in Example 11.15 retrieves the name of a student given the student number. The predefined exception *No_Data_Found* is true if the SELECT statement does not return at least one row. The SELECT statement uses the INTO clause to associate the variables with the database columns. The INTO clause can be used only when the SELECT statement returns at most one row. If an INTO clause is used when a SELECT statement returns more than one row, an exception is generated. The *Raise_Application_Error* procedure displays an error message and an error number. This predefined procedure is useful to handle unexpected errors.

[1] Chapter 17 explains the usage of timeouts with transaction locking to prevent deadlocks.

Example 11.15

Function to Retrieve the Student Name Given the Student Number

```
CREATE OR REPLACE FUNCTION fn_RetrieveStdName
(aStdNo IN Student.StdNo%type) RETURN VARCHAR2 IS
-- Retrieves the student name (concatenate first and last name)
-- given a student number. If the student does not exist,
-- return null.
aFirstName Student.StdFirstName%TYPE;
aLastName Student.StdLastName%TYPE;

BEGIN
SELECT StdFirstName, StdLastName
 INTO aFirstName, aLastName
 FROM Student
 WHERE StdNo = aStdNo;

RETURN(aLastName || ', ' || aFirstName);

EXCEPTION
-- No_Data_Found is raised if the SELECT statement returns no data.
 WHEN No_Data_Found THEN
  RETURN(NULL);

 WHEN OTHERS THEN
   raise_application_error(-20001, 'Database error');

END;
/
-- Testing code
SET SERVEROUTPUT ON;
DECLARE
aStdName VARCHAR2(50);
BEGIN
-- This call should display a student name.
aStdName := fn_RetrieveStdName('901-23-4567');
IF aStdName IS NULL THEN
     dbms_output.put_line('Student not found');
ELSE
     dbms_output.put_line('Name is ' || aStdName);
END IF;

-- This call should not display a student name.
aStdName := fn_RetrieveStdName('905-23-4567');
IF aStdName IS NULL THEN
     dbms_output.put_line('Student not found');
ELSE
     dbms_output.put_line('Name is ' || aStdName);
END IF;
END;
/
```

Example 11.16 shows a function with a more complex query than the function in Example 11.15. The testing code contains two cases to test for an existing student and a non-existing student along with a SELECT statement that uses the function in the result. An important benefit of functions is that they can be used in expressions in SELECT statements.

Example 11.16

Function to Compute the Weighted GPA Given the Student Number and Year

```
CREATE OR REPLACE FUNCTION fn_ComputeWeightedGPA
(aStdNo IN Student.StdNo%TYPE, aYear IN Offering.OffYear%TYPE)
   RETURN NUMBER IS
-- Computes the weighted GPA given a student number and year.
-- Weighted GPA is the sum of units times the grade
-- divided by the total units.
-- If the student does not exist, return null.
WeightedGPA NUMBER;

BEGIN
SELECT SUM(EnrGrade*CrsUnits)/SUM(CrsUnits)
 INTO WeightedGPA
 FROM Student, Registration, Enrollment, Offering, Course
 WHERE Student.StdNo = aStdNo
   AND Offering.OffYear = aYear
   AND Student.StdNo = Registration.StdNo
   AND Registration.RegNo = Enrollment.RegNo
   AND Enrollment.OfferNo = Offering.OfferNo
   AND Offering.CourseNo = Course.CourseNo;

RETURN(WeightedGPA);

EXCEPTION
 WHEN No_Data_Found THEN
  RETURN(NULL);

 WHEN OTHERS THEN
   raise_application_error(-20001, 'Database error');

END;
/
-- Testing code
SET SERVEROUTPUT ON;
DECLARE
aGPA DECIMAL(3,2);
BEGIN
-- This call should display a weighted GPA.
aGPA := fn_ComputeWeightedGPA('901-23-4567', 2017);
IF aGPA IS NULL THEN
     dbms_output.put_line('Student or enrollments not found');
ELSE
     dbms_output.put_line('Weighted GPA is ' || to_char(aGPA));
END IF;

-- This call should not display a weighted GPA.
aGPA := fn_ComputeWeightedGPA('905-23-4567', 2017);
IF aGPA IS NULL THEN
     dbms_output.put_line('Student or enrollments not found');
ELSE
     dbms_output.put_line('Weighted GPA is ' || to_char(aGPA));
END IF;
END;
/
-- Use the function in a query
SELECT StdNo, StdFirstName, StdLastName,
       fn_ComputeWeightedGPA(StdNo, 2017) AS WeightedGPA
FROM Student;
```

11.2.3 Using Cursors

The previous procedures and functions are rather simple as they involve retrieval of a single row. More complex procedures and functions involve iteration through multiple rows using a cursor. PL/SQL provides cursor declaration (explicit or implicit), a specialized FOR statement for cursor iteration, cursor attributes to indicate the status of cursor operations, and statements to perform actions on explicit cursors. PL/SQL supports static cursors in which the SQL statement is known at compile-time as well as dynamic cursors in which the SQL statement is not determined until run-time.

Example 11.17 depicts an implicit cursor to return the class rank of a student in an offering. Implicit cursors are not declared in the DECLARE section. Instead, implicit cursors are declared, opened, and iterated inside a FOR statement. In Example 11.17, the FOR statement iterates through each row of the SELECT statement using the implicit cursor *EnrollRec*. The SELECT statement sorts the result in descending order by

> **Implicit PL/SQL Cursor:** a cursor that is neither explicitly declared nor explicitly opened. Instead a special version of the FOR statement declares, opens, iterates, and closes a locally named SELECT statement. An implicit cursor cannot be referenced outside of the FOR statement in which it is declared.

Example 11.17

Using an Implicit Cursor to Determine the Class Rank of a Given Student and Offering

```
CREATE OR REPLACE FUNCTION fn_DetermineRank
(aStdNo IN Student.StdNo%TYPE, anOfferNo IN Offering.OfferNo%TYPE)
   RETURN INTEGER IS
-- Determines the class rank given a StdNo and OfferNo.
-- Computes dense ranking with no gap in ranks for matching grades
-- Uses an implicit cursor.
-- If the student or offering do not exist, return 0.
TmpRank INTEGER :=0;
PrevEnrGrade Enrollment.EnrGrade%TYPE := 9.9;
FOUND BOOLEAN := FALSE;

BEGIN
-- Loop through implicit cursor
FOR EnrollRec IN
 ( SELECT Student.StdNo, EnrGrade
    FROM Student, Registration, Enrollment
    WHERE Enrollment.OfferNo = anOfferNo
      AND Student.StdNo = Registration.StdNo
      AND Registration.RegNo = Enrollment.RegNo
    ORDER BY EnrGrade DESC ) LOOP

    IF EnrollRec.EnrGrade < PrevEnrGrade THEN
    -- Increment the class rank when the grade changes
       TmpRank := TmpRank + 1;
       PrevEnrGrade := EnrollRec.EnrGrade;
    END IF;
    IF EnrollRec.StdNo = aStdNo THEN
       Found := TRUE;
       EXIT;
    END IF;
END LOOP;
```

```
IF Found THEN
  RETURN(TmpRank);
ELSE
  RETURN(0);
END IF;

EXCEPTION
WHEN OTHERS THEN
   raise_application_error(-20001, 'Database error');

END;
/
-- Testing code
SET SERVEROUTPUT ON;
-- Execute query to see test data
SELECT Student.StdNo, EnrGrade
 FROM Student, Registration, Enrollment
 WHERE Enrollment.OfferNo = 5679
   AND Student.StdNo = Registration.StdNo
   AND Registration.RegNo = Enrollment.RegNo
 ORDER BY EnrGrade DESC;

-- Test script
DECLARE
aRank INTEGER;
BEGIN
-- This call should return a rank of 6.
aRank := fn_DetermineRank('789-01-2345', 5679);
IF aRank > 0 THEN
     dbms_output.put_line('Rank is ' || to_char(aRank));
ELSE
     dbms_output.put_line('Student is not enrolled.');
END IF;

-- This call should return a rank of 0.
aRank := fn_DetermineRank('789-01-2005', 5679);
IF aRank > 0 THEN
     dbms_output.put_line('Rank is ' || to_char(aRank));
ELSE
     dbms_output.put_line('Student is not enrolled.');
END IF;
END;
/
```

enrollment grade. The function exits the FOR statement when the *StdNo* value matches the parameter value. The class rank is incremented only when the grade changes so that two students with the same grade have the same rank. The function computes a dense ranking with no gaps in ranks.

Example 11.18 depicts a procedure with an explicit cursor to return the class rank and the grade of a student in an offering. The explicit cursor *EnrollCursor* in the CURSOR statement contains offer number as a parameter. Explicit cursors must use parameters for nonconstant search values in the associated SELECT statement. The OPEN, FETCH, and CLOSE statements replace the FOR statement of Example 11.17. After the FETCH statement, the condition *EnrollCursor%NotFound* tests for the empty cursor.

PL/SQL supports a number of cursor attributes as listed in Table 11-5. When used with an explicit cursor, the cursor name precedes the cursor attribute. When used with an implicit cursor, the SQL keyword precedes the cursor attribute. For example, *SQL%RowCount* denotes the number of rows in an implicit cursor. The implicit cursor name is not used.

Explicit PL/SQL Cursor
a cursor that is declared with the CURSOR statement in the DECLARE section. Explicit cursors are usually manipulated by the OPEN, CLOSE, and FETCH statements. Explicit cursors can be referenced anyplace inside the BEGIN section.

Example 11.18

Using an Explicit Cursor to Determine the Class Rank and Grade of a Given Student and Offering

```
CREATE OR REPLACE PROCEDURE pr_DetermineRank
(aStdNo IN Student.StdNo%TYPE, anOfferNo IN Offering.OfferNo%TYPE,
 OutRank OUT INTEGER, OutGrade OUT Enrollment.EnrGrade%TYPE ) IS
-- Determines the class rank and grade for a given student number
-- and OfferNo using an explicit cursor.
-- Computes dense ranking with no gap in ranks for matching grades
-- If the student or offering do not exist, return 0.
TmpRank INTEGER :=0;
PrevEnrGrade Enrollment.EnrGrade%TYPE := 9.9;
Found BOOLEAN := FALSE;
TmpGrade Enrollment.EnrGrade%TYPE;
TmpStdNo Student.StdNo%TYPE;
-- Explicit cursor
CURSOR EnrollCursor (tmpOfferNo Offering.OfferNo%TYPE) IS
   SELECT Student.StdNo, EnrGrade
    FROM Student, Registration, Enrollment
    WHERE Enrollment.OfferNo = anOfferNo
      AND Student.StdNo = Registration.StdNo
      AND Registration.RegNo = Enrollment.RegNo
    ORDER BY EnrGrade DESC;

BEGIN
-- Open and loop through explicit cursor
OPEN EnrollCursor(anOfferNo);
LOOP
    FETCH EnrollCursor INTO TmpStdNo, TmpGrade;
    EXIT WHEN EnrollCursor%NotFound;
    IF TmpGrade < PrevEnrGrade THEN
    -- Increment the class rank when the grade changes
       TmpRank := TmpRank + 1;
       PrevEnrGrade := TmpGrade;
    END IF;
    IF TmpStdNo = aStdNo THEN
       Found := TRUE;
       EXIT;
    END IF;
END LOOP;

CLOSE EnrollCursor;
IF Found THEN
  OutRank := TmpRank;
  OutGrade := PrevEnrGrade;
ELSE
  OutRank := 0;
  OutGrade := 0;
END IF;

EXCEPTION
WHEN OTHERS THEN
  raise_application_error(-20001, 'Database error');
END;
/
-- Testing code
SET SERVEROUTPUT ON;
-- Execute query to see test data
```

```
SELECT Student.StdNo, EnrGrade
 FROM Student, Registration, Enrollment
 WHERE Student.StdNo = Registration.StdNo
   AND Registration.RegNo = Enrollment.RegNo
   AND Enrollment.OfferNo = 5679
 ORDER BY EnrGrade DESC;

-- Test script
DECLARE
aRank INTEGER;
aGrade Enrollment.EnrGrade%TYPE;
BEGIN
-- This call should produce a rank of 6.
pr_DetermineRank('789-01-2345', 5679, aRank, aGrade);
IF aRank > 0 THEN
     dbms_output.put_line('Rank is ' || to_char(aRank) || '.');
     dbms_output.put_line('Grade is ' || to_char(aGrade) || '.');
ELSE
     dbms_output.put_line('Student is not enrolled.');
END IF;

-- This call should produce a rank of 0.
pr_DetermineRank('789-01-2005', 5679, aRank, aGrade);
IF aRank > 0 THEN
     dbms_output.put_line('Rank is ' || to_char(aRank) || '.');
     dbms_output.put_line('Grade is ' || to_char(aGrade) || '.');
ELSE
     dbms_output.put_line('Student is not enrolled.');
END IF;
END;
/
```

TABLE 11-5
List of Common Cursor Attributes

Cursor Attribute	Value
%IsOpen	True if cursor is open
%Found	True if cursor is not empty following a FETCH statement
%NotFound	True if cursor is empty following a FETCH statement
%RowCount	Number of rows fetched. After each FETCH, the RowCount is incremented

11.2.4 PL/SQL Packages

Packages support a larger unit of modularity than procedures or functions. A package may contain procedures, functions, exceptions, variables, constants, types, and cursors. By grouping related objects together, a package provides easier reuse than individual procedures and functions. Oracle provides many predefined packages such as the *DBMS_Output* package containing groups of related objects. In addition, a package separates a public interface from a private implementation to support reduced software maintenance efforts. Changes to a private implementation do not affect the usage of a package through its interface. Chapter 19 on object databases provides more details about the benefits of larger units of modularity.

A package interface contains the definitions of procedures and functions along with other objects that can be specified in the DECLARE section of a PL/SQL block. All objects in a package interface are public. Example 11.19 demonstrates the interface for a package combining some of the procedures and functions presented in previous sections.

Package Interface Structure

```
CREATE [OR REPLACE] PACKAGE PackageName IS
[ Constant, variable, and type declarations ]
[ Cursor declarations ]
[ Exception declarations ]
[ Procedure definitions ]
[ Function definitions ]
END PackageName;
```

Example 11.19

Package Interface Containing Related Procedures and Functions for the University Database

```
CREATE OR REPLACE PACKAGE pck_University IS
PROCEDURE pr_DetermineRank
 (aStdNo IN Student.StdNo%TYPE, anOfferNo IN Offering.OfferNo%TYPE,
  OutRank OUT INTEGER, OutGrade OUT Enrollment.EnrGrade%TYPE );
FUNCTION fn_ComputeWeightedGPA
(aStdNo IN Student.StdNo%TYPE, aYear IN Offering.OffYear%TYPE)
   RETURN NUMBER;
END pck_University;
/
```

A package implementation or body contains the private details of a package. For each object in the package interface, the package body must define an implementation. In addition, private objects can be defined in a package body. Private objects can be used only inside the package body. External users of a package cannot access private objects. Example 11.20 demonstrates the body for the package interface in Example 11.19. Note that each procedure or function terminates with an END statement containing the procedure or function name. Otherwise the procedure and function implementations are identical to creating a procedure or function outside of a package.

Package Body Structure

```
CREATE [OR REPLACE] PACKAGE BODY PackageName IS
[ Variable and type declarations ]
[ Cursor declarations ]
[ Exception declarations ]
[ Procedure implementations ]
[ Function implementations ]
[ BEGIN sequence of statements ]
[ EXCEPTION exception handling statements ]
END PackageName;
```

To use the objects in a package, you need to use the package name before the object name. In Example 11.21, you should note that the package name (*pck_University*) precedes the procedure and function names.

Example 11.20

Package Body Containing Implementations of Procedures and Functions

```
CREATE OR REPLACE PACKAGE BODY pck_University IS
PROCEDURE pr_DetermineRank
 (aStdNo IN Student.StdNo%TYPE, anOfferNo IN Offering.OfferNo%TYPE,
  OutRank OUT INTEGER, OutGrade OUT Enrollment.EnrGrade%TYPE ) IS
-- Determines the class rank and grade for a given student number
-- and OfferNo using an explicit cursor.
-- If the student or offering do not exist, return 0.
TmpRank INTEGER :=0;
PrevEnrGrade Enrollment.EnrGrade%TYPE := 9.9;
Found BOOLEAN := FALSE;
TmpGrade Enrollment.EnrGrade%TYPE;
TmpStdNo Student.StdNo%TYPE;
-- Explicit cursor
CURSOR EnrollCursor (tmpOfferNo Offering.OfferNo%TYPE) IS
   SELECT Student.StdNo, EnrGrade
    FROM Student, Registration, Enrollment
    WHERE Enrollment.OfferNo = anOfferNo
      AND Student.StdNo = Registration.StdNo
      AND Registration.RegNo = Enrollment.RegNo
    ORDER BY EnrGrade DESC;

BEGIN
-- Open and loop through explicit cursor
OPEN EnrollCursor(anOfferNo);
LOOP
    FETCH EnrollCursor INTO TmpStdNo, TmpGrade;
    EXIT WHEN EnrollCursor%NotFound;
    IF TmpGrade < PrevEnrGrade THEN
    -- Increment the class rank when the grade changes
       TmpRank := TmpRank + 1;
       PrevEnrGrade := TmpGrade;
    END IF;
    IF TmpStdNo = aStdNo THEN
       Found := TRUE;
       EXIT;
    END IF;
END LOOP;

CLOSE EnrollCursor;
IF Found THEN
  OutRank := TmpRank;
  OutGrade := PrevEnrGrade;
ELSE
  OutRank := 0;
  OutGrade := 0;
END IF;

EXCEPTION
WHEN OTHERS THEN
  raise_application_error(-20001, 'Database error');
END pr_DetermineRank;

FUNCTION fn_ComputeWeightedGPA
(aStdNo IN Student.StdNo%TYPE, aYear IN Offering.OffYear%TYPE)
   RETURN NUMBER IS
-- Computes the weighted GPA given a student number and year.
-- Weighted GPA is the sum of units times the grade
```

```
-- divided by the total units.
-- If the student does not exist, return null.
WeightedGPA NUMBER;

BEGIN

SELECT SUM(EnrGrade*CrsUnits)/SUM(CrsUnits)
 INTO WeightedGPA
 FROM Student, Registration, Enrollment, Offering, Course
 WHERE Student.StdNo = aStdNo
   AND Offering.OffYear = aYear
   AND Student.StdNo = Registration.StdNo
   AND Registration.RegNo = Enrollment.RegNo
   AND Enrollment.OfferNo = Offering.OfferNo
   AND Offering.CourseNo = Course.CourseNo;

RETURN(WeightedGPA);

EXCEPTION
 WHEN no_data_found THEN
  RETURN(NULL);

 WHEN OTHERS THEN
   raise_application_error(-20001, 'Database error');

END fn_ComputeWeightedGPA;
END pck_University;
/
```

Example 11.21

Script to Use the Procedures and Functions of the University Package

```
SET SERVEROUTPUT ON;
DECLARE
aRank INTEGER;
aGrade Enrollment.EnrGrade%TYPE;
aGPA NUMBER;
BEGIN
-- This call should produce a rank of 6.
pck_University.pr_DetermineRank('789-01-2345', 5679, aRank, aGrade);
IF aRank > 0 THEN
     dbms_output.put_line('Rank is ' || to_char(aRank) || '.');
     dbms_output.put_line('Grade is ' || to_char(aGrade) || '.');
ELSE
     dbms_output.put_line('Student is not enrolled.');
END IF;
-- This call should display a weighted GPA.
aGPA := pck_University.fn_ComputeWeightedGPA('901-23-4567', 2017);
IF aGPA IS NULL THEN
     dbms_output.put_line('Student or enrollments not found');
ELSE
     dbms_output.put_line('Weighted GPA is ' || to_char(aGPA));
END IF;
END;
/
```

11.3 TRIGGERS

Triggers are rules managed by a DBMS. Because a trigger involves an event, a condition, and a sequence of actions, it also is known as an event-condition-action rule. Writing the action part or trigger body is similar to writing a procedure or a function except that a trigger has no parameters. Triggers are executed by the rule system of the DBMS not by explicit calls as for procedures and functions. Triggers officially became part of SQL:1999 although most DBMS vendors implemented triggers long before the release of SQL:1999.

Trigger
a rule that is stored and executed by a DBMS. Because a trigger involves an event, a condition, and a sequence of actions, it also is known as an event-condition-action rule.

This section covers Oracle triggers with background about SQL:2016 triggers. The first part of this section discusses the reasons that triggers are important parts of database application development and provides a classification of triggers. The second part demonstrates trigger coding in PL/SQL for a variety of common tasks. The third part presents specialized triggers for maintaining generalization hierarchies and processing view updates using the Oracle INSTEAD OF trigger event. The final part presents the trigger execution procedures of Oracle and SQL:2016.

11.3.1 Motivation and Classification of Triggers

Triggers are widely implemented in DBMSs because they have a variety of uses in business applications. The following list explains typical uses of triggers.

- *Complex integrity constraints:* Integrity constraints that cannot be specified by constraints in CREATE TABLE statements. A typical restriction on constraints in CREATE TABLE statements is that columns from other tables cannot be referenced. Triggers allow reference to columns from multiple tables to overcome this limitation. An alternative to a trigger for a complex constraint is an assertion discussed in Chapter 16. However, most DBMSs do not support assertions so triggers are the only choice for complex integrity constraints.
- *Transition constraints:* Integrity constraints that compare the values before and after an update occurs. For example, you can write a trigger to enforce the transition constraint that salary increases do not exceed 10 percent.
- *Update propagation:* Update derived columns in related tables such as to maintain perpetual inventory balance or the seats remaining on a scheduled flight.
- *Exception reporting:* Create a record of unusual conditions as an alternative to rejecting a transaction. A trigger can also send a notification in an e-mail message. For example, instead of rejecting a salary increase of 10 percent, a trigger can create an exception record and notify a manager to review the salary increase.
- *Audit trail:* Create a historical record of a transaction such as a history of automated teller usage.
- *Generalization hierarchy simulation:* Perform update propagation to maintain generalization hierarchy relationships and enforce generalization hierarchy constraints. Although SQL:2016 supports generalization hierarchies for tables, many DBMSs do not support table generalization hierarchies. Triggers can be written to support generalization hierarchies converted into a table design as specified by the Generalization Hierarchy Rule (see Section 6.4.3).
- *Operations on updatable join views:* Map modification operations on complex views to underlying base tables. Triggers on updatable join views extend limitations that Oracle imposes on updatable join views.

SQL:2016 classifies triggers by granularity, timing, and applicable event. For granularity, a trigger can involve each row affected by an SQL statement or an entire SQL statement. Row triggers are more common than statement triggers. For timing, a trigger

can fire before or after an event. Typically, triggers for constraint checking fire before an event, while triggers updating related tables and performing other actions fire after an event. For applicable event, a trigger can apply to INSERT, UPDATE, and DELETE statements. Update triggers should specify a list of applicable columns.

Because the SQL:1999 trigger specification was defined in response to vendor implementations, most trigger implementations varied from the original specification in SQL:1999 and the revised specification in SQL:2016. Oracle supports most parts of the specification while adding proprietary extensions. An important extension is the INSTEAD OF trigger that fires in place of an event, not before or after an event. Oracle also supports data definition events and other database events. Microsoft SQL Server provides statement triggers with access to row data in place of row triggers. Thus, most DBMSs support the spirit of the SQL:2016 trigger specification in trigger granularity, timing, and applicable events but do not adhere strictly to the SQL:2016 trigger syntax.

11.3.2 Basic Trigger Development using Oracle PL/SQL

An Oracle trigger contains a trigger name, a timing specification, an optional referencing clause, an optional granularity, an optional WHEN clause, and a PL/SQL block for the body as explained in the following list:

- The timing specification involves the keywords BEFORE, AFTER, or INSTEAD OF along with a triggering event using the keywords INSERT, UPDATE, or DELETE. With the UPDATE event, you can specify an optional list of columns. To specify multiple events, you can use the OR keyword. Oracle also supports data definition and other database events, but these events are beyond the scope of this chapter.
- The referencing clause allows alias names for the old (values before triggering event) and new (values after triggering event) data that can be referenced in a trigger.
- The granularity is specified by the FOR EACH ROW keywords. If you omit these keywords, the trigger is a statement trigger.
- The WHEN clause restricts when a trigger fires or executes. Because Oracle has numerous restrictions on conditions in WHEN clauses, the WHEN clause is used infrequently.
- The body of a trigger looks like other PL/SQL blocks except that triggers have more restrictions on the statements in a block.

Oracle Trigger Structure

```
CREATE [OR REPLACE] TRIGGER TriggerName
TriggerTiming TriggerEvent
[ Referencing clause ]
[ FOR EACH ROW ]
[ WHEN ( Condition ) ]
[ DECLARE sequence of declarative statements ]
BEGIN sequence of statements
[ EXCEPTION exception handling statements ]
END;
```

Introductory Triggers and Testing Code To start on some simple Oracle triggers, Examples 11.22 through 11.24 contain triggers that fire respectively on every INSERT, UPDATE, and DELETE statement on the *Course* table. Example 11.25 demonstrates a trigger with a combined event that fires for every action on the *Course* table. The triggers in Examples 11.22 through 11.25 have no purpose except to depict a wide range of trigger syntax as explained in the following list.

- A common naming scheme for triggers identifies the table name, the triggering actions (I for INSERT, U for UPDATE, and D for DELETE), and the timing (B for BEFORE and A for AFTER). For example, the last part of the trigger name (DIUA) in Example 11.25 denotes the DELETE, INSERT, and UPDATE events along with the AFTER timing.
- In Example 11.25, the OR keyword in the trigger event specification supports compound events involving more than one event.
- There is no referencing clause as the default names for the old (*:OLD*) and the new (*:NEW*) row are used in the trigger bodies.

Example 11.22

Trigger That Fires for INSERT statements on the *Course* Table Along with Testing Code to Fire the Trigger

```
CREATE OR REPLACE TRIGGER tr_Course_IA
AFTER INSERT
ON Course
FOR EACH ROW
BEGIN
     -- No references to OLD row because only NEW exists for INSERT
     dbms_output.put_line('Inserted Row');
     dbms_output.put_line('CourseNo: ' || :NEW.CourseNo);
     dbms_output.put_line('Course Description: ' || :NEW.CrsDesc);
     dbms_output.put_line('Course Units: ' || To_Char(:NEW.CrsUnits));
END;
/
-- Testing statements
SET SERVEROUTPUT ON;
INSERT INTO Course (CourseNo, CrsDesc, CrsUnits)
 VALUES ('IS485','Advanced Database Management',4);

ROLLBACK;
```

Example 11.23

Trigger That Fires for Every UPDATE Statement on the *Course* Table Along with Testing Code to Fire the Trigger

```
CREATE OR REPLACE TRIGGER tr_Course_UA
AFTER UPDATE
ON Course
FOR EACH ROW
BEGIN
     dbms_output.put_line('New Row Values');
     dbms_output.put_line('CourseNo: ' || :NEW.CourseNo);
     dbms_output.put_line('Course Description: ' || :NEW.CrsDesc);
     dbms_output.put_line('Course Units: ' || To_Char(:NEW.CrsUnits));
```

```
    dbms_output.put_line('Old Row Values');
    dbms_output.put_line('CourseNo: ' || :OLD.CourseNo);
    dbms_output.put_line('Course Description: ' || :OLD.CrsDesc);
    dbms_output.put_line('Course Units: ' || To_Char(:OLD.CrsUnits));
END;
/
-- Testing statements
SET SERVEROUTPUT ON;
-- Add row so it can be updated
INSERT INTO Course (CourseNo, CrsDesc, CrsUnits)
 VALUES ('IS485','Advanced Database Management',4);

UPDATE Course
 SET CrsUnits = 3
 WHERE CourseNo = 'IS485';

ROLLBACK;
```

Example 11.24

Trigger That Fires for Every DELETE Statement on the *Course* Table Along with Testing Code to Fire the Trigger

```
CREATE OR REPLACE TRIGGER tr_Course_DA
AFTER DELETE
ON Course
FOR EACH ROW
BEGIN
    -- No references to NEW row because only OLD exists for DELETE
    dbms_output.put_line('Deleted Row');
    dbms_output.put_line('CourseNo: ' || :OLD.CourseNo);
    dbms_output.put_line('Course Description: ' || :OLD.CrsDesc);
    dbms_output.put_line('Course Units: ' || To_Char(:OLD.CrsUnits));
END;
/
-- Testing statements
SET SERVEROUTPUT ON;
-- Insert row so that it can be deleted
INSERT INTO Course (CourseNo, CrsDesc, CrsUnits)
 VALUES ('IS485','Advanced Database Management',4);

DELETE FROM Course
 WHERE CourseNo = 'IS485';

ROLLBACK;
```

Triggers, unlike procedures, cannot be tested directly. Instead, you should use SQL statements that cause a trigger to fire. When the trigger in Example 11.25 fires for an INSERT statement, the old values are null. Likewise, when the trigger fires for a DELETE statement, the new values are null. When the trigger fires for an UPDATE statement, the old and new values are not null unless the table had null values before the update.

Example 11.25

Trigger with a Combined Event That Fires for Every action on the *Course* Table Along with testing Code to Fire the Trigger

```
CREATE OR REPLACE TRIGGER tr_Course_DIUA
AFTER INSERT OR UPDATE OR DELETE
ON Course
FOR EACH ROW
BEGIN
     dbms_output.put_line('Inserted Table');
     dbms_output.put_line('CourseNo: ' || :NEW.CourseNo);
     dbms_output.put_line('Course Description: ' || :NEW.CrsDesc);
     dbms_output.put_line('Course Units: ' || To_Char(:NEW.CrsUnits));

     dbms_output.put_line('Deleted Table');
     dbms_output.put_line('CourseNo: ' || :OLD.CourseNo);
     dbms_output.put_line('Course Description: ' || :OLD.CrsDesc);
     dbms_output.put_line('Course Units: ' || To_Char(:OLD.CrsUnits));
END;
/
-- Testing statements
SET SERVEROUTPUT ON;
INSERT INTO Course (CourseNo, CrsDesc, CrsUnits)
VALUES ('IS485','Advanced Database Management',4);

UPDATE Course
 SET CrsUnits = 3
 WHERE CourseNo = 'IS485';

DELETE FROM Course
 WHERE CourseNo = 'IS485';

ROLLBACK;
```

BEFORE ROW Trigger for Constraint Checking BEFORE ROW triggers typically are used for complex integrity constraints because BEFORE ROW triggers should not contain SQL manipulation statements. For example, enrolling in an offering involves a complex integrity constraint to ensure that a seat exists in the related offering. Example 11.26 demonstrates a BEFORE ROW trigger to ensure that a seat remains when a student enrolls in an offering. The trigger ensures that the number of students enrolled in the offering is less than the limit. The testing code inserts students and modifies the number of students enrolled so that the next insertion raises an error.

The trigger in Example 11.26 contains a user-defined exception to handle the error. The trigger code declares the name of the user-defined exception (*NoSeats*) in the DECLARE section using the EXCEPTION keyword. To cause the exception to occur, the RAISE statement uses the exception name typically as part of an IF statement. To handle the user-defined exception, the EXCEPTION section uses a WHEN clause with the name of the user-defined exception.

AFTER ROW Trigger for Update Propagation The testing code for the BEFORE ROW trigger in Example 11.26 includes an UPDATE statement to increment the number of students enrolled. An AFTER trigger can automate this task as shown in Example 11.27. The triggers in Examples 11.26 and 11.27 work in tandem. The BEFORE ROW trigger ensures that a seat remains in the offering. The AFTER ROW trigger then updates the related *Offering* row.

Example 11.26

Trigger to Ensure That a Seat Remains in an Offering

```
CREATE OR REPLACE TRIGGER tr_Enrollment_IB
-- This trigger ensures that the number of enrolled
-- students is less than the offering limit.
BEFORE INSERT
ON Enrollment
FOR EACH ROW
DECLARE
   anOffLimit Offering.OffLimit%TYPE;
   anOffNumEnrolled Offering.OffNumEnrolled%TYPE;
   -- user defined exception declaration
   NoSeats EXCEPTION;
   ExMessage VARCHAR(200);
BEGIN
   SELECT OffLimit, OffNumEnrolled
     INTO anOffLimit, anOffNumEnrolled
     FROM Offering
     WHERE Offering.OfferNo = :NEW.OfferNo;

   IF anOffNumEnrolled >= anOffLimit THEN
      RAISE NoSeats;
   END IF;
EXCEPTION
  WHEN NoSeats THEN
    -- error number between -20000 and -20999
     ExMessage := 'No seats remaining in offering ' ||
                    to_char(:NEW.OfferNo) || '.';
     ExMessage := ExMessage || 'Number enrolled: ' ||
               to_char(anOffNumEnrolled) || '. ';
     ExMessage := ExMessage || 'Offering limit: ' ||
               to_char(anOffLimit);
     Raise_Application_Error(-20001, ExMessage);
END;
/
-- Testing statements
SET SERVEROUTPUT ON;
-- See offering limit and number enrolled
SELECT OffLimit, OffNumEnrolled
 FROM Offering
 WHERE Offering.OfferNo = 5679;
-- Insert the last student
INSERT INTO Enrollment (RegNo, OfferNo, EnrGrade)
 VALUES (1234,5679,0);

-- update the number of enrolled students
UPDATE Offering
 SET OffNumEnrolled = OffNumEnrolled + 1
 WHERE OfferNo = 5679;

-- See offering limit and number enrolled
SELECT OffLimit, OffNumEnrolled
 FROM Offering
 WHERE Offering.OfferNo = 5679;
-- Insert a student beyond the limit
INSERT INTO Enrollment (RegNo, OfferNo, EnrGrade)
 VALUES (1236,5679,0);

ROLLBACK;
```

Example 11.27

Trigger to Update the Number of Enrolled Students in an Offering

```
CREATE OR REPLACE TRIGGER tr_Enrollment_IA
-- This trigger updates the number of enrolled
-- students in the related Offering row.
AFTER INSERT
ON Enrollment
FOR EACH ROW
BEGIN
   UPDATE Offering
    SET OffNumEnrolled = OffNumEnrolled + 1
    WHERE OfferNo = :NEW.OfferNo;
EXCEPTION
  WHEN OTHERS THEN
    RAISE_Application_Error(-20001, 'Database error');
END;
/
-- Testing statements
SET SERVEROUTPUT ON;
-- See the offering limit and number enrolled
SELECT OffLimit, OffNumEnrolled
 FROM Offering
 WHERE Offering.OfferNo = 5679;
-- Insert the last student
INSERT INTO Enrollment (RegNo, OfferNo, EnrGrade)
VALUES (1234,5679,0);

-- See the offering limit and number enrolled
SELECT OffLimit, OffNumEnrolled
 FROM Offering
 WHERE Offering.OfferNo = 5679;

ROLLBACK;
```

Combining Trigger Events to Reduce the Number of Triggers The triggers in Examples 11.26 and 11.27 involve insertions to the *Enrollment* table. Additional triggers are needed for updates to the *Enrollment.OfferNo* column and deletions of *Enrollment* rows.

As an alternative to separate triggers for events on the same table, one large BEFORE trigger and one large AFTER trigger can be written. Each trigger contains multiple events as shown in Examples 11.28 and 11.29. The action part of the trigger in Example 11.29 uses the keywords INSERTING, UPDATING, and DELETING to determine the triggering event. The script in Example 11.30 is rather complex because it tests two complex triggers.

There is no clear preference for many smaller triggers or fewer larger triggers. Although smaller triggers are easier to understand than larger triggers, the number of triggers is a complicating factor to understand interactions among triggers. Section 11.3.4 explains trigger execution procedures to clarify issues of trigger interactions.

Additional BEFORE ROW Trigger Examples BEFORE triggers can also be used for transition constraints and data standardization. Example 11.31 depicts a trigger for a transition constraint. The trigger contains a WHEN clause to restrict the trigger execution. Example 11.32 depicts a trigger to enforce uppercase usage for the faculty name columns. Although BEFORE triggers should not perform updates with SQL statements, they can change the new values as the trigger in Example 11.32 demonstrates.

Example 11.28

Trigger to Ensure That a Seat Remains in an Offering When Inserting or Updating an *Enrollment* Row

```
-- Drop the previous trigger to avoid interactions
DROP TRIGGER tr_Enrollment_IB;
CREATE OR REPLACE TRIGGER tr_Enrollment_IUB
-- This trigger ensures that the number of enrolled
-- students is less than the offering limit.
BEFORE INSERT OR UPDATE OF OfferNo
ON Enrollment
FOR EACH ROW
DECLARE
    anOffLimit Offering.OffLimit%TYPE;
    anOffNumEnrolled Offering.OffNumEnrolled%TYPE;
    NoSeats EXCEPTION;
    ExMessage VARCHAR(200);
BEGIN
   SELECT OffLimit, OffNumEnrolled
     INTO anOffLimit, anOffNumEnrolled
     FROM Offering
     WHERE Offering.OfferNo = :NEW.OfferNo;

   IF anOffNumEnrolled >= anOffLimit THEN
      RAISE NoSeats;
   END IF;
EXCEPTION
  WHEN NoSeats THEN
    -- error number between -20000 and -20999
     ExMessage := 'No seats remaining in offering ' ||
                    to_char(:NEW.OfferNo) || '.';
     ExMessage := ExMessage || 'Number enrolled: ' ||
               to_char(anOffNumEnrolled) || '. ';
     ExMessage := ExMessage || 'Offering limit: ' ||
               to_char(anOffLimit);
     raise_application_error(-20001, ExMessage);
END;
```

Example 11.29

Trigger to Update the Number of Enrolled Students in an Offering When Inserting, Updating, or Deleting an *Enrollment* Row

```
-- Drop the previous trigger to avoid interactions
DROP TRIGGER tr_Enrollment_IA;
CREATE OR REPLACE TRIGGER tr_Enrollment_DIUA
-- This trigger updates the number of enrolled
-- students the related offering row.
AFTER INSERT OR DELETE OR UPDATE OF OfferNo
ON Enrollment
FOR EACH ROW
```

```
BEGIN
 -- Increment the number of enrolled students for insert, update
 IF INSERTING OR UPDATING THEN
    UPDATE Offering
      SET OffNumEnrolled = OffNumEnrolled + 1
      WHERE OfferNo = :NEW.OfferNo;
 END IF;
-- Decrease the number of enrolled students for delete, update
IF UPDATING OR DELETING THEN
    UPDATE Offering
      SET OffNumEnrolled = OffNumEnrolled - 1
      WHERE OfferNo = :OLD.OfferNo;
END IF;

EXCEPTION
  WHEN OTHERS THEN
    raise_application_error(-20001, 'Database error');
END;
```

Example 11.30

Script to Test the Triggers in Examples 11.28 and 11.29

```
-- Test case 1
-- See the offering limit and number enrolled
SELECT OffLimit, OffNumEnrolled
 FROM Offering
 WHERE Offering.OfferNo = 5679;
-- Insert the last student
INSERT INTO Enrollment (RegNo, OfferNo, EnrGrade)
 VALUES (1234,5679,0);
-- See the offering limit and the number enrolled
SELECT OffLimit, OffNumEnrolled
 FROM Offering
 WHERE Offering.OfferNo = 5679;

-- Test case 2
-- Insert a student beyond the limit: exception raised
INSERT INTO Enrollment (RegNo, OfferNo, EnrGrade)
 VALUES (1236,5679,0);
-- Transfer a student to offer 5679: exception raised
UPDATE Enrollment
 SET OfferNo = 5679
 WHERE RegNo = 1234 AND OfferNo = 1234;

-- Test case 3
-- See the offering limit and the number enrolled before update
SELECT OffLimit, OffNumEnrolled
 FROM Offering
 WHERE Offering.OfferNo = 4444;
-- Update a student to a non full offering
UPDATE Enrollment
 SET OfferNo = 4444
 WHERE RegNo = 1234 AND OfferNo = 1234;
-- See the offering limit and the number enrolled after update
SELECT OffLimit, OffNumEnrolled
 FROM Offering
 WHERE Offering.OfferNo = 4444;
```

```
-- Test case 4
-- See the offering limit and the number enrolled before delete
SELECT OffLimit, OffNumEnrolled
 FROM Offering
 WHERE Offering.OfferNo = 1234;
-- Delete an enrollment
DELETE Enrollment
 WHERE OfferNo = 1234;
-- See the offering limit and the number enrolled
SELECT OffLimit, OffNumEnrolled
 FROM Offering
 WHERE Offering.OfferNo = 1234;

-- Erase all changes
ROLLBACK;
```

Example 11.31

Trigger to Ensure That a Salary Increase Does Not Exceed 10 percent

Note that the NEW and OLD keywords should not be preceded by a colon (:) when used in a condition in the WHEN clause.

```
CREATE OR REPLACE TRIGGER tr_FacultySalary_UB
-- This trigger ensures that a salary increase does not exceed
-- 10%.
BEFORE UPDATE OF FacSalary
ON Faculty
FOR EACH ROW
WHEN (NEW.FacSalary > 1.1 * OLD.FacSalary)
DECLARE
     SalaryIncreaseTooHigh EXCEPTION;
     ExMessage VARCHAR(200);
BEGIN
    RAISE SalaryIncreaseTooHigh;
EXCEPTION
  WHEN SalaryIncreaseTooHigh THEN
   -- error number between -20000 and -20999
      ExMessage := 'Salary increase exceeds 10%. ';
      ExMessage := ExMessage || 'Current salary: ' ||
                to_char(:OLD.FacSalary) || '. ';
      ExMessage := ExMessage || 'New salary: ' ||
                to_char(:NEW.FacSalary) || '.';
      Raise_Application_Error(-20001, ExMessage);
END;
/
SET SERVEROUTPUT ON;
-- Test case 1: salary increase of 5%
UPDATE Faculty
 SET FacSalary = FacSalary * 1.05
 WHERE FacNo = '543-21-0987';
SELECT FacSalary FROM Faculty WHERE FacNo = '543-21-0987';
-- Test case 2: salary increase of 20% should generate an error.
UPDATE Faculty
 SET FacSalary = FacSalary * 1.20
 WHERE FacNo = '543-21-0987';
ROLLBACK;
```

Example 11.32

Trigger to Change the Case of a Faculty First and Last Name

```
CREATE OR REPLACE TRIGGER tr_FacultyName_IUB
-- This trigger changes the case of FacFirstName and FacLastName.
BEFORE INSERT OR UPDATE OF FacFirstName, FacLastName
ON Faculty
FOR EACH ROW
BEGIN
   :NEW.FacFirstName := UPPER(:NEW.FacFirstName);
   :NEW.FacLastName := UPPER(:NEW.FacLastName);
END;
/
-- Testing statements
UPDATE Faculty
 SET FacFirstName = 'Joe', FacLastName = 'Smith'
 WHERE FacNo = '543-21-0987';
-- Display the changed faculty name.
SELECT FacFirstName, FacLastName
 FROM Faculty
 WHERE FacNo = '543-21-0987';
ROLLBACK;
```

AFTER ROW Trigger for Exception Reporting The trigger in Example 11.31 implements a hard constraint in that large raises (greater than 10 percent) are rejected. A more flexible approach is a soft constraint in which a large raise causes a row to be written to an exception table. The update succeeds but an administrator can review the exception table at a later point to take additional action. A message can also be sent to alert the administrator to review specific rows in the exception table.

Example 11.33 depicts a trigger to implement a soft constraint for large employee raises. The AFTER trigger timing is used because a row should only be written to the exception table if the update succeeds. As demonstrated in Section 11.3.4, AFTER ROW triggers only execute if there are no errors encountered in integrity constraint checking.

Example 11.33

Trigger to Insert a Row into an Exception Table. When a salary increase exceeds 10 percent, the trigger fires. Since *LogTable* must be created before creating the trigger, the CREATE TABLE statement precedes the trigger. The SEQUENCE is an Oracle object that maintains unique values. The expression *LogSeq.NextVal* generates the next value of the sequence.

```
-- Create exception table and sequence
CREATE TABLE LogTable
(ExcNo              INTEGER         PRIMARY KEY,
 ExcTrigger  VARCHAR2(25) NOT NULL,
 ExcTable           VARCHAR2(25) NOT NULL,
 ExcKeyValue VARCHAR2(15) NOT NULL,
 ExcDate            DATE DEFAULT SYSDATE NOT NULL,
 ExcText            VARCHAR2(255) NOT NULL );

CREATE SEQUENCE LogSeq INCREMENT BY 1;

CREATE OR REPLACE TRIGGER tr_FacultySalary_UA
-- This trigger inserts a row into LogTable when
-- when a raise exceeds 10%.
```

```
AFTER UPDATE OF FacSalary
ON Faculty
FOR EACH ROW
WHEN (NEW.FacSalary > 1.1 * OLD.FacSalary)
DECLARE
     SalaryIncreaseTooHigh EXCEPTION;
     ExMessage VARCHAR(200);
BEGIN
    RAISE SalaryIncreaseTooHigh;
EXCEPTION
  WHEN SalaryIncreaseTooHigh THEN
     INSERT INTO LogTable
      (ExcNo, ExcTrigger, ExcTable, ExcKeyValue, ExcDate, ExcText)
       VALUES (LogSeq.NextVal, 'TR_ FacultySalary_UA', 'Faculty',
               to_char(:New.FacNo), SYSDATE,
                  'Salary raise greater than 10%');
END;
/
SET SERVEROUTPUT ON;
-- Test case 1: salary increase of 5%
UPDATE Faculty
 SET FacSalary = FacSalary * 1.05
 WHERE FacNo = '543-21-0987';
SELECT FacSalary FROM Faculty WHERE FacNo = '543-21-0987';
SELECT * FROM LogTable;

-- Test case 2: salary increase of 20% should generate an exception.
UPDATE Faculty
 SET FacSalary = FacSalary * 1.20
 WHERE FacNo = '543-21-0987';
SELECT FacSalary FROM Faculty WHERE FacNo = '543-21-0987';
SELECT * FROM LogTable;
ROLLBACK;
```

Trigger Formulation Guidelines Now that you have seen a variety of triggers, you are ready to write your own triggers. This brief subsection provides guidelines to help in writing triggers and tips to avoid common coding errors. You are encouraged to apply these guidelines to the trigger problems at the end of the chapter.

Formulating a trigger involves identifying trigger events and timing along with planning coding details. When coverage of multiple events is needed, a trigger developer should consider designs of a single trigger with a compound or multiple triggers with individual events. As mentioned previously, there is no clear preference for many smaller triggers or fewer larger triggers. Although smaller triggers are easier to understand than larger triggers, the number of triggers is a complicating factor to understand interactions among triggers. Trigger timing is usually clearer than designs for multiple events. BEFORE ROW triggers should not manipulate data except through the OLD and NEW keywords. Triggers that use SQL data manipulation statements should be written as AFTER ROW triggers.

Trigger coding involves manipulation of data in the target table and often related parent tables. The target table appears after the ON keyword. Columns of the target table should be accessed using the NEW and OLD keywords. Oracle triggers execute with a run-time error if the target table appears in the FROM clause of a SELECT statement inside a trigger[2]. Every trigger presented in this section uses the OLD or NEW keywords to access data in the target table.

As an aid to correct coding for the target and related parent tables in a trigger, you should study trigger patterns shown in Tables 11-6 and 11-7. The trigger pattern

[2] See the details in Section 11.3.4 about mutating table errors for an explanation about this restriction.

Trigger Pattern Template

```
-- Use for UPDATE events and sometimes INSERT events
SELECT <Parent columns>
 INTO <Trigger variables>
 FROM ParentTable
 WHERE ParentTable.PK = :NEW.FK
```

Trigger Pattern Example

```
-- Offering parent table and Enrollmment target table
-- See Example 11.28
SELECT OffLimit, OffNumEnrolled
   INTO anOffLimit, anOffNumEnrolled
   FROM Offering
   WHERE Offering.OfferNo = :NEW.OfferNo;
```

TABLE 11-6
Trigger Pattern for BEFORE ROW Trigger Using :NEW Values

for BEFORE ROW events uses :NEW values to retrieve a related row of a parent table. This trigger pattern typically applies to triggers performing integrity constraints on a column in a related row of a parent table. Related tables should be accessed using SELECT statements often using the INTO clause to retrieve column values in trigger variables. The only restriction on the INTO clause is that the SELECT statement must return at most one row. Typically, the foreign key value in the target table row is used to find the related row in the parent table. In Example 11.28, the *OfferNo* value of the target table (*Enrollment*) is used (*:NEW.OfferNo*) in a WHERE condition to retrieve the related row of the parent table (*Offering*). The INTO clause assigns the *OffLimit* and *OffNumEnrolled* column values to trigger variables.

The trigger pattern for AFTER ROW events uses :OLD values to retrieve a related row in a parent table. This trigger pattern typically applies to triggers updating a column in a related row of a parent table. Related tables should be accessed using a WHERE condition in an UPDATE statement on the parent table. Typically, the foreign key value in the target table row is used to find the related row in the parent table. In Example 11.29, the *OfferNo* value of the target table (*Enrollment*) is used (*:OLD.OfferNo*) to find the related row of the parent table (*Offering*). The SET clause modifies column values in the related parent row.

These coding patterns should help you avoid common coding errors as shown in Table 11-8. The coding patterns apply directly to the first two rows of Table 11-8. In the next two rows, the null value errors involve appropriate usage of the NEW and OLD keywords. You should understand the difference between INSERT and DELETE events for appropriate usage. The Too_Many_Rows exception is an uncommon error. To avoid syntax errors with the NEW and OLD keywords, you should prefix the NEW and OLD keywords with a colon (:) in the trigger body. In the WHEN clause, you just use the NEW and OLD keywords without the colon (:) prefix.

Trigger Pattern Template

```
-- Use for UPDATE events and sometimes DELETE event
UPDATE ParentTable
  SET <Assignments for parent columns>
  WHERE ParentTable.PK = :OLD.FK
```

Trigger Pattern Example

```
-- Offering parent table and Enrollmment target table
-- See Example 11.29
UPDATE Offering
  SET OffNumEnrolled = OffNumEnrolled - 1
  WHERE Offering.OfferNo = :OLD.OfferNo;
```

TABLE 11-7
Trigger Pattern for AFTER ROW Using :OLD Values

TABLE 11-8
Common Trigger Coding Errors

Error	Resolution
Target table columns appear in a SELECT clause	Use :NEW and :OLD keywords to access column values in the target table. Remove target table in FROM clause.
Join conditions using target table columns	Use :NEW and :OLD keywords for conditions to connect target table to related tables. Remove target table in FROM clause.
Null value for a :NEW column value	New values are undefined for DELETE events. Use old values instead.
Null value for an :OLD column value	Old values are undefined for INSERT events. Use new values instead.
Too_Many_Rows exception raised	Need cursor instead of SELECT ... INTO statement. Sometimes too many rows are returned because the new and/or old values were not used in WHERE conditions.
Syntax for OLD and NEW keywords	Use :NEW and :OLD in trigger body. Use NEW and OLD in WHEN clause.

11.3.3 Specialized Oracle Triggers using the INSTEAD OF Event

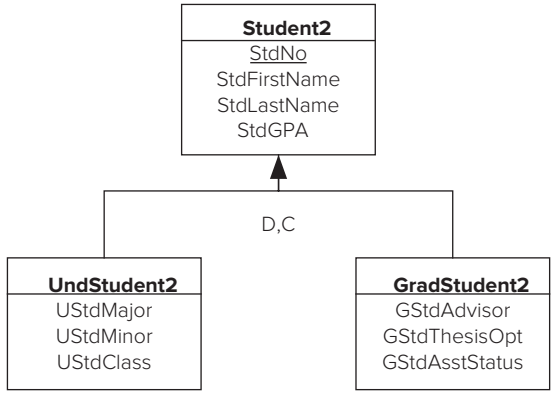
Oracle provides the INSTEAD OF event, a proprietary event especially useful in simulating generalization hierarchies and processing view updates. This section presents detailed examples to depict both trigger usages.

Triggers to Simulate Operations on a Generalization Hierarchy The traditional relational model does not directly support generalization hierarchies as presented in Chapters 5 and 6. Even though SQL:2016 has added support for generalization hierarchies among tables (see Chapter 19), most DBMSs including Oracle do not support this part of the SQL:2016 specification. Because generalization hierarchies are a specialized but useful feature, it is important to provide some level of support.

In Oracle, triggers can provide a level of support for generalization hierarchies using the INSTEAD OF event for views. An INSTEAD OF trigger is used in place of a manipulation event (INSERT, UPDATE, or DELETE) on a view. INSTEAD OF triggers can only be used with views, typically to support view updates. INSTEAD OF triggers are also useful for operations on tables related by generalization relationships as an INSTEAD OF trigger can map an operation to more than one base table.

Before presenting the trigger example, some additional tables and views are needed. The Generalization Hierarchy Rule (see Section 6.4.3) is applied to convert the student generalization hierarchy in Figure 11.2 into tables as shown in Example 11.34. The Generalization Hierarchy Rule generates one table per entity type with the primary key replicated in each table. Thus the subtype tables contain the primary key plus the specialized columns specific to the subtype table. Example 11.35 contains two

FIGURE 11.2
ERD for Student Generalization Hierarchy



views combining the parent and subtype tables. These views will be used in INSTEAD OF triggers to map manipulation actions on the views to the underlying base tables.

Example 11.34

CREATE TABLE Statements for the Student Generalization Hierarchy. The table names use the number 2 appended at the end to avoid conflicts with other table names

```
-- Student2 is the parent table in the generalization hierarchy
CREATE TABLE Student2(
 StdNo CHAR(11),
 StdFirstName VARCHAR2(20) not null,
 StdLastName VARCHAR2(30) not null,
 StdGPA DECIMAL(3,2),
 CONSTRAINT Student2PK PRIMARY KEY (StdNo) );
-- UndStudent2 is a child table in the generalization hierarchy
CREATE TABLE UndStudent2(
 StdNo CHAR(11),
 UStdMajor CHAR(6),
 UStdMinor CHAR(6),
 UStdClass CHAR(2),
 CONSTRAINT UndStudent2PK PRIMARY KEY (StdNo),
 CONSTRAINT UndStudent2FK FOREIGN KEY(StdNo) REFERENCES Student2
   ON DELETE CASCADE );
-- GradStudent2 is a child table in the generalization hierarchy
CREATE TABLE GradStudent2(
 StdNo CHAR(11),
 GStdAdvisor VARCHAR2(20),
 GStdThesisOpt CHAR(10) DEFAULT 'NONTHESIS', -- NONTHESIS or THESIS
 GStdAsstStatus CHAR(6) DEFAULT 'NONE', -- NONE, TA, RA
 CONSTRAINT GradStudent2PK PRIMARY KEY (StdNo),
 CONSTRAINT GradStudent2FK FOREIGN KEY(StdNo) REFERENCES Student2
   ON DELETE CASCADE );
```

Example 11.35

CREATE VIEW Statements for Views Used to Manipulate the Student Generalization Hierarchy

```
-- View for undergraduate students
CREATE VIEW AllUndStudent AS
  SELECT Student2.StdNo, StdFirstName, StdLastName, StdGPA,
         UStdMajor, UStdMinor, UStdClass
    FROM Student2, UndStudent2
    WHERE Student2.StdNo = UndStudent2.Stdno;
-- View for graduate students
CREATE VIEW AllGradStudent AS
  SELECT Student2.StdNo, StdFirstName, StdLastName, StdGPA,
         GStdAdvisor, GStdThesisOpt, GStdAsstStatus
    FROM Student2, GradStudent2
    WHERE Student2.StdNo = GradStudent2.Stdno;
```

To insert a row into a child table in a generalization hierarchy, a row should be added to the parent table. Example 11.36 contains INSTEAD OF triggers that insert rows in the child and parent tables. For example, the tr_UndStudent_II trigger inserts a row into the *UndStudent2* and *Student2* tables. The testing code in Example 11.37 contains INSERT statements for each view. The INSTEAD OF trigger executes in place of the INSERT statements. Note that INSTEAD OF triggers are proprietary to Oracle.

Example 11.36

Triggers to Insert Rows in Child and Parent Tables of the Student Generalization Hierarchy

```
-- Insert trigger for undergraduate students
CREATE OR REPLACE TRIGGER tr_AllUndStudent_II
INSTEAD OF INSERT ON AllUndStudent
FOR EACH ROW
BEGIN
-- Insert into parent (Student2)
INSERT INTO Student2 (StdNo, StdFirstName, StdLastName,StdGPA)
 VALUES(:New.StdNo,:New.StdFirstName,:New.StdLastName,:New.StdGPA);
-- Insert into child (UndStudent2)
INSERT INTO UndStudent2 (StdNo, UStdMajor, UStdMinor, UStdClass)
 VALUES(:New.StdNo, :New.UStdMajor, :New.UStdMinor, :New.UStdClass);
EXCEPTION
 WHEN OTHERS THEN
  raise_application_error(-20001, 'DB error in tr_UndStudent_II');
END;
-- Insert trigger for graduate students
CREATE OR REPLACE TRIGGER tr_AllGradStudent_II
INSTEAD OF INSERT ON AllGradStudent
FOR EACH ROW
BEGIN
-- Insert into parent (Student2)
INSERT INTO Student2 (StdNo, StdFirstName, StdLastName,StdGPA)
 VALUES(:New.StdNo,:New.StdFirstName,:New.StdLastName,:New.StdGPA);
-- Insert into child (UndStudent2)
INSERT INTO GradStudent2(StdNo,GStdAdvisor,GStdThesisOpt, GStdAsstStatus)
VALUES(:New.StdNo,:New.GStdAdvisor,:New.GStdThesisOpt, :New.GStdAsstStatus);
EXCEPTION
 WHEN OTHERS THEN
  raise_application_error(-20001, 'DB error in tr_GradStudent_II');
END;
```

Example 11.37

Testing Code for INSTEAD OF INSERT Triggers

```
-- Test cases for tr_UndStudent_II
INSERT INTO AllUndStudent
 (StdNo,StdFirstName,StdLastName,StdGPA,UStdMajor,UStdMinor,
  UStdClass)
 VALUES('123-45-6789','HOMER','WELLS',3.00,'IS','ACCT','FR');
```

```
INSERT INTO AllUndStudent
 (StdNo, StdFirstName, StdLastName, StdGPA, UStdMajor, UStdMinor,
  UStdClass)
 VALUES('234-56-7890','CANDY','KENDALL',2.70,'FIN','IS','JR');
-- SELECT statements to verify insertions
SELECT * FROM AllUndStudent;
SELECT * FROM Student2;
SELECT * FROM UndStudent2
-- Test cases for tr_GradStudent_II
INSERT INTO AllGradStudent
 (StdNo, StdFirstName, StdLastName, StdGPA, GStdAdvisor,
  GStdThesisOpt, GStdAsstStatus)
 VALUES ('345-67-8901','WALLY','KENDALL', 2.80, 'Jones', 'NONTHESIS',
          'NONE');
INSERT INTO AllGradStudent
 (StdNo, StdFirstName, StdLastName, StdGPA, GStdAdvisor,
  GStdThesisOpt, GStdAsstStatus)
 VALUES ('456-78-9012','JOE','ESTRADA', 3.20,'Jones','THESIS','RA');
-- SELECT statements to verify insertions
SELECT * FROM AllGradStudent;
SELECT * FROM Student2;
SELECT * FROM GradStudent2;
ROLLBACK;
```

The student generalization hierarchy has a disjointness constraint so that a student cannot simultaneously be both an undergraduate and graduate student. Example 11.38 contains a modified trigger to check for membership in the other child table of the generalization hierarchy. In the original trigger in Example 11.36, inserting the same row in the other child (*GradStudent2*) using the *AllGradStudent* view will fail because a row already exists in the parent table (*Student2*).

Example 11.38

Modification of Trigger in Example 11.36

Ensure that an Undergraduate Student is not in the GradStudent Table along with Testing Code.

```
-- Extended INSTEAD OF trigger to enforce disjointness constraint
CREATE OR REPLACE TRIGGER tr_UndStudent_II
INSTEAD OF INSERT ON AllUndStudent
FOR EACH ROW
DECLARE
    GradStdExists EXCEPTION;
    ExMessage VARCHAR(200);
    GrdStdCnt INTEGER;
BEGIN
SELECT COUNT(*) INTO GrdStdCnt FROM GradStudent2
  WHERE GradStudent2.StdNo = :New.StdNo;
IF GrdStdCnt > 0 THEN
    RAISE GradStdExists;
END IF;
-- Insert into parent (Student2)
INSERT INTO Student2 (StdNo, StdFirstName, StdLastName,StdGPA)
  VALUES(:New.StdNo,:New.StdFirstName,:New.StdLastName,:New.StdGPA);
-- Insert into child (UndStudent2)
INSERT INTO UndStudent2 (StdNo, UStdMajor, UStdMinor, UStdClass)
  VALUES(:New.StdNo, :New.UStdMajor, :New.UStdMinor, :New.UStdClass);
```

```
EXCEPTION
  WHEN GradStdExists THEN
   -- error number between -20000 and -20999
     ExMessage := 'Graduate student already exists. ';
     ExMessage := ExMessage || 'StdNo: ' || :New.StdNo;
     Raise_Application_Error(-20001, ExMessage);
  WHEN OTHERS THEN
    raise_application_error(-20001, 'DB error in tr_UndStudent_II');
END;
/
-- Trigger testing statements
-- Test cases for tr_UndStudent_II: should succeed
INSERT INTO AllUndStudent
 (StdNo, StdFirstName, StdLastName, StdGPA, UStdMajor, UStdMinor,
  UStdClass)
 VALUES ('123-45-6789','HOMER','WELLS',3.00,'IS','ACCT','FR');
INSERT INTO AllUndStudent
 (StdNo, StdFirstName, StdLastName, StdGPA, UStdMajor, UStdMinor,
  UStdClass)
 VALUES ('234-56-7890','CANDY','KENDALL',2.70,'FIN','IS','JR');
-- Test cases for tr_GradStudent_IA: should succeed
INSERT INTO AllGradStudent
 (StdNo, StdFirstName, StdLastName, StdGPA, GStdAdvisor,
  GStdThesisOpt, GStdAsstStatus)
 VALUES('345-67-8901','WALLY','KENDALL', 2.80, 'Jones', 'NONTHESIS',
        'NONE');
INSERT INTO AllGradStudent
 (StdNo, StdFirstName, StdLastName, StdGPA, GStdAdvisor,
  GStdThesisOpt, GStdAsstStatus)
 VALUES ('456-78-9012','JOE','ESTRADA', 3.20,'Jones','THESIS','RA');
-- Test cases should fail because grad student exists
INSERT INTO AllUndStudent
(StdNo, StdFirstName, StdLastName, StdGPA, UStdMajor, UStdMinor,
 UStdClass)
 VALUES ('345-67-8901','WALLY','KENDALL', 3.00,'IS','ACCT','FR');
ROLLBACK;
```

Another trigger is needed to manage updates. Updates to a view should be directed to the appropriate base table. Example 11.39 contains an INSTEAD OF UPDATE trigger to direct updates to the appropriate table (*Student2* or *UndStudent2*). A similar trigger is necessary to direct graduate student updates to the *Student2* or *GradStudent2* table. Note that Oracle does not allow column specification in INSTEAD OF UPDATE triggers. In BEFORE and AFTER triggers, it is good coding practice for UPDATE triggers to specify the column.

Example 11.39

INSTEAD OF UPDATE Trigger to Update either *Student2* or *UndStudent2* Table along with Testing Code

```
-- Update trigger for UndStudent
-- Cannot specify column list for INSTEAD of UPDATE triggers
CREATE OR REPLACE TRIGGER tr_AllUndStudent_UI
INSTEAD OF UPDATE ON AllUndStudent
FOR EACH ROW
BEGIN
```

```
-- Update UndStudent2
  IF UPDATING('UStdMajor') THEN
    UPDATE UndStudent2
    SET UStdMajor = :NEW.UStdMajor
    WHERE StdNo = :OLD.StdNo;
  END IF;
  IF UPDATING('UStdMinor') THEN
    UPDATE UndStudent2
    SET UStdMinor = :NEW.UStdMinor
    WHERE StdNo = :OLD.StdNo;
  END IF;
  IF UPDATING('UStdClass') THEN
    UPDATE UndStudent2
    SET UStdClass = :NEW.UStdClass
    WHERE StdNo = :OLD.StdNo;
  END IF;
-- Update Student2
  IF UPDATING('StdGPA') THEN
    UPDATE Student2
    SET StdGPA = :NEW.StdGPA
    WHERE StdNo = :OLD.StdNo;
  END IF;
  IF UPDATING('StdFirstName') THEN
    UPDATE Student2
    SET StdFirstName = :NEW.StdFirstName
    WHERE StdNo = :OLD.StdNo;
  END IF;
  IF UPDATING('StdLastName') THEN
    UPDATE Student2
    SET StdLastName = :NEW.StdLastName
    WHERE StdNo = :OLD.StdNo;
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    raise_application_error(-20001, 'DB error in tr_UStdMajor_UI');
END;
/
-- Trigger testing statements
-- Insert data: depends on tr_UndStudent_II
INSERT INTO AllUndStudent
 (StdNo, StdFirstName, StdLastName, StdGPA, UStdMajor, UStdMinor,
  UStdClass)
 VALUES ('123-45-6789','HOMER','WELLS',3.00,'IS','ACCT','FR');
INSERT INTO AllUndStudent
 (StdNo, StdFirstName, StdLastName, StdGPA, UStdMajor, UStdMinor,
  UStdClass)
 VALUES ('234-56-7890','CANDY','KENDALL',2.70,'FIN','IS','JR');
-- Update statements
UPDATE AllUndStudent
 SET UStdMajor = 'MGMT'
 WHERE StdNo = '123-45-6789';
UPDATE AllUndStudent
 SET StdGPA = 3.1
 WHERE StdNo = '234-56-7890';
-- View results
SELECT * FROM AllUndStudent;
SELECT * FROM Student2;
SELECT * FROM UndStudent2;
ROLLBACK;
```

Triggers for Multiple Table View Updates Chapter 10 (Appendix 10.B) presented rules for updatable join views in Oracle. Updatable join views are more restrictive than Microsoft Access 1-M updatable queries on the supported modification operations.

Oracle restricts modification operations to one underlying table (known as the key preserving table) in modification statements on views. This subsection presents INSTEAD OF trigger examples to extend the range of modification operations on multiple table views in Oracle.

In a trigger supporting insert operations on both parent and child tables (the child table is typically the key preserving table), you should check for existence of the parent row. If the parent row does not exist, the insert operation should be mapped to both parent and child tables. Otherwise, the insert operation only maps to the child table. In Example 11.40, *Course* is the parent table and *Offering* is the child table in the updatable join view. The INSTEAD OF INSERT trigger in Example 11.41 maps an insert operation on the view to both tables or just the *Offering* table.

Example 11.40 (Oracle)

Updatable Join View

Updatable Join View Combining the *Course* and the *Offering* Tables.

```
CREATE VIEW CourseOfferingView AS
  SELECT Course.CourseNo, CrsDesc, CrsUnits,
         OfferNo, OffTerm, OffYear,
         OffLocation, OffTime, FacNo, OffDays,
         OffLimit, OffNumEnrolled
    FROM Course INNER JOIN Offering
        ON Course.CourseNo = Offering.CourseNo
```

Example 11.41

INSTEAD OF INSERT Trigger to Map a View Insertion to the Underlying Tables (*Course* and *Offering*) along with Testing Code

```
CREATE OR REPLACE TRIGGER tr_CourseOfferingView_II
INSTEAD OF INSERT ON CourseOfferingView
FOR EACH ROW
DECLARE
  CourseCnt INTEGER;
BEGIN
SELECT COUNT(*) INTO CourseCnt FROM Course
  WHERE CourseNo = :NEW.CourseNo;
IF CourseCnt = 0 THEN
    -- INSERT into Course table
    INSERT INTO Course (CourseNo, CrsDesc, CrsUnits)
    VALUES(:NEW.CourseNo, :NEW.CrsDesc, :NEW.CrsUnits);
END IF;
-- INSERT into Offering table
INSERT INTO Offering (OfferNo, CourseNo, OffTerm, OffYear,
            OffLocation, OffTime, FacNo, OffDays, OffLimit,
            OffNumEnrolled)
VALUES(:NEW.OfferNo, :NEW.CourseNo, :NEW.OffTerm, :NEW.OffYear,
       :NEW.OffLocation, :NEW.OffTime, :NEW.FacNo, :NEW.OffDays,
       :NEW.OffLimit, :NEW.OffNumEnrolled);
```

```
EXCEPTION
  WHEN OTHERS THEN
   raise_application_error(-20001,
      'DB error in tr_CourseOfferingView_II');
END;
/
-- Trigger testing statements
-- Insert only an Offering as Course exists
INSERT INTO CourseOfferingView
 (OfferNo, CourseNo, OffTerm, OffYear, OffLocation, OffTime,
  FacNo, OffDays, OffLimit, OffNumEnrolled)
 VALUES (9999,'IS320','SUMMER',2017,'BLM402','9:00:00', NULL,'MW',
         10, 0);
-- Ensure that a row has been added to the Offering table
SELECT * FROM Offering WHERE OfferNo = 9999;
-- Insert into both Offering and Course as Course does not exist
INSERT INTO CourseOfferingView
 (CourseNo, CrsDesc, CrsUnits, OfferNo, OffTerm, OffYear,
  OffLocation, OffTime, FacNo, OffDays, OffLimit, OffNumEnrolled)
 VALUES ('IS321', 'IT Security', 3, 9009, 'SUMMER', 2017,
         'BLM412','9:00:00', NULL,'TTH', 10, 0);
-- Ensure that a row has been added to both tables
SELECT * FROM Course WHERE CourseNo = 'IS321';
SELECT * FROM Offering WHERE OfferNo = 9009;
ROLLBACK;
```

In a trigger supporting update operations on columns from both parent and child tables, you should use a similar approach to the trigger in Example 11.39. Updates to a view should be directed to the appropriate base table. Example 11.42 contains an INSTEAD OF UPDATE trigger to direct updates to the appropriate table (*Course* or *Offering*).

Example 11.42

INSTEAD OF UPDATE Trigger to Update Columns from the *Course* and *Offering* Tables along with Testing Code

```
-- Update trigger for CourseOfferingView
-- Does not support updates to the primary keys of the tables
CREATE OR REPLACE TRIGGER tr_CourseOfferingView_UI
INSTEAD OF UPDATE ON CourseOfferingView
FOR EACH ROW
BEGIN
-- Update Course columns
  IF UPDATING('CrsDesc') THEN
    UPDATE Course
    SET CrsDesc = :NEW.CrsDesc
    WHERE CourseNo = :OLD.CourseNo;
  END IF;
  IF UPDATING('CrsUnits') THEN
    UPDATE Course
    SET CrsUnits = :NEW.CrsUnits
    WHERE CourseNo = :OLD.CourseNo;
  END IF;
```

```
-- Update Offering columns
  IF UPDATING('OffTerm') THEN
    UPDATE Offering
    SET OffTerm = :NEW.OffTerm
    WHERE OfferNo = :OLD.OfferNo;
  END IF;
  IF UPDATING('OffYear') THEN
    UPDATE Offering
    SET OffYear = :NEW.OffYear
    WHERE OfferNo = :OLD.OfferNo;
  END IF;
  IF UPDATING('OffLocation') THEN
    UPDATE Offering
    SET OffLocation = :NEW.OffLocation
    WHERE OfferNo = :OLD.OfferNo;
  END IF;
  IF UPDATING('OffDays') THEN
    UPDATE Offering
    SET OffDays = :NEW.OffDays
    WHERE OfferNo = :OLD.OfferNo;
  END IF;
  IF UPDATING('OffTime') THEN
    UPDATE Offering
    SET OffTime = :NEW.OffTime
    WHERE OfferNo = :OLD.OfferNo;
  END IF;
  IF UPDATING('FacNo') THEN
    UPDATE Offering
    SET FacNo = :NEW.FacNo
    WHERE OfferNo = :OLD.OfferNo;
  END IF;
  IF UPDATING('OffLimit') THEN
    UPDATE Offering
    SET OffLimit = :NEW.OffLimit
    WHERE OfferNo = :OLD.OfferNo;
  END IF;
  IF UPDATING('OffNumEnrolled') THEN
    UPDATE Offering
    SET OffNumEnrolled = :NEW.OffNumEnrolled
    WHERE OfferNo = :OLD.OfferNo;
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    raise_application_error(-20001,
      'DB error in tr_CourseOfferingView_UI');
END;
/
-- Trigger testing statements
-- Insert rows for test
INSERT INTO Course
 (CourseNo, CrsDesc, CrsUnits)
 VALUES ('IS321', 'IT Security', 3);
INSERT INTO Offering
 (OfferNo, CourseNo, OffTerm, OffYear,
  OffLocation, OffTime, FacNo, OffDays, OffLimit, OffNumEnrolled)
 VALUES (9009, 'IS321', 'SUMMER', 2017,
         'BLM412','9:00:00', NULL,'TTH', 10, 0);
-- Update statements
-- Course columns
UPDATE CourseOfferingView
 SET CrsDesc = 'IT Security II'
 WHERE CourseNo = 'IS321';
UPDATE CourseOfferingView
 SET CrsUnits = 4
 WHERE CourseNo = 'IS321';
```

```
-- Offering columns
UPDATE CourseOfferingView
 SET OffTerm = 'Fall'
 WHERE OfferNo = 9009;
UPDATE CourseOfferingView
 SET OffYear = 2014
 WHERE OfferNo = 9009;
UPDATE CourseOfferingView
 SET OffDays = 'MW'
 WHERE OfferNo = 9009;
UPDATE CourseOfferingView
 SET OffLocation = 'BLM305'
 WHERE OfferNo = 9009;
UPDATE CourseOfferingView
 SET OffTime = '10:30:00'
 WHERE OfferNo = 9009;
UPDATE CourseOfferingView
 SET FacNo = '543-21-0987'
 WHERE OfferNo = 9009;
UPDATE CourseOfferingView
 SET OffLimit = OffLimit +1
 WHERE OfferNo = 9009;
UPDATE CourseOfferingView
 SET OffNumEnrolled = OffNumEnrolled +1
 WHERE OfferNo = 9009;
-- View results
SELECT * FROM Course WHERE CourseNo = 'IS321';
SELECT * FROM Offering WHERE OfferNo = 9009;
ROLLBACK;
```

11.3.4 Understanding Trigger Execution

As the previous subsection demonstrated, individual triggers are usually easy to understand. Collectively, however, triggers can be difficult to understand especially in conjunction with integrity constraint enforcement and database actions. To understand the collective behavior of triggers, integrity constraints, and database manipulation actions, you need to understand the execution procedure used by a DBMS. Although SQL:2016 specifies a **trigger execution procedure**, most DBMSs do not adhere strictly to it. Therefore, this subsection emphasizes the Oracle trigger execution procedure with comments about the differences between the Oracle and the SQL:2016 execution procedures.

Trigger Execution Procedure
specifies the order of execution among the various kinds of triggers, integrity constraints, and database manipulation statements. Trigger execution procedures can be complex because the actions of a trigger may fire other triggers.

Simplified Trigger Execution Procedure The trigger execution procedure applies to data manipulation statements (INSERT, UPDATE, and DELETE). Before this procedure begins, Oracle determines the applicable triggers for an SQL statement. A trigger is applicable to a statement if the trigger contains an event that matches the statement type. To match an UPDATE statement with a column list, at least one column in the triggering event must be in the list of columns updated by the statement. After determining the applicable triggers, Oracle executes triggers in the order of BEFORE STATEMENT, BEFORE ROW, AFTER ROW, and AFTER STATEMENT. An applicable trigger does not execute if the WHEN condition is not true.

Overlapping Triggers
two or more triggers with the same timing, granularity, and target table. The triggers overlap if an SQL statement may cause both triggers to fire. You should not depend on a particular firing order for overlapping triggers.

The trigger execution procedure of Oracle differs slightly from the SQL:2016 execution procedure for **overlapping triggers**. Two triggers with the same timing, granularity, and target table overlap if an SQL statement may cause both triggers to fire. For example, a BEFORE ROW trigger with the UPDATE ON Customer event overlaps with a BEFORE ROW trigger with the UPDATE OF CustBal ON Customer event. Both triggers fire when updating the *CustBal* column. For overlapping triggers, Oracle provides arbitrary execution order by default. For SQL:2016, the execution order depends

on the time in which the trigger is defined. Overlapping triggers are executed in the order in which the triggers were created.

Simplified Oracle Trigger Execution Procedure

1. Execute the applicable BEFORE STATEMENT triggers.
2. For each row affected by the SQL manipulation statement:
   2.1. Execute the applicable BEFORE ROW triggers.
   2.2. Perform the data manipulation operation on the row.
   2.3. Perform integrity constraint checking.
   2.4. Execute the applicable AFTER ROW triggers.
3. Perform deferred integrity constraint checking.
4. Execute the applicable AFTER statement triggers.

Trigger overlap can be subtle for UPDATE triggers. Two UPDATE triggers on the same table can overlap even if the triggers involve different columns. For example, UPDATE triggers on *OffLocation* and *OffTime* overlap if an UPDATE statement changes both columns. For UPDATE statements changing only one column, the triggers do not overlap.

Oracle 12c provides the FOLLOWS clause to specify firing order among overlapping triggers. The FOLLOWS clause guarantees the firing order of overlapping triggers. This feature should be used sparingly because it requires knowledge of more than one trigger to understand the impact of overlapping triggers.

As demonstrated in the Simple Trigger Execution Procedure, most constraint checking occurs after executing the applicable BEFORE ROW triggers but before executing the applicable AFTER ROW triggers. Deferred constraint checking is performed at the end of a transaction. Chapter 17 on transaction management presents SQL statements for deferred constraint checking. In most applications, few constraints are declared with deferred checking.

Trigger Execution Procedure with Recursive Execution Data manipulation statements in a trigger complicate the simplified execution procedure. Data manipulation statements in a trigger may cause other triggers to fire. Consider the AFTER ROW trigger in Example 11.29 that fires when an *Enrollment* row is added. The trigger updates the *OffNumEnrolled* column enrolled in the related *Offering* row. Suppose there is another trigger on the *OffNumEnrolled* column of the *Offering* table that fires when the *OffNumEnrolled* column becomes large (say within two of the limit). This second trigger should fire as a result of the first trigger firing when an offering becomes almost full.

When a data manipulation statement is encountered in a trigger, the trigger execution procedure is recursively executed. Recursive execution means that a procedure calls itself. In the previous example, the trigger execution procedure is recursively executed when a data manipulation statement is encountered in the trigger in Example 11.29. In the Oracle execution procedure, steps 2.1 and 2.4 may involve recursive execution of the procedure. In the SQL:2016 execution procedure, only step 2.4 may involve recursive execution because SQL:2016 prohibits data manipulation statements in BEFORE ROW triggers.

Actions on referenced rows also complicate the simplified execution procedure. When deleting or updating a referenced row, the foreign key constraint can specify actions (CASCADE, SET NULL, and SET DEFAULT) on related rows. For example, a foreign key constraint containing ON DELETE CASCADE for *Offering.CourseNo* means that deletion of a *Course* row causes deletion of the related *Offering* rows. Actions on referenced rows can cause other triggers to fire leading to recursive execution of the trigger execution procedure in step 2.3 for both Oracle and SQL:2016. Actions on referenced rows are performed as part of constraint checking in step 2.3.

With these complications that cause recursive execution, the full trigger execution procedure is presented below. Most DBMSs such as Oracle limit the recursion depth in steps 2.1, 2.3, and 2.4.

Oracle Trigger Execution Procedure with Recursive Execution

1. Execute the applicable BEFORE STATEMENT triggers.
2. For each row affected by the SQL manipulation statement
   - 2.1. Execute the applicable BEFORE ROW triggers. Recursively execute the procedure for data manipulation statements in a trigger.
   - 2.2. Perform the data manipulation operation on the row.
   - 2.3. Perform integrity constraint checking. Recursively execute the procedure for actions on referenced rows.
   - 2.4. Execute the applicable AFTER ROW triggers. Recursively execute the procedure for data manipulation statements in a trigger.
3. Perform deferred integrity constraint checking.
4. Execute the applicable AFTER statement triggers.

The full execution procedure shows considerable complexity when executing a trigger. To control complexity among a collection of triggers, you should follow these guidelines:

- Define triggers for general purpose business rules, not rules specific to a given application.
- Avoid data manipulation statements in BEFORE triggers.
- Limit data manipulation statements in AFTER triggers to statements that are likely to succeed.
- For triggers that fire on UPDATE statements, always list the columns in which the trigger applies.
- Ensure that overlapping triggers do not depend on a specific order to fire. In most DBMSs, the firing order is arbitrary. Even if the order is not arbitrary (as in SQL:2016), it is risky to depend on a specific firing order.
- Be cautious about triggers on tables affected by actions on referenced rows. These triggers will fire as a result of actions on the parent tables.

Mutating Table Errors Oracle has a restriction on trigger execution that can impede the development of specialized triggers. In trigger actions, Oracle prohibits SQL statements on the table in which the trigger is defined or on related tables affected by DELETE CASCADE actions. The underlying trigger target table and the related tables are known as mutating tables. For example in a trigger on the *Registration* table, Oracle prohibits SQL statements on the *Registration* table as well as on the *Enrollment* table if the *Enrollment* table contains a foreign key constraint on *Enrollment.RegNo* with the ON DELETE CASCADE action. If a trigger executes an SQL statement on a mutating table, a run-time error occurs.

Oracle has the mutating table restriction to provide a consistent snapshot of data available in a trigger. Executing a SELECT statement on a mutating table would show a trigger a possibly inconsistent snapshot of a table. If the statement executed with the data existing before trigger execution, the new data does not appear. If the statement executed with the data existing after execution of the underlying SQL statement, a failure would remove the new data. Chapter 17 presents concurrency control principles to clarify this notion of consistency.

For most triggers, you can avoid mutating table errors by using row triggers with new and old values. In specialized situations, you must redesign a trigger to avoid a mutating table error. One situation involves a trigger to enforce an integrity constraint involving other rows of the same table. For example, a trigger to ensure that no more

than five rows contain the same value for a column would have a mutating table error. Another example would be a trigger that ensures that a row cannot be deleted if it is the last row associated with a parent table. A second situation involves a trigger for a parent table that inserts rows into a child table if the child table has a foreign key constraint with ON DELETE CASCADE.

To write triggers in these situations, you will need a more complex solution. For complete details, you should consult some websites that show solutions to avoid mutating table errors. The Oracle documentation mentions the following four approaches:

1. In simple situations, you can put the SELECT statement on the trigger target table in a procedure or function and then call the procedure or function in the trigger. The procedure or function can execute a query on the trigger table without the mutation restriction. This solution will not work for triggers on child tables having a referential integrity constraint with ON DELETE CASCADE.
2. In more complex situations, you may need to write a package and a collection of triggers that use procedures in the package. The package maintains a private array that contains the old and new values of the mutating table. Typically, you will need a BEFORE STATEMENT trigger to initialize the private array, an AFTER ROW trigger to insert into the private array, and an AFTER STATEMENT trigger to enforce the integrity constraint using the private array.
3. Create a view and use an INSTEAD OF trigger for the view. View triggers do not have any mutating table restrictions.
4. Write a compound trigger, a new feature in Oracle 11g. A compound trigger can have multiple timings, BEFORE STATEMENT, BEFORE ROW, AFTER ROW, and AFTER STATEMENT. A compound trigger supports accumulation of facts from row changes and then uses the collection of row changes at after statement time. In this manner, a compound trigger provides an alternative to a package and collection of triggers.

To depict a relatively simple situation involving a mutation restriction, Example 11.43 uses the first method to avoid a mutating table error. The *tr_Registration_IB* trigger implements an integrity constraint preventing a student from registering more than one time. The trigger calls a function to determine if a registration exists with the same student number, term, and year. The function uses a SELECT statement on the target table (*Registration*) without causing a mutating table error. The SELECT statement in the trigger body would have caused a mutating table error.

Example 11.43

Trigger to determine if a student is already registered. The trigger calls the function defined before the trigger

```
CREATE OR REPLACE FUNCTION fn_RegExists
(aStdNo IN Student.StdNo%TYPE, aRegTerm IN Registration.RegTerm%TYPE,
 aRegYear IN Registration.RegYear%TYPE)
RETURN BOOLEAN IS
-- Returns true if a Registration row exists with aStdNo,
-- aRegTerm, and aRegYear. Returns false otherwise.
RegCount INTEGER;
BEGIN
SELECT COUNT(*)
 INTO RegCount
 FROM Registration
 WHERE StdNo = aStdNo AND RegTerm = aRegTerm AND RegYear = aRegYear;
```

```
IF RegCount = 0 THEN
  RETURN(FALSE);
ELSE
  RETURN(TRUE);
END IF;
END;
/

CREATE OR REPLACE TRIGGER tr_Registration_IB
-- This trigger raises an error if the student is already registered
-- by determining if a row with the same StdNo, RegTerm, and RegYear
-- exists in the Registration table.
BEFORE INSERT
ON Registration
FOR EACH ROW
DECLARE
   ExMessage VARCHAR(256);
BEGIN
   ExMessage := 'Registration exists: Error in tr_Registration_IB';
   IF fn_RegExists(:NEW.StdNo, :NEW.RegTerm, :NEW.RegYear) THEN
     raise_application_error(-20001, ExMessage);
   END IF;
END;
/

-- Testing statements
-- Insert a new student
INSERT INTO Student
     (StdNo, StdFirstName, StdLastName, StdCity,
      StdState, StdMajor, StdClass, StdGPA, StdZip)
     VALUES ('999-99-9999', 'JOE', 'JONES', 'DENVER','CO', 'IS',
             'SO',3.00,'80217-3364');
-- Insert a registration for the new student without a failure.
INSERT INTO Registration
     (RegNo,StdNo,RegStatus,RegDate,RegTerm,RegYear)
     VALUES (1301,'999-99-9999','F','27-Feb-2017','Spring',2017);
-- Insert another registration in a different term without a failure.
INSERT INTO Registration
     (RegNo,StdNo,RegStatus,RegDate,RegTerm,RegYear)
     VALUES (1302,'999-99-9999','F','27-Apr-2017','Fall',2017);
-- Insert a third registration in the same term with a failure.
INSERT INTO Registration
     (RegNo,StdNo,RegStatus,RegDate,RegTerm,RegYear)
     VALUES (1303,'999-99-9999','F','27-Apr-2017','Fall',2017);
ROLLBACK;
```

CLOSING THOUGHTS

This chapter has augmented your knowledge of database application development with details about database programming languages, stored procedures, and triggers. Database programming languages are procedural languages with an interface to one or more DBMSs. Database programming languages support customization, batch processing, and data intensive web applications as well as improved efficiency and portability in some cases. The major design issues in a database programming language are language style, binding, database connections, and result processing. This chapter presented background about PL/SQL, a widely used database programming language available as part of Oracle.

After learning about database programming languages and PL/SQL, the chapter presented stored procedures. Stored procedures provide modularity like programming

language procedures. Stored procedures managed by a DBMS provide additional benefits including reuse of access plans, dependency management, and security control by the DBMS. You learned about PL/SQL procedure coding through examples demonstrating procedures, functions, exception processing, and embedded SQL containing single row results and multiple row results with cursors. You also learned about PL/SQL packages that group related procedures, functions, and other PL/SQL objects.

The final part of the chapter covered triggers for business rule processing. A trigger involves an event, a condition, and a sequence of actions. You learned the varied uses for triggers as well as a classification of triggers by granularity, timing, and applicable event. After this background material, you learned about coding Oracle triggers using PL/SQL statements in a trigger body. To provide understanding about the complexity of large collections of triggers, you learned about trigger execution procedures specifying the order of execution among various kinds of triggers, integrity constraints, and SQL statements.

The material in this chapter is important for both application developers and database administrators. Stored procedures and triggers can be a significant part of large applications, perhaps as much as 25 percent of the code. Application developers use database programming languages to code stored procedures and triggers, while database administrators provide oversight in the development process. In addition, database administrators may write stored procedures and triggers to support the process of database monitoring. Thus, database programming languages, stored procedures, and triggers are important tools for careers in both application development and database administration.

REVIEW CONCEPTS

- Primary motivation for database programming languages: customization, batch processing, and data intensive web applications
- Secondary motivation for database programming languages: efficiency and portability
- Statement-level interface to support embedded SQL in a programming language
- Call-level interface to provide procedures to invoke SQL statements in a programming language
- Popularity of proprietary call-level interfaces (ODBC and JDBC) instead of the SQL:2016 call-level interface
- Support for static and dynamic binding of SQL statements in statement-level interfaces
- Support for dynamic binding with access plan reuse for repetitive executions in call-level interfaces
- Implicit versus explicit database connections
- Usage of cursors to integrate set-at-a-time processing of SQL with record-at-a-time processing of programming languages
- PL/SQL data types and variable declaration
- Anchored variable declaration in PL/SQL
- Conditional statements in PL/SQL: IF-THEN, IF-THEN-ELSE, IF-THEN-ELSIF, and CASE
- Looping statements in PL/SQL: FOR LOOP, WHILE LOOP, and LOOP with an EXIT statement
- Anonymous blocks to execute PL/SQL statements and test stored procedures and triggers

- Motivations for stored procedures: compilation of access plans, flexibility in client-server development, implementation of complex operators, and convenient management using DBMS tools for security control and dependency management
- Specification of parameters in PL/SQL procedures and functions
- Exception processing in PL/SQL procedures and functions
- Using static cursors in PL/SQL procedures and functions
- Implicit versus explicit cursors in PL/SQL
- PL/SQL packages to group related procedures, functions, and other objects
- Public versus private specification of packages
- Typical uses of triggers in business applications: complex integrity constraints, transition constraints, update propagation, exception reporting, audit trails, simulation of generalization hierarchies, and view updates
- Trigger granularity: statement versus row-level triggers
- Trigger timing: before or after an event
- Trigger events: INSERT, UPDATE, or DELETE as well as compound events with combinations of these events
- SQL:2016 trigger specification versus proprietary trigger syntax
- Oracle BEFORE ROW triggers for complex integrity constraints, transition constraints, and data entry standardization
- Oracle AFTER ROW triggers for update propagation and exception reporting
- Most common trigger coding error: trigger target table appears in a SELECT statement in the trigger body
- Avoid most common trigger coding error by using the NEW and OLD keywords to access column values of the trigger target table
- Oracle INSTEAD OF triggers that execute in place of manipulation operations on views
- The order of trigger execution in a trigger execution procedure: BEFORE STATEMENT, BEFORE ROW, AFTER ROW, AFTER STATEMENT
- The order of integrity constraint enforcement in a trigger execution procedure
- Arbitrary execution order for overlapping triggers
- Recursive execution of a trigger execution procedure for data manipulation statements in a trigger body and actions on referenced rows
- Mutating table errors in Oracle triggers and approaches to avoid mutating table errors

QUESTIONS

1. What is a database programming language?
2. Why is customization an important motivation for database programming languages?
3. How do database programming languages support customization?
4. Why is batch processing an important motivation for database programming languages?
5. Why is development of data intensive web applications an important motivation for database programming languages?
6. Why is efficiency a secondary motivation for database programming languages, not a primary motivation?

7. Why is portability a secondary motivation for database programming languages, not a primary motivation?
8. What is a statement-level interface?
9. What is a call-level interface?
10. What is binding for a database programming language?
11. What is the difference between dynamic and static binding?
12. What is the relationship between language style and binding?
13. What SQL:2016 statements and procedures support explicit database connections?
14. What differences must be resolved to process the results of an SQL statement in a computer program?
15. What is a cursor?
16. What statements and procedures does SQL:2016 provide for cursor processing?
17. Why study PL/SQL?
18. Explain case sensitivity in PL/SQL. Why are most elements case insensitive?
19. What is an anchored variable declaration?
20. What is a logical expression?
21. What conditional statements are provided by PL/SQL?
22. What iteration statements are provided by PL/SQL?
23. Why use an anonymous block?
24. Why should a DBMS manage stored procedures rather than a programming environment?
25. What are the usages of a parameter in a stored procedure?
26. What is the restriction on the data type in a parameter specification?
27. Why use predefined exceptions and user-defined exceptions?
28. Why use the OTHERS exception?
29. How does a function differ from a procedure?
30. What are the two kinds of cursor declaration in PL/SQL?
31. What is the difference between a static and a dynamic cursor in PL/SQL?
32. What is a cursor attribute?
33. How are cursor attributes referenced?
34. What is the purpose of a PL/SQL package?
35. Why separate the interface from the implementation in a PL/SQL package?
36. What does a package interface contain?
37. What does a package implementation contain?
38. What is an alternative name for a trigger?
39. What are typical uses for triggers?
40. How does SQL:2016 classify triggers?
41. Why do most trigger implementations differ from the SQL:2016 specification?
42. How are compound events specified in a trigger?
43. How are triggers tested?
44. Is it preferable to write many smaller triggers or fewer larger triggers?
45. What is a trigger execution procedure?
46. What is the order of execution for various kinds of triggers?
47. What is an overlapping trigger? What is the execution order of overlapping triggers?

48. What situations lead to recursive execution of the trigger execution procedure?
49. List at least two ways to reduce the complexity of a collection of triggers.
50. What is a mutating table error in an Oracle trigger?
51. How are mutating table errors avoided?
52. What are typical uses of BEFORE ROW triggers?
53. What are typical uses of AFTER ROW triggers?
54. What is the difference between a hard constraint and a soft constraint?
55. What kind of trigger can be written to implement a soft constraint?
56. How does the Oracle trigger execution procedure differ from the SQL:2016 execution procedure for recursive execution?
57. What is an INSTEAD OF trigger?
58. Why are INSTEAD OF triggers useful to support operations on generalization hierarchies?
59. How does an Oracle compound trigger differ from a trigger with multiple manipulation actions?
60. What is the purpose of the FOLLOWS clause in an Oracle trigger?
61. Briefly describe common trigger coding errors and methods to resolve these errors.
62. What is the simplest method to avoid a mutating table error in an Oracle trigger?
63. Why does Oracle have the mutating table restriction?

PROBLEMS

Each problem uses the revised order entry database shown in Chapter 10. For your reference, Figure 11.P1 shows a relationship window for the revised order entry database. More details about the revised database can be found in the Chapter 10 problems.

The problems provide practice with PL/SQL coding and development of procedures, functions, packages, and triggers. In addition, some problems involve anonymous blocks and scripts to test the procedures, functions, packages, and triggers.

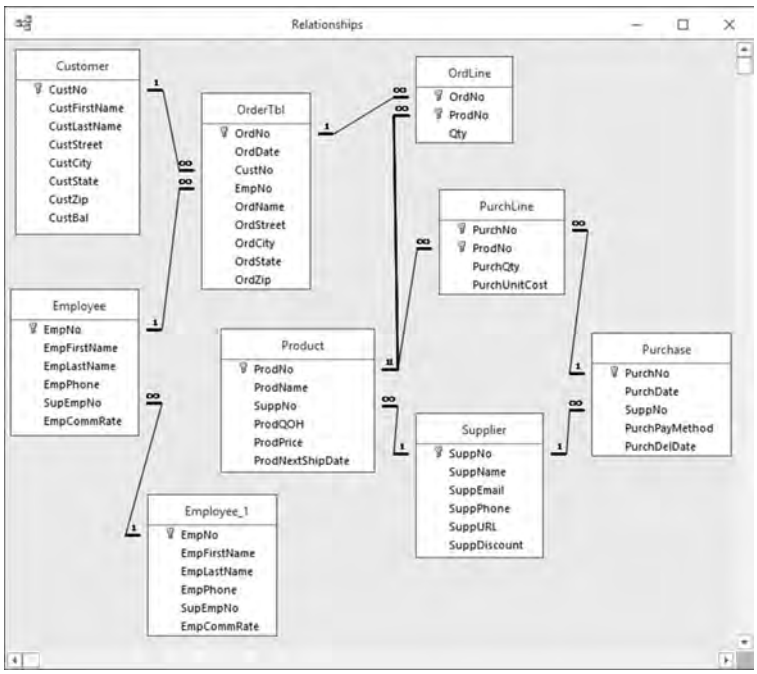


FIGURE 11.P1
Relationship Diagram for the Revised Order Entry Database

1. Write a PL/SQL anonymous block to calculate the number of days in a nonleap year. Your code should loop through the months of the year (1 to 12) using a FOR LOOP. You should use an IF-THEN-ELSIF statement to determine the number of days to add for the month. You can group months together that have the same number of days. Display the number of days after the loop terminates.
2. Revise problem 1 to calculate the number of days in a leap year. If working in Oracle 9i or beyond, use a CASE statement instead of an IF-THEN-ELSIF statement. Note that you cannot use a CASE statement in Oracle 8i.
3. Write a PL/SQL anonymous block to calculate the future value of $1,000 at 8 percent interest, compounded annually for 10 years. The future value at the end of year *i* is the amount at the beginning of the year plus the beginning amount times the yearly interest rate. Use a WHILE LOOP to calculate the future value. Display the future amount after the loop terminates.
4. Write a PL/SQL anonymous block to display the price of product number P0036577. Use an anchored variable declaration and a SELECT INTO statement to determine the price. If the price is less than $100, display a message that the product is a good buy. If the price is between $100 and $300, display a message that the product is competitively priced. If the price is greater than $300, display a message that the product is feature laden.
5. Write a PL/SQL procedure to insert a new row into the *Product* table using input parameters for the product number, product name, product price, next ship date, quantity on hand, and supplier number. For a successful insert, display an appropriate message. If an error occurs in the INSERT statement, raise an exception with an appropriate error message.
6. Revise problem 5 to generate an output value instead of displaying a message about a successful insert. In addition, the revised procedure should catch a duplicate primary key error. If the user tries to insert a row with an existing product number, your procedure should raise an exception with an appropriate error message.
7. Write testing scripts for the procedures in problems 5 and 6. For the procedure in problem 6, your script should test for a primary key violation and a foreign key violation.
8. Write a PL/SQL function to determine if the most recent order for a given customer number was sent to the customer's billing address. The function should return TRUE if each order address column (street, city, state, and zip) is equal to the corresponding customer address column. If any address column is not equal, return false. The most recent order has the largest order date. Return NULL if the customer does not exist or there are no orders for the customer.
9. Create a testing script for the PL/SQL function in problem 8.
10. Create a procedure to compute the commission amount for a given order number. The commission amount is the commission rate of the employee taking the order times the amount of the order. The amount of an order is the sum of the quantity of a product ordered times the product price. If the order does not have a related employee (a web order), the commission is zero. The procedure should have an output variable for the commission amount. The output variable should be null if an order does not exist.
11. Create a testing script for the PL/SQL procedure in problem 10.
12. Create a function to check the quantity on hand of a product. The input parameters are a product number and a quantity ordered. Return FALSE if the quantity on hand is less than the quantity ordered. Return TRUE if the quantity on hand is greater than or equal to the quantity ordered. Return NULL if the product does not exist.
13. Create a procedure to insert an order line. Use the function from problem 12 to check for adequate stock. If there is not sufficient stock, the output parameter

should be FALSE. Raise an exception if there is an insertion error such as a duplicate primary key.

14. Create testing scripts for the function in problem 12 and the procedure in problem 13.
15. Write a function to compute the median of the customer balance column. The median is the middle value in a list of numbers. If the list size is even, the median is the average of the two middle values. For example, if there are 18 customer balances, the median is the average of the ninth and tenth balances. You should use an implicit cursor in your function. You may want to use the Oracle SQL functions *Trunc* and *Mod* in writing your function. Write a test script for your function. Note that this function does not have any parameters. Do not use parentheses in the function declaration or in the function invocation when a function does not have parameters.
16. Revise the function in problem 15 with an explicit cursor using the CURSOR, the OPEN, the FETCH, and the CLOSE statements. Write a test script for your revised function.
17. Create a package containing the function in problem 15, the procedure in problem 13, the procedure in problem 10, the function in problem 8, and the procedure in problem 6. The function in problem 12 should be private to the package. Write a testing script to execute each public object in the package. You do not need to test each public object completely. One execution per public object is fine because you previously tested the procedures and functions outside the package.
18. Write an AFTER ROW trigger to fire for every action on the *Customer* table. In the trigger, display the new and old customer values every time that the trigger fires. Write a script to test the trigger.
19. Write a trigger for a transition constraint on the *Employee* table. The trigger should prevent updates that increase or decrease the commission rate by more than 10 percent of the previous commission rate. Write a script to test your trigger.
20. Write a trigger to remove the prefix "http://" in the column *Supplier.SuppURL* on insert and update operations. Your trigger should work regardless of the case of the prefix "http://". You need to use Oracle SQL functions for string manipulation. You should study Oracle SQL functions such as *SubStr*, *Lower*, and *LTrim*. Write a script to test your trigger.
21. Write a trigger to ensure that there is adequate stock when inserting a new *OrdLine* row or updating the quantity of an *OrdLine* row. On insert operations, the *ProdQOH* of the related *Product* row should be greater than or equal to the quantity in the new row. On update operations, the *ProdQOH* should be greater than or equal to the difference in the quantity (new quantity minus old quantity).
22. Write a trigger to propagate updates to the *Product* table after an operation on the *OrdLine* table. For insertions, the trigger should decrease the quantity on hand by the order quantity. For updates, the trigger should decrease the quantity on hand by the difference between the new order quantity minus the old order quantity. For deletions, the trigger should increase the quantity on hand by the old order quantity.
23. Write a script to test the triggers from problems 21 and 22.
24. Write a trigger to propagate updates to the *Product* table after insert operations on the *PurchLine* table. The quantity on hand should increase by the purchase quantity. Write a script to test the trigger.
25. Write a trigger to propagate updates to the *Product* table after update operations on the *PurchLine* table. The quantity on hand should increase by the difference between the new purchase quantity and the old purchase quantity. Write a script to test the trigger.

26. Write a trigger to propagate updates to the *Product* table after delete operations on the *PurchLine* table. The quantity on hand should decrease by the old purchase quantity. Write a script to test the trigger.
27. Write a trigger to propagate updates to the *Product* table updates to the *ProdNo* column of the *PurchLine* table. The quantity on hand of the old product should decrease while the quantity on hand of the new product should increase. Write a script to test the trigger.
28. Suppose that you have an UPDATE statement that changes both the *ProdNo* column and the *PurchQty* column of the *PurchLine* table. What triggers (that you wrote in previous problems) fire for such an UPDATE statement? If more than one trigger fires, why do the triggers overlap and what is the firing order? Modify the overlapping triggers and prepare a test script so that you can determine the firing order. Does the Oracle trigger execution procedure guarantee the firing order?
29. For the UPDATE statement in problem 28, do the triggers that you created in previous problems work correctly? Write a script to test your triggers for such an UPDATE statement. If the triggers do not work correctly, rewrite them so that they work correctly for an UPDATE statement on both columns as well as UPDATE statements on the individual columns. Write a script to test the revised triggers. Hint: you need to specify the column in the UPDATING keyword in the trigger body. For example, you can specify `UPDATING('PurchQty')` to check if the *PurchQty* column is being updated.
30. Can you devise another solution to the problem of UPDATE statements that change both *ProdNo* and *PurchQty*? Is it reasonable to support such UPDATE statements in online applications?
31. Write a trigger to implement a hard constraint on the *Product.ProdPrice* column. The trigger should prevent updates that increase or decrease the value more than 15 percent. Write a script to test the trigger.
32. Write a trigger to implement a soft constraint on the *Product.ProdPrice* column. The trigger should insert a row into an exception table for updates that increase or decrease the value more than 15 percent. You should use the exception table shown in Example 11.33. Write a script to test the trigger.
33. Convert the employee generalization hierarchy shown in Figure 11.P2 into a table design using the Generalization Hierarchy Rule (see Section 6.4.3).
34. Define two views for the table design of the employee generalization hierarchy view. The first view (salary employee view) should contain all rows and columns (direct and inherited) of salaried employees. The second view (hourly employee view) should contain all rows and columns (direct and inherited) of hourly employees.
35. Write triggers and associated testing code to insert rows into the salary and hourly employee views. You should use INSTEAD OF INSERT triggers to map the view operations into base table operations.

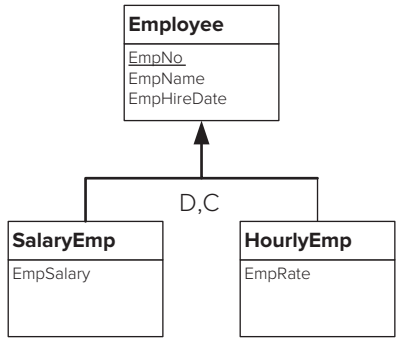


FIGURE 11.P2
Generalization Hierarchy for Employees

36. Modify the INSTEAD OF INSERT trigger from problem 35 on the salary employee view to enforce the disjointness constraint. Write testing code for the modified trigger.
37. As an alternative to the modification of the INSTEAD OF INSERT trigger in problem 36, write a trigger on the *SalaryEmp* table to enforce the disjointness constraint. Write testing code for the new trigger. Does the trigger fire when inserting a row in the salary employee view? Please explain your answer.
38. Write a trigger and associated testing code for updating columns of the salary employee view. You should use an INSTEAD OF UPDATE trigger to map the view operations into base table operations.
39. Do you need to write a trigger to enforce the completeness constraint for a generalization hierarchy? Please explain your answer. Try to write a trigger to explain your answer.
40. Write a trigger for a soft transition constraint on the *Employee* table. The trigger should insert a row into an exception table for updates that increase or decrease the commission rate by more than 10 percent of the previous commission rate. You should use the exception table shown in Example 11.33. Write a script to test the trigger.
41. Write an INSTEAD OF INSERT trigger to support insert operations on a view joining the *Customer* and *OrderTbl* tables. You should create a view containing all columns of each table. The trigger should map insert operations on the view to the base tables. If the customer number provided in the insert statement does not exist in the *Customer* table, a row should be inserted in both tables. Otherwise, just insert a row into the *OrderTbl* table. Write a script to test the trigger.
42. Write an INSTEAD OF UPDATE trigger to support update operations on a view joining the *Customer* and *OrderTbl* tables. You should create a view containing all columns of each table. The trigger should map update operations on the view to the appropriate base tables excluding the primary keys of each table and the *CustNo* foreign key of *OrderTbl*. Write a script to test the trigger.
43. Consider a trigger on the *PurchLine* table involving insert or update operations on *ProdNo*. The trigger should raise an error if a row is inserted or *ProdNo* changed such that the new *ProdNo* value has a different supplier than other products for the same purchase. Identify issues involving mutating tables for this trigger and develop a plan to avoid a mutating table error when coding the trigger if a problem exists.
44. Implement the trigger in problem 43 including any required procedures or functions to avoid a mutating table error. Provide test cases to demonstrate that your trigger works correctly.

REFERENCES FOR FURTHER STUDY

Although this chapter provides the most authoritative coverage of triggers of any reference, you may want to augment this background for additional DBMS specific details. The Oracle Technology Network (www.oracle.com/technetwork) contains a wealth of material about PL/SQL, stored procedures, and triggers. The *PL/SQL User's Guide* provides details about PL/SQL and stored procedures. The *Oracle SQL Reference* provides details about triggers as well as descriptions of predefined functions such as **Mod** and **SubStr**. More details and examples can be found in the *Oracle Concepts* and the *Oracle Application Developers Guide*. Melton and Simon (2001) describe triggers in SQL:1999.

part six

Data Warehouse Processing

Part 6 provides detailed coverage of data warehouse management and design, data integration, and query formulation. Chapter 12 presents basic concepts, management background, and examples of data warehouses in important industries. Chapter 13 describes conceptual design of data warehouses with coverage of multidimensional representation, schema patterns, summarizability patterns, and the schema integration process. Chapter 14 provides details about data integration concepts, techniques, and tools. Chapter 15 covers query formulation details about online analytic processing, SQL SELECT statement extensions for subtotal calculations, SQL SELECT statement extensions for analytic functions, and summary data management.

chapter

12

Data Warehouse Concepts and Management

Learning Objectives

This chapter explains basic concepts and management principles of data warehouses, historical databases used for business intelligence. After this chapter, the student should have acquired the following knowledge and skills:

- Explain conceptual differences between operational databases and data warehouses
- Relate learning curve concepts to management of data warehouse development
- Describe architectures for data warehouse development in organizations
- Explain maturity concepts for development of data warehouses including architecture selection, stages of growth, and capability assessment
- Discuss insights about enterprise data warehouse development through details of a business strategy game for data warehouse development
- Discuss architecture and maturity concepts of data warehouses in retail, education, and health care

OVERVIEW

Imagine a corporate executive of a global retail firm asking the question, "What retail stores were the top producers during the past 12 months in major geographic regions?" Follow-up questions may include, "What were the most profitable products in the top producing stores?" and "What were the most successful product promotions at the top producing stores?" These questions are typical business intelligence questions, asked every day by managers all over the world. Answers to these questions often require complex SQL statements that may take hours to code and execute. Furthermore, formulating some of these queries may require data from a diverse set of internal legacy systems and external market sources, involving both relational databases and unstructured data.

Business intelligence questions such as those in the previous paragraph pose new requirements for data modeling, database design methodologies, database infrastructure, query formulation, data integration, and DBMS features. This chapter presents characteristics and management concepts for data warehouses deployed to satisfy requirements of business intelligence. You will initially learn about unique requirements for data warehouse processing as opposed to transaction processing. Then you will learn about management concepts for developing data warehouses in organizations over time.

This chapter augments this conceptual background with details of a business strategy game for data warehouse development. To provide a concrete context for conceptual material in this chapter, you will see examples of enterprise data warehouses in retail, education, and health care.

Chapter 12 begins Part 6 with four chapters about data warehouses. This chapter provides a conceptual foundation about data warehouse characteristics and management principles. For basic concepts, you will learn historical reasons for data warehouse usage and technology development and characteristics of data warehouses. For management principles, you will learn development challenges, learning curve characteristics, and maturity concepts as well as examples of data warehouse usage in important industries. Chapters 13 to 15 extend this conceptual foundation with detailed skills about data warehouse design, data integration, and query formulation for enterprise data warehouses. Collectively, the chapters in Part 6 provide breadth and depth of coverage to support career aspirations as a data warehouse or business intelligence professional.

12.1 BASIC CONCEPTS

The type of data used for business intelligence purposes is conceptually different from data used in transaction processing systems. Data requirement differences reflect fundamentally different processing environments for daily operations and business intelligence. This section examines differences in processing environments leading to characteristics of data warehouses that support business intelligence needs of organizations.

12.1.1 Transaction Processing versus Business Intelligence

Transaction processing involves different needs in an organization than business intelligence. A transaction is a unit of repetitive work such as a product order, airline reservation, bill payment, or course enrollment. Transaction processing, as presented in Chapter 17, allows organizations to conduct daily business in an efficient manner. Operational or production databases used in transaction processing assist with decisions such as tracking orders, resolving customer complaints, and assessing staffing requirements. These decisions involve detailed data about business processes. In contrast, business intelligence processing helps management provide medium-term and long-term direction for an organization. Management needs support for decisions such as capacity planning, product development, store locations, product promotions, and sales forecasts.

These organizational needs involve different data requirements for transaction processing and business intelligence as shown in Figure 12.1. Transaction processing requires primary data from large volumes of transactions to support daily operations and short-term decision making of an organization. In contrast, business intelligence processing requires transformed secondary data from primary data sources to

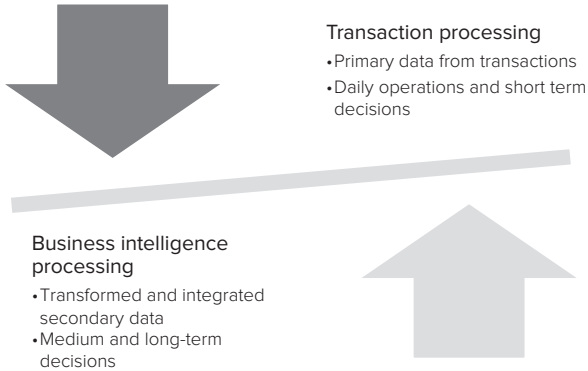


FIGURE 12.1
Data Requirements for Transaction Processing and Business Intelligence

support medium and long-term decision-making. A data warehouse supports decision-making with longer-term impact through transformations and integration of operational databases and external data sources.

Historically, most organizations have assumed that operational databases along with relational database technology could provide adequate support for business intelligence. As organizations developed operational databases for various functions, an information gap developed. Gradually, organizations and DBMS vendors realized limitations of operational databases and relational database technology for business intelligence as depicted in Figure 12.2 and explained in the following points.

- Organizations and DBMS vendors experienced performance problems with using an individual database for both transaction processing and business intelligence processing. The demands of transaction processing and business intelligence processing differ so sharply that an individual database could not provide adequate performance for both purposes.
- Organizations realized that lack of integration among operational databases hindered higher-level decision-making. This lack of integration was not a design failure as operational databases primarily support transaction processing, not business intelligence processing. Organizations learned that retrofitting integration for operational databases was difficult.
- Product vendors realized that DBMSs lacked key features to support summary data retrieval and analytical calculations, vital for business intelligence processing. The SQL GROUP BY clause was inadequate for queries involving summary data. The SELECT statement lacks features for analytical calculations such as moving averages. Storage and optimization methods were inadequate for queries involving summary data.

Since the early 1990s, a consensus has emerged that operational databases must be transformed for business intelligence support. Operational databases can contain inconsistencies in formats, entity identification, frequency of update, and units of measure that hamper usage in business intelligence. In addition, organizations need a broad view that integrates business processes for business intelligence. Because of different requirements and performance limitations, operational databases are usually separate from databases for business intelligence. Using a common database for both kinds of processing can significantly degrade performance and make it difficult to summarize activity across business units.

Commercial software vendors have performed substantial amounts of research and development to add features for business intelligence. Initially, a new breed of companies developed storage engines, summary data retrieval, and data transformation tools for business intelligence. Later, relational DBMS vendors added features for efficient management of summary data, query language extensions for summary data, optimization of queries involving summary data, and transformation of operational databases. The technology provided by both business intelligence firms and relational DBMS vendors has rapidly matured to provide strong commercial solutions for developing and managing data warehouses.

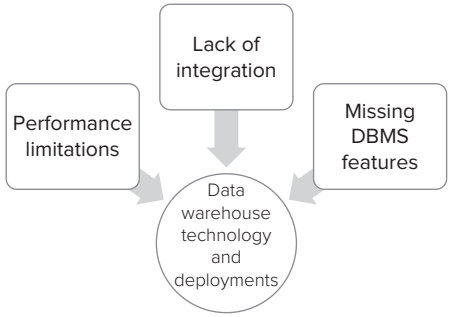


FIGURE 12.2
Technology and Deployment Limitations

12.1.2 Characteristics of Data Warehouses

Data Warehouse
a logically centralized repository containing transformed data from operational databases and external data sources.

Data warehouse, a term coined by William Inmon in 1990, refers to a logically centralized data repository where data from operational databases and other sources are integrated, cleaned, and standardized to support business intelligence. The transformational activities (cleaning, integrating, and standardizing) are essential for achieving benefits. Data warehouses are optimized for reporting often involving summarization of large amounts of data as well as periodic processing to integrate and transform source data.

This definition of a data warehouse extends to four distinguishing characteristics, as described in the following points.

1. *Subject-Oriented:* A data warehouse is organized around major business subjects or entities such as customers, orders, and products. This subject orientation contrasts to the process orientation for transaction processing.
2. *Integrated:* A data warehouse integrates data from multiple operational databases and external data sources to provide a single, unified database for business intelligence. Integration requires consistent naming conventions, common data formats, and comparable measurement scales across data sources. In addition, integration involves matching and merging entities such as customers across data sources.
3. *Time-Variant:* Data warehouses use time stamps to represent historical data. The time dimension supports identification of trends, prediction of future operations, and determination of operating targets. Data warehouses essentially consist of a long series of snapshots, each of which represents operational data captured at a point in time.
4. *Nonvolatile:* New facts in a data warehouse are appended, rather than replaced, preserving historical data. Refreshing a data warehouse primarily involves appending new facts with a much lower level of updates to related business entities. Dominance of inserting new facts ensures that update and deletion anomalies are secondary for data warehouses. Transaction data are transferred to a data warehouse only when most updating activity has been completed.

Table 12-1 further depicts characteristics of data warehouses compared to operational databases. Transaction processing relies on operational databases with current data at the individual level, while business intelligence processing utilizes data warehouses with historical data at both the individual and summarized levels. Individual-level data provides flexibility for responding to a wide range of business intelligence needs while summarized data provides fast response to repetitive queries. For example, an order-entry transaction requires data about individual customers, orders, and inventory items, while a business intelligence application may use monthly sales to customers over a period of several years. Operational databases therefore have a process orientation relevant to a particular business activity, compared to a subject orientation

TABLE 12-1
Comparison of Operational Databases and Data Warehouses

Characteristic	Operational Database	Data Warehouse
Currency	Current	Historical
Detail level	Individual	Individual and summary
Design orientation	Process orientation	Subject orientation
Rows per retrieval	Few	Thousands to millions
Normalization level	Mostly normalized	Relaxed design from BCNF
Modification level	Volatile	Nonvolatile (refreshed)
Data representation	Relational	Relational model with star schemas and multidimensional model with data cubes

for data warehouses with all customer data across business processes. A transaction typically updates only a few rows, whereas a business intelligence application may query thousands to millions of rows.

Data integrity and usage patterns of transaction processing require that operational databases be highly normalized. In contrast, data warehouses typically have some violations from Boyce-Codd Normal Form to reduce effort to join large tables. Most applications in business intelligence involve retrievals and periodic insertions of new data. These operations do not suffer from redundancies caused by violations of normal forms.

Because of the different processing requirements, data warehouse designs use different data representations than operational databases. The relational data model dominates for operational databases. In the early years of data warehouse deployment, the multidimensional data model dominated. Relational databases now dominate for data warehouses with a schema pattern known as a star schema. The multidimensional data model now typically supports a business analyst representation of a data warehouse.

12.1.3 Applications of Data Warehouses

Organizations undertake data warehousing projects for competitive reasons, to achieve strategic advantage or to stay competitive. In many industries, a few organizations pioneered deployment of data warehouses to gain competitive advantage. Often, organizations undertook a data warehousing project as part of a corporate strategy to shift from a product focus to a customer focus. Successful data warehouses have helped identify new markets, focus resources on profitable customers, improve retention of customers, and reduce inventory costs. After success by the pioneering organizations, other organizations quickly followed to stay competitive.

Organizations in a wide range of industries and government areas have developed data warehouses. A few key applications have driven the adoption of data warehousing projects as listed in Table 12-2. Highly competitive industries such as retail, insurance, airlines, and telecommunications invested early in data warehouse technology and projects. Less competitive industries such as regulated utilities were slower to invest although they now have substantial investments in data warehouse deployment.

The maturity of data warehouse deployment varies among industries and organizations. Early adopters of data warehouses have deployed data warehouses since the early 1990s while later adopters have deployed data warehouses since the 2000s. With the rapid development of data warehouse technology and best practices, continued investment in data warehouse technology and management practices are necessary to sustain business value. Many large organizations have made major redevelopments of their data warehouses to leverage improved technology and practices. To provide guidance about investment decisions and management practices, organizations use comparisons to peer organizations to gauge the level of data warehouse usage.

Data Mining Data mining has emerged as a key application of data warehouses across industries. The ability to discover hidden patterns in data can substantially increase benefits provided by a data warehouse. **Data mining** refers to the process of discovering implicit patterns in data and using these patterns for business advantage. Data mining facilitates the ability to detect, understand, and predict patterns.

Data Mining
the process of discovering implicit patterns in data and using those patterns for business advantage.

TABLE 12-2
Data Warehousing Applications by Industry

Industry	Key Applications
Airline	Yield management, route assessment
Telecommunications	Customer retention, network design
Insurance	Risk assessment, product design, fraud detection
Retail	Target marketing, supply-chain management

The most common application of data mining techniques is target marketing. Retail companies can increase revenues and decrease costs if they can identify likely customers and eliminate customers not likely to purchase. Data mining techniques allow decision makers to focus marketing efforts by customer demographic and psychographic data. The entertainment, financial services, travel, and consumer goods industries also have benefited from data mining techniques. For example, the financial services industry uses data mining techniques to develop new financial products and promote them to customers likely to purchase them.

Requirements of data mining require extensions to capabilities traditionally provided by a mature data warehouse. Data mining needs more detailed data than traditional data warehouses provide. The volumes and dimensionality of data can be much greater for data mining techniques than other analysis methods using data warehouse queries. Data mining techniques thrive with clean, high-dimensional, transaction data. To support these data mining requirements, many data warehouses now store data at the level of the individual customer, product, and so on.

Data mining requires a collection of tools that extend beyond traditional statistical analysis tools. Traditional statistical analysis tools are not well suited to high dimensional data with a mix of numeric and categorical data. In addition, traditional statistical techniques do not scale well to large amounts of data. Data mining typically includes the following kinds of tools:

- Data access tools to extract and sample transaction data according to complex criteria from large databases
- Data visualization tools that enable a decision maker to gain a deeper, intuitive understanding of data
- A rich collection of models to cluster, predict, and determine association rules from large amounts of data. The models involve neural networks, genetic algorithms, decision tree induction, rule discovery algorithms, probability networks, and other technologies.
- An architecture that provides optimization, client-server processing, and parallel processing to scale to large amounts of data

Data mining provides insights that may elude traditional techniques for data warehouse queries. Data mining holds the promise of more effectively leveraging data warehouses by providing the ability to identify hidden relationships in data. It facilitates data-driven discovery, using techniques such as building association rules (e.g., between advertising budget and seasonal sales), generating profiles (e.g., buying patterns for a specific customer segment), and generating predictions. This knowledge may help to improve business operations in critical areas, enabling target-marketing efforts, better customer service, and improved fraud detection.

12.2 MANAGEMENT OF DATA WAREHOUSE DEVELOPMENT

Despite potential benefits of a data warehouse, many organizations have struggled with data warehouse development and usage. A typical data warehouse project involves a large capital investment, typically more than $1 million in just the first year (AbuAli and Abu-Addose, 2010). According to a Gartner study in 2005, half of organizations have experienced failures in initial efforts to develop a data warehouse or achieved limited acceptance after deployment. Thus, organizations must carefully manage data warehouse development to achieve potential benefits.

To provide background and insight about data warehouse development, this section presents management concepts important for data warehouse deployment in organizations and demonstrates a business strategy game involving data warehouse development. You will first learn about development difficulties and organizational learning concepts typical in data warehouse projects. Then, you will learn about common architectures to deploy data warehouses in organizations. You will next learn

about major concepts of data warehouse maturity including architecture selection, stages of growth, and capability assessment. After studying architectures and maturity concepts, you will learn details about a business strategy game that provides simulated experience with data warehouse development in organizations.

12.2.1 Development Challenges and Learning Effects

Development projects for data warehouses face potentially high and unforeseen costs along with intangible benefits as depicted in Figure 12.3. Coordination challenges and uncertain data quality levels drive potential high costs. Large data warehouse projects involve coordination among many parts of an organization. Many organizations have underestimated the time and effort to achieve coordination to reconcile different parts of a data warehouse. Because an organization may not have attempted to integrate operational databases, data quality problems may be unknown. A slow period of discovery of data quality problems can be costly. Coordination and poor data quality have caused substantial cost overruns in development projects for data warehouses.

In addition to potentially high development costs, organizations struggle with intangible benefits from data warehouse investments. Although organizations struggle to quantify intangible benefits, organizations often deem intangible benefits important to long-term success. Traditionally, intangible benefits involve brand recognition, employee expertise, and management skill. Intangible benefits for a data warehouse typically involve increased data quality through fewer missing values, larger number of matched entities particularly customers, higher levels of standardization, and more data availability. After years of usage, intangible benefits may become tangible as examples of increased revenue and reduced expenses can be associated to analysis not possible before deployment of a data warehouse. For example, a data warehouse may enable reduced losses due to improved fraud detection, improved customer retention through targeted marketing, and reduction in inventory carrying costs through improved demand forecasting.

Learning curves provide insight to understand intangible benefits and high costs in data warehouse projects. The traditional learning curve (Figure 12.4) indicates improvement in performance of a skill as a function of learning effort such as number of trials. The skill learning curve features a slow beginning with a high fixed cost (such as many trials) to gain a rudimentary skill level. Essentially, the slow beginning indicates a high fixed cost to learn a skill. At some point, a learner gains enough insight to propel performance to much higher levels with few additional trials. After this period of rapid advancement, a learner reaches a plateau in which it is difficult to advance skill performance much.

The learning curve for production (Figure 12.5) has a different shape but some similar lessons as the learning curve for skills. Organizations in high technology manufacturing have applied the production learning curve to complex products such as aircraft production. This curve switches the axes with units produced on the x-axis

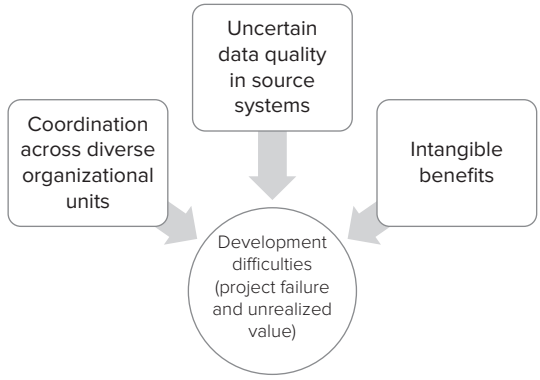


FIGURE 12.3
Drivers of Difficulties in Data Warehouse Development Projects

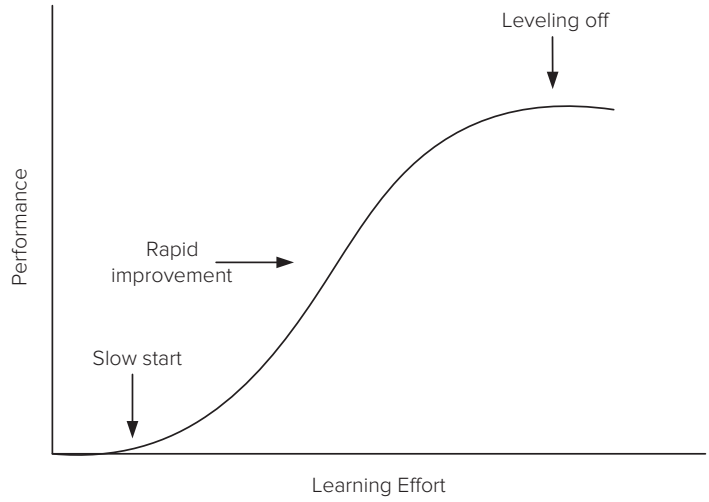


FIGURE 12.4
Traditional Learning Curve for Skill Attainment

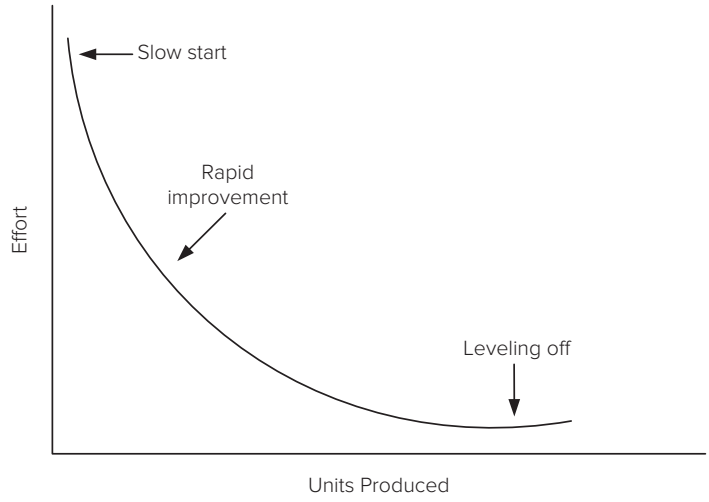


FIGURE 12.5
Traditional Learning Curve for Production

and effort such as work hours or cost on the y-axis. The curve features a steep negative slope with few units produced at high cost due to discovery and correction of problems in initial production. The curve then enters steep deceleration after resolution of production problems and production of more units. Finally, the curve enters the plateau region after an organization resolves most quality problems.

The major lesson from the production cost curve is to expect high initial costs to resolve quality problems especially for products with large levels of innovation. Productivity improvements occur rapidly after this initial period with sharply falling costs per unit produced.

The skill and production learning curves provide insight about maturity of data warehouse usage in an organization. The business value learning curve (Figure 12.6) shows the hypothetical relationship between deployment time of a data warehouse in an organization and business value derived from its usage. The business value learning curve shows the characteristics of a skill attainment curve with an initial period of low business value, followed by a period of rapid acceleration of business value until reaching a plateau. The key insight from the business value learning curve is the initial difficulty to create high value from combining data sources.

The data transformation learning curve (Figure 12.7) shows the hypothetical relationship between time that a data warehouse has been deployed and transformation cost to resolve data quality problems. The data transformation learning curve has

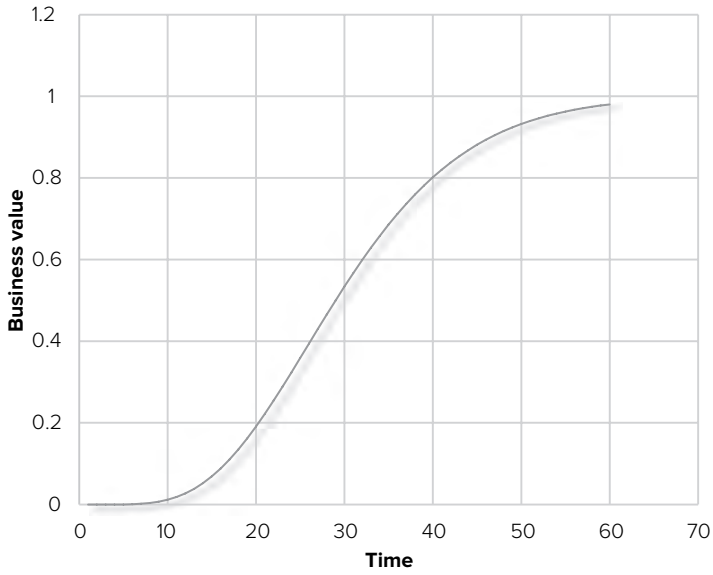


FIGURE 12.6
Learning Curve for Business Value of a Data Warehouse

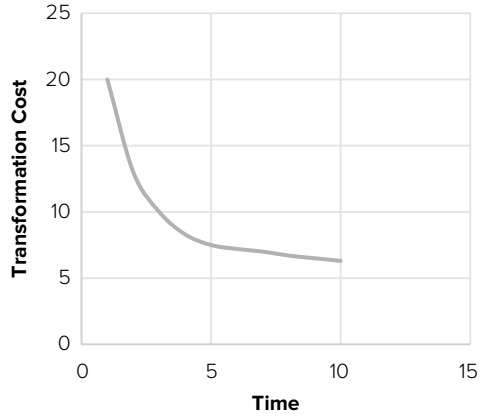


FIGURE 12.7
Learning Curve for Data Transformation Costs

characteristics of a production learning curve with high costs to resolve initial data quality problems followed by a period of steep decline in costs until reaching a bottom plateau. The key insight from the curve is a high fixed cost to discover and resolve unknown data quality problems during the initial period of usage.

According to learning curve concepts, organizations should mature in their usage of data warehouses over time with increasing benefits and decreasing costs. However, many organizations struggle with stagnant benefits and increasing costs, lacking a coherent strategy to develop a data warehouse over time. The next subsections present important tools that help organizations mature in deployment of data warehouses.

12.2.2 Architectures for Data Warehouse Deployment

When applied to computer systems, architecture refers to an organization of components to support specified goals. For data warehouses, business goals drive technology choices so architecture is primarily a business issue, not a technology issue. The scope and integration level are important factors in determining an appropriate architecture. Scope refers to the breadth of an organization supported by a data warehouse. Scope can be measured in various ways such as the number of data sources used, number of source systems providing inputs, or the number of organizational units providing inputs or using a data warehouse. Integration level refers to several data quality

indicators across data sources. Integration level involves completeness, consistency, conformity, and duplication across data sources, not within individual data sources.

The enterprise data warehouse architecture provides a scalable approach to support a large number of data sources and business units as depicted in Figure 12.8. To assist with the transformation process, an organization should create an **enterprise data model** (EDM). An EDM describes the structure of a data warehouse and contains descriptive data required to access operational databases and external data sources. The EDM may also contain details about cleaning and integrating data sources.

Enterprise Data Model
a conceptual data model of the data warehouse defining the structure of a data warehouse and the descriptive data to access and transform operational databases and external data sources.

Departmental users generally need access and control of small portions of a data warehouse, instead of an entire warehouse. To provide them with faster access while isolating them from data needed by other user groups, smaller data warehouses called **data marts** are often used. Data marts act as the interface between end users and a corporate data warehouse, storing a subset of data and refreshing those data on a periodic (e.g., daily or weekly) basis. Generally, a data warehouse and associated data marts reside on different servers to improve performance and fault tolerance. Organizational units retain control over their own data marts, while the data warehouse remains under the control of corporate information systems staff.

Data Mart
a subset or view of a data warehouse, typically at a department or functional level, that contains all data required for business intelligence tasks of that department.

A staging area can augment an enterprise data warehouse to support data transformation and operational decision-making. A staging area provides temporary storage of transformed data before loading into the data warehouse. Organizations with a large number of data sources and complex transformations typically use staging areas. Some organizations use staging areas to support operational decision-making for performance or productivity reasons. Staging areas supporting operational reporting requirements are known as operational data stores.

Given the large investment required to develop an enterprise data warehouse, some organizations opt for lower cost approaches without an enterprise data model. These organizations employ a bottom-up approach to data warehousing as depicted in Figure 12.9. In the data mart architecture, an organization develops separate data marts with relatively small scope and no integration among data marts. The set of data marts may evolve into a large data warehouse if the organization can justify the expense of building an enterprise data model.

The data mart bus architecture (Kimball 2003a) combines features of the enterprise data warehouse and data mart architectures as depicted in Figure 12.10. A data mart bus transforms data sources into common business entities according to organizational standards before loading transformed data into data marts. Similar to the data mart architecture, each data mart contains its own data model to support a limited number of user departments.

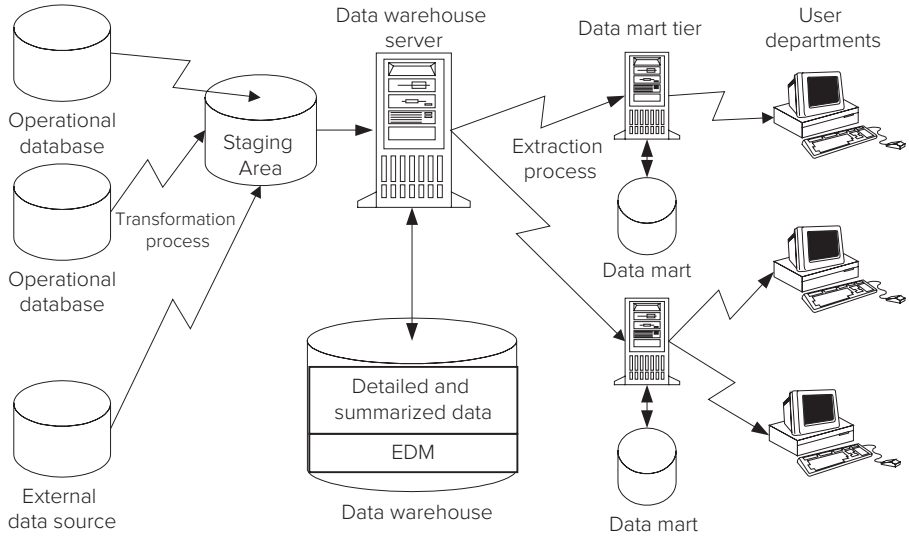


FIGURE 12.8
Enterprise Data Warehouse Architecture with Staging Area

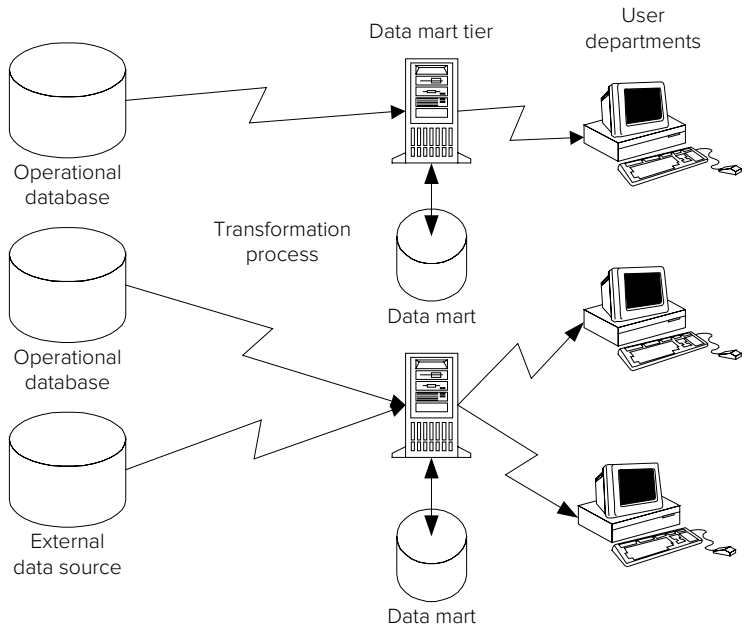


FIGURE 12.9
Data Mart Architecture

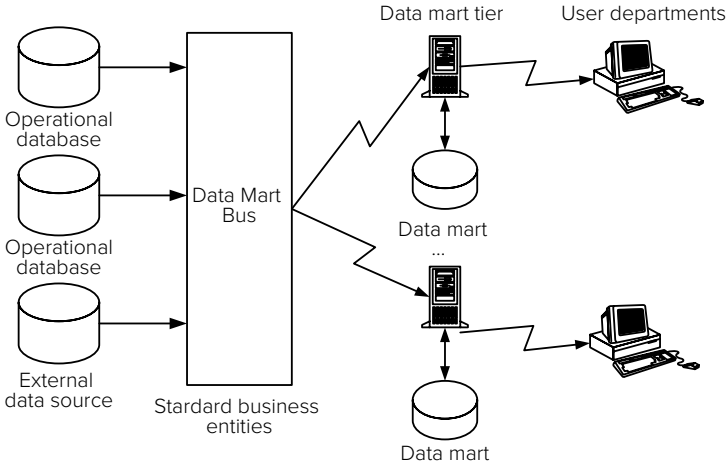


FIGURE 12.10
Data Warehouse Bus Architecture

For highly decentralized or independent organizations, the federated data warehouse architecture provides another compromise approach. As depicted in Figure 12.11, the federated data warehouse approach supports two levels of data warehouses. Each organization independently maintains one or more data warehouses using any of the architectures. To provide inter-organizational sharing, each organization contributes to the federated data warehouse. Typically, another layer of data integration and a query portal support data sharing in the federated data warehouse. Depending on the environment, participation can be voluntary or compulsory (typically required by government agencies). Some users of a federated data warehouse may be external stakeholders, not members of participating organizations.

Organizations driven by major events may develop oper marts (short for operational mart) as noted by Imhoff (2001). An oper mart, a just-in-time data mart, is usually built from one operational database in response to major events such as disasters and new product introductions. An oper mart supports peak demand for reporting and business analysis that accompanies a major event. After the decision support demand subsides, an organization may dismantle an oper mart or merge it into an existing data warehouse or data mart.

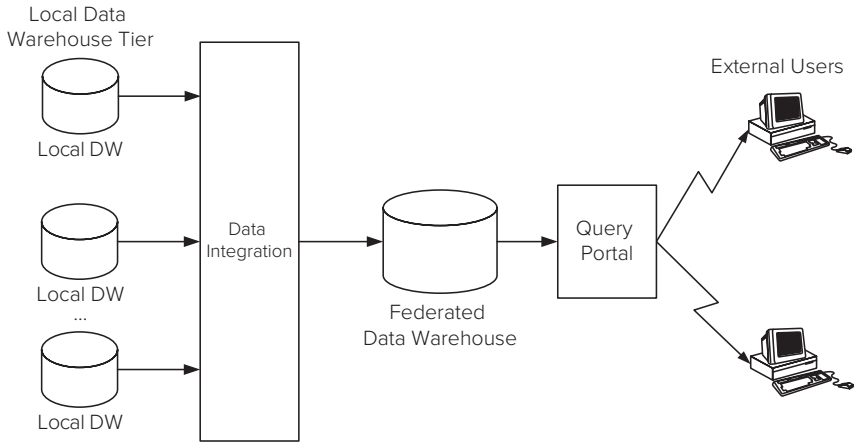


FIGURE 12.11
Federated Data Warehouse Architecture

12.2.3 Data Warehouse Maturity Concepts

Maturity in deployment of information technology assets has become an important concern for organizations. For data warehouses, maturity involves selection of an appropriate architecture, monitoring of stages of maturity, and acquisition of capabilities to implement an architecture.

Architecture Selection Two opposing approaches provide limits on architectures for data warehouses as depicted in Figure 12.12. The top-down approach develops an enterprise data warehouse, a single data warehouse for an organization. Through logical centralization, the top-down approach has a high level of integration and wide scope. The bottom-up approach provides independent data marts with lower integration levels. The data mart bus approach provides a compromise approach, combining development of data marts with some level of integration. The federated approach provides another compromise approach to facilitate data sharing among additional stakeholders.

Learning effects provide bias for data warehouse projects of smaller scope and lower integration level. Intangible business value creates uncertainty about return on data warehouse investments, especially data warehouses with a wide scope. Project risks provide uncertainty about costs for data warehouses with a wide scope. Learning effects typically postpone benefits while incurring costs early. Over longer periods, an organization can mitigate these learning difficulties, however.

To overcome bias for data warehouses with narrow scope and low integration levels, senior management in an organization should develop a high strategic view of information technology. The level of sponsorship, information independence, and task routineness influence an organization's strategic view. Organizations with broad sponsorship for projects, information dependence for long-term decision making, and lower task routineness will have a higher strategic view, favorable to the enterprise data warehouse approach.

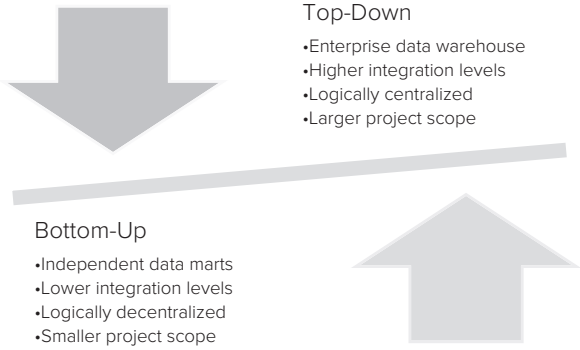


FIGURE 12.12
Characteristics of Opposing Data Warehouse Architectures

Choudary (2010) identified factors influencing architecture selection for an organization as summarized in Tables 12-3 and 12-4. An organization should select a lower level of integration with a data mart architecture if it has high resource constraints, high urgency for information requirements, and low levels of other factors. An organization should select a medium level of integration in a data mart bus architecture if it has moderate resource constraints, moderate need for data integration, and moderate strategic view. An organization should select a higher integration level in an enterprise data warehouse architecture if it perceives data as a strategic resource. An organization should have low resource constraints, high need for data integration, and high sponsorship levels.

Stages of Growth The **data warehouse maturity model** (Eckerson 2007) provides guidance for investment decisions for data warehouses by stages of growth. The maturity model consists of six stages as summarized in Table 12-5. The stages provide a framework to view an organization's progress, not an absolute metric as organizations may demonstrate aspects of multiple stages at the same time. As organizations move from lower to more advanced stages, increased business value can occur. However, organizations may have difficulty justifying significant new data warehouse investments as benefits can be intangible, difficult to quantify.

Data Warehouse Maturity Model
proposed to provide guidance for data warehouse investment decisions. The stages of the Data Warehouse Maturity Model provide a framework to view an organization's progress, not an absolute metric as organizations may demonstrate aspects of multiple stages at the same time.

An important insight of the maturity model is the difficulty of moving between certain stages. For small but growing organizations, moving from the infant to the child stages can be difficult because a significant investment in data warehouse technology is necessary. For large organizations, the struggle is to move from the teenager to the adult stage. To make the transition, upper management must perceive the data warehouse as a vital enterprise resource, not just a tool provided by the information technology department.

The Data Warehouse Institute (TDWI) has developed a survey tool (available at tdwi.org) to assess the maturity of individual organizations in data warehouse development. The tool provides a maturity score in the categories of scope, sponsorship, funding, value, architecture, data, development, and delivery. The tool uses maturity scores to compare organizations based on industry, company size, budget, and other variables.

TABLE 12-3
Summary of Organization Factors

Factor	Description
Resource constraints	Level of slack resources (financial, computing capacity, and personnel) for data warehouse implementation
Information technology skills	Level of skill of information technology organization
Integration need	Level of reports and business decisions requiring data from different business units and data sources
Sponsorship	Ability to obtain funding and support from top management
Strategic view	Perception of organizational value from business intelligence investments
Urgency	Need for fast implementation of information technology requirements

TABLE 12-4
Data Warehouse Architecture and Organization Factor Levels

	Data Warehouse Architecture		
Organizational Factor	**Data Mart**	**Data Mart Bus**	**Enterprise**
Resource constraints	High	Moderate or High	Low
Information technology skills	Low	Moderate	High
Integration need	Low	Moderate	High
Sponsorship	Low	Low or Moderate	High
Strategic view	Low	Moderate	High
Urgency	High	High or Moderate	Low

TABLE 12-5
Stages of the Data Warehouse Maturity Model

Stage	Scope	Architecture	Management Usage
Prenatal	System	Management reporting	Individual employees
Infant	Individual business analysts	Operational reports and spreadsheets (known as spreadmarts)	Management insight
Child	Departments	Data marts	Support business analysis
Teenager	Divisions	Data warehouses	Track business processes
Adult	Enterprise	Enterprise data warehouse	Drive organization
Sage	Inter-enterprise	Web services and external networks	Drive market and industry

Source: Eckerson 2007

The Data Warehouse Maturity Model has been extended into the Business Intelligence (BI) Maturity Model by the Data Warehouse Institute. The BI Maturity Model adds dimensions of usage, insight, control, and business value to the adoption dimension of the original maturity model. As BI usage matures, organizations progress from information backlogs to utility support with access to power users, casual users, and external parties. Insight improves as BI adoption matures with organizations improving decision automation by increasing data freshness and decreasing decision latency. For control, organizations balance flexibility and standards, generally adopting more organization wide standards as maturity increases. Business value increases slowly as adoption matures with final stages of maturity providing competitive business advantage.

Capability Assessment Capability assessment involves attainment of appropriate maturity levels in process areas and activities. The most influential approach, the Capability Maturity Model (CMM), was developed in the early 1990s by the Software Engineering Institute. The CMM contains five maturity levels (initial, repeatable, defined, managed, and optimized) showing progress in software development capabilities. To achieve a specified maturity level, the CMM stipulates features for key process areas. For example to achieve the managed level, an organization must achieve levels of quantitative process management and software quality management.

Sen et al. (2012) applied the CMM to data warehouse maturity. They proposed a detailed set of activities and features to achieve each capability level for both data warehouse development and operations. Table 12-6 summarizes key process areas for each maturity level for data warehouse development. The optimizing level requires maturity in higher level activities neglected by most organizations. Lower levels of maturity require more basic activities common to many organizations.

12.2.4 Business Strategy Game for Data Warehouse Development

Data warehouse architectures and maturity are abstract concepts, difficult to apply in an organizational setting. Business strategy games provide a learning approach to apply abstract concepts in simulated environments.

For data warehouse development, students have difficulty to obtain insights gained from experiencing project relationships involving costs and benefits over extended time periods. Benefits of data warehouse deployment are often intangible especially during initial periods of usage. Benefits become tangible and increase as organizational units increase usage. In contrast, costs are tangible and high during data warehouse development especially with uncertain levels of data quality. Costs decline as benefits increase during usage of a data warehouse over time. Learners need to gain experience from balancing costs and benefits as organizations acquire capabilities to support an organization's strategy for developing a data warehouse.

This section provides an overview of Emerge2Maturity, a business strategy game that addresses these difficulties. Emerge2Maturity decomposes complexity of data warehouse development into a sequence of standard steps with common factors across

Maturity Level	Key Process Areas
Optimizing	Change management for meta data and technology, defect prevention, process improvement program
Managed	Data quality management, process management, service level management
Defined	Process definition, stakeholder management process, architecture alignment, data quality assurance, service level agreement, resource management, configuration management
Repeatable	Sponsor assurance, project planning and tracking, business justification, requirements management
Initial	Project management

TABLE 12-6
Summary of Key Process Areas for Data Warehouse Development

organizations. To help learners understand relationships among strategy and capability assessment, Emerge2Maturity combines aspects of strategy and capability assessment. For strategy, Emerge2Maturity shows trends, costs, and benefits with increased benefits and decreased costs over time. For capability assessment, Emerge2Matury uses simulation so that learners can observe results of their decisions before implementing them. Simulation in Emerge2Maturity uses simple models to quantify costs and benefits related to development choices made by players.

Game Flow Emerge2Maturity supports decision making over a number of phases as depicted in Figure 12.13. Players attempt to maximize profit (benefits minus costs) subject to constraints about budget and resource levels. Emerge2Maturity uses uncertain demand for information assets so players deal with risk in assessing capabilities.

In each phase, a player makes sequential decisions about capabilities for extraction, transformation, and integration for data source categories. Extraction involves selecting data sources and transporting data to a staging area. Transformation involves increasing data quality through operations on individual data sources. Integration combines data from different sources, matching and consolidating common data.

To manage complexity from a large number of data sources, Emerge2Maturity groups data sources into categories by levels of technology, complexity, and size. Technology ranges from legacy systems to modern systems depending on currency of

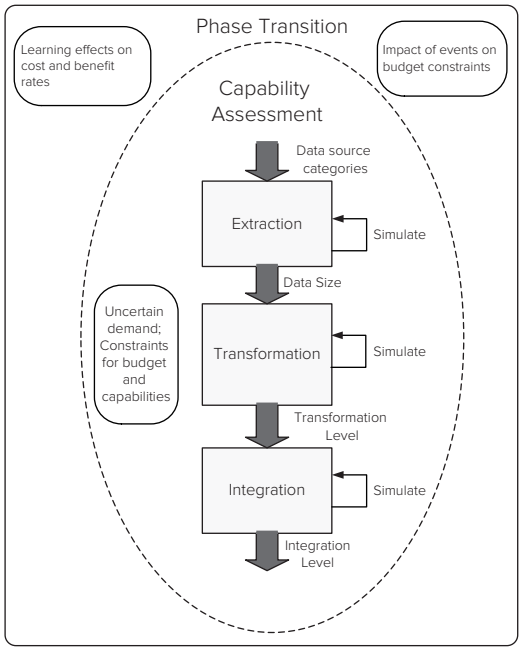


FIGURE 12.13
Game Flow in Emerge2Maturity

programming environment, DBMS, operating system, and hardware platform. Complexity involves difficulty to transform diverse data with more complex data requiring extensive time and effort to analyze. Size involves processing effort for data with larger data requiring additional capacity to process.

Categories determine cost and benefits for individual data sources as all data sources in a category have the same feature values. Feature values determine levels of fixed costs, variable costs, production (number of queries produced), benefits, and risk as depicted in Table 12-7. For example, legacy technology involves higher fixed costs and complexity influences production, variable costs, benefits, and risks.

Transition among phases involves new constraint levels about budgets and capabilities, a learning effect revising rates for costs and benefits, and random economic events influencing budget constraints. Emerge2Maturity determines constraint levels dynamically based on organizational strategy and capabilities achieved in previous phases. The learning effect progresses over phases, affecting rates for costs and benefits. Emerge2Maturity adjusts rates for costs and benefits according to a parameter in a learning function. As an organization acquires capabilities, costs decrease and benefits increase. Emerge2Maturity uses external economic events (recession or expansion) with small probabilities of occurrence. If an event occurs, Emerge2Maturity randomly adjusts the budget constraint, increasing the budget constraint for an expansion and decreasing the budget constraint for a recession.

Game Implementation and Demonstration Emerge2Maturity uses a web interface dependent on common JavaScript packages and an external database. The web interface controls player interactions and provides documentation to explain play aspects. The external database contains static tables with the configuration of a game (number of phases, category features, constraint levels, and other details) and dynamic tables with game play results.

To begin game play, a player chooses a game and begins in phase 1. Emerge2Maturity provides games with several skill levels based on number of phases, constraint levels, and category features. At the beginning of a phase, Emerge2Maturity displays constraint levels (budget and resource levels for data source categories) and features of data source categories. Figure 12.14 shows three categories with constraints limiting each category to a maximum of 5 data sources, 30% transformation level, and 30% integration level. The feature table, below the constraint table, shows the levels of technology, complexity, and size for each category as well as the maximum number of data sources. For example, category 1 has high technology, medium complexity, high size, and 20 data sources.

In phase simulation for extraction, a player selects the number of data sources for each category that maximizes expected profit (Figure 12.15). For each choice, a player uses the simulation button to observe potential results from an uncertain demand. A player has a small number of simulation attempts before committing an answer. After committing an answer, Emerge2Maturity displays costs (expected and optimal) and profit (optimal, expected, and simulated) in bar graphs. Figure 12.15 shows committed choices of 5 data sources for category 1, 3 data sources for category 2, and 4 data sources for category 3. A player then continues to the transformation and integration decisions.

TABLE 12-7
Influence of Feature Levels on Development Variables

	Data Source Feature		
Development Variable	**Technology**	**Complexity**	**Size**
Production level		✓	✓
Fixed cost	✓		✓
Variable cost		✓	✓
Benefit rate		✓	✓
Risk level		✓	✓

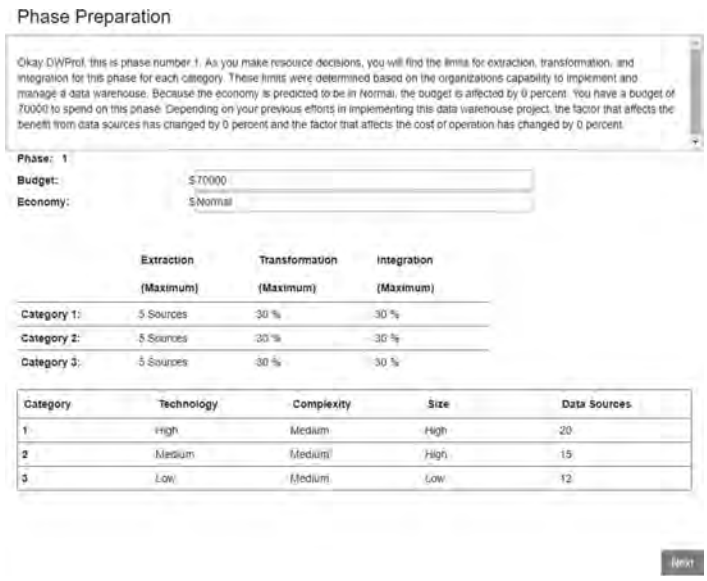


FIGURE 12.14
Phase 1 Preparation in Emerge2Maturity

At the end of a phase, Emerge2Maturity saves a player's decisions and outcomes and then initiates the next phase. The Phase Summary page shows expected costs and profits based on a player's choices for capabilities for each data source category. As a reference, the Phase Summary page also shows the optimal costs and profits. Figure 12.16 shows a good result with expected profit from player choices as $21,348.05 compared to optimal profit of $21,490.15. For more detail, the Phase Summary page decomposes costs and profits by category and decision, showing both expected results from choices and optimal results.

At the end of a game, Emerge2Maturity calculates a numeric score based on the difference between a player's total profit and the optimal total profit. Emerge2Maturity converts the profit difference to a qualitative score displayed on a five-point scale as shown in Figure 12.17. In addition, Emerge2Maturity ranks players by score and displays highest scores in a leaderboard. Emerge2Maturity uses points and a leaderboard to reward players for their accomplishments and encourage additional play.

As this demonstration indicates, Emerge2Mature provides a simulated, educational experience about management of data warehouse development. Players focus

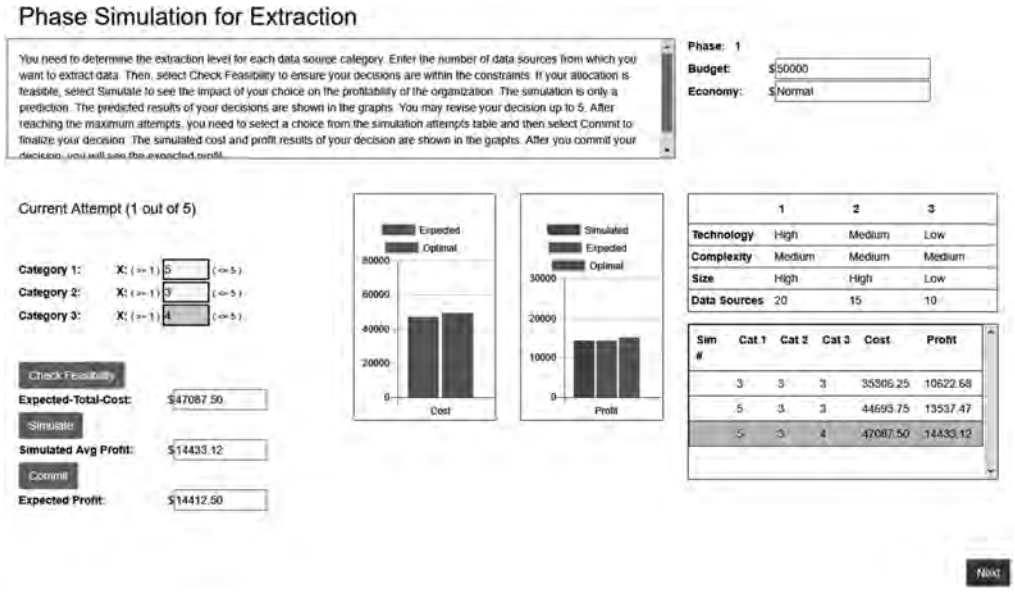


FIGURE 12.15
Phase Simulation for Extraction Decisions in Phase 1

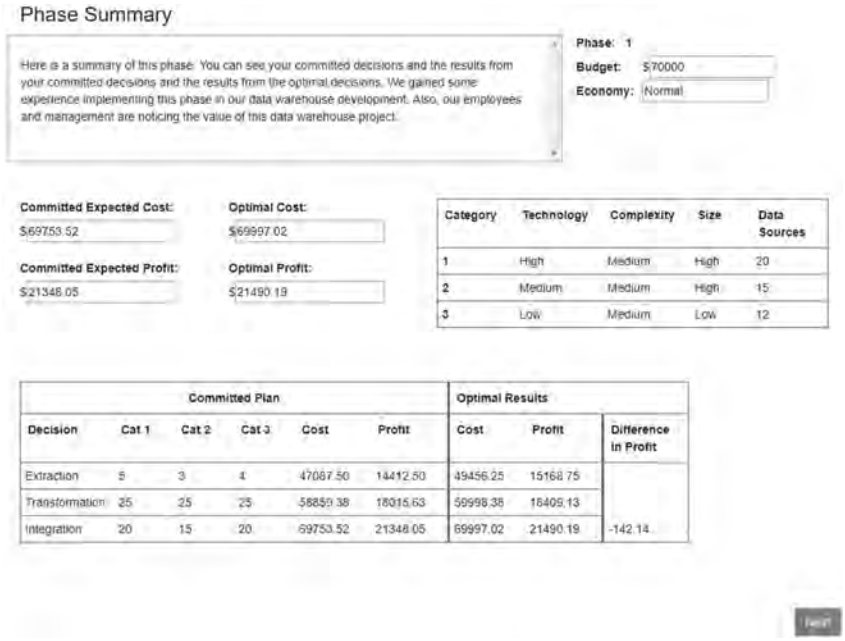


FIGURE 12.16
Phase Summary for Decisions in Phase 1

FIGURE 12.17
Game Score and Leaderboard

on data sources grouped by important features for technology, complexity, and size. For data source categories, players manipulate capabilities for three related decisions in data warehouse development (extraction, transformation, and integration). Simulation allows players to observe impacts of a limited number of choices. Phase results compare player choices for capabilities with optimal choices. In transition among phases, players observe a learning effect, strategy changes for capability and budget constraints, and impact of external events. A simple point system and leaderboard provide incentives to improve and compete with other players.

Although Emerge2Maturity was not developed as a tool for actual management of data warehouse development, it substantially extends abstract concepts traditionally taught about maturity. The simulated game experience engages players, stimulating them to think carefully about difficulties and trade-offs with data warehouse development in organizations over decision-making phases.

12.3 DATA WAREHOUSE EXAMPLES

To elaborate on characteristics and management concepts presented in the first two sections, this section presents business aspects of data warehouses in key industries. Specifically, this section elaborates about data warehouse usage and characteristics beyond summaries presented in Section 12.1.3. You will learn about business entities and historical facts maintained in data warehouses for retail, education, and health care. These industries have contrasting environments with retail dominated by large, tightly integrated firms, K-12 education featuring autonomous school districts subject to compulsory cooperation by higher levels of government, and health care influenced by complex regulations and data standards imposed by third-party payers on independent providers. For perspective about data warehouse maturity, you will learn about development history and organizational units using these data warehouses.

12.3.1 Data Warehouses in Retail

The retail industry pioneered data warehouse usage. Wal-Mart first deployed a data warehouse in 1990 to support decision-making about historical sales data. The retail industry now has decades of experience with data warehouse usage to support vital decision making about store site selection, product mix, supply chain management, product pricing, and vendor management. As recognition of demand in the retail industry, major DBMS vendors (such as Oracle, IBM, and TeraData) provide customized data warehouse solutions for the retail industry.

To depict details of data warehouse usage in the retail industry, this section covers business requirements for the TPC-DS Benchmark[1], a major data warehousing benchmark developed by the Transaction Processing Performance Council (TPC). The nonprofit TPC develops benchmarks for transaction processing and business intelligence processing. The TPC has wide membership and contributions from vendors providing DBMSs, hardware, and enterprise software as well as major professional associations. This section focuses just on the business requirements for the TPC-DS Benchmark, not detailed technical specifications.

The TPC-DS Benchmark represents business intelligence requirements of a retail firm offering products in geographically disbursed stores, online through web shopping, and printed catalogs. Operational systems used by the retail firm maintain customer sales and returns from these sales channels. The retail firm also has systems to manage inventory levels, modify prices according to promotions, create dynamic web pages based on customer profiles, and manage customer profiles using customer relationship management software.

The TPC-DS Benchmark involves processing for transforming data sources and query execution to support business analysts in decision-making tasks. Figure 12.18 depicts major data sources and processing to support requirements in the TPC-DS Benchmark. Three major data sources (Store, Web, and Catalog) provide transaction data about retail sales in physical stores, online, and printed catalogs. Two secondary data sources (inventory and promotions) provide details about inventory levels and promotion offerings. The data integration process transforms and integrates these data sources and loads them into the data warehouse. The data warehouse supports ad hoc queries, repetitive reports, iterative queries showing relationships and trends, and queries supporting data mining analysis.

The data warehouse in the TPC-DS Benchmark maintains data about important business entities and historical facts related to sales, inventory maintenance, promotions, and managing customer profiles. Figure 12.19 provides a simplified representation of major types of business entities and historical facts in the data warehouse. For the most important business entity, customer, the data warehouse maintains contact

[1] TPC-DS Benchmark is a trademark of the Transaction Processing Performance Council.

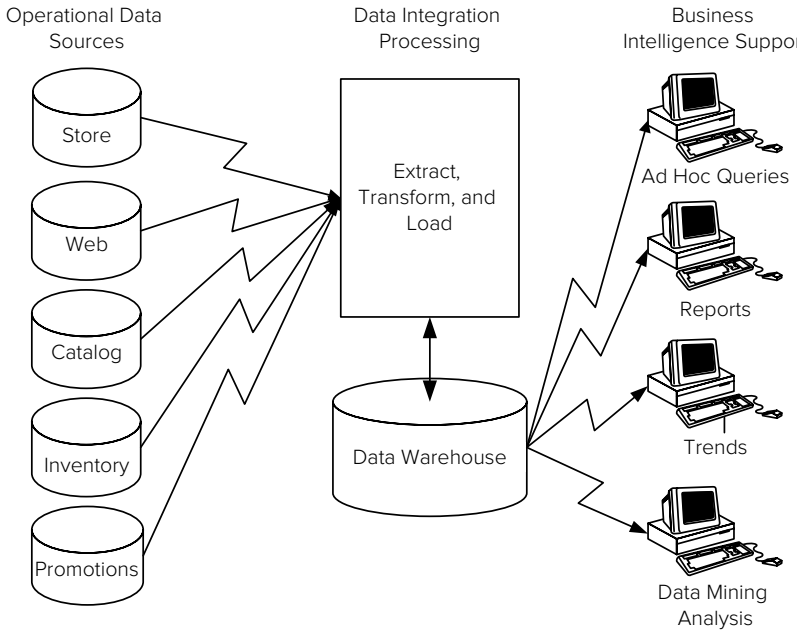


FIGURE 12.18
Components of the TPC-DS Benchmark[2]

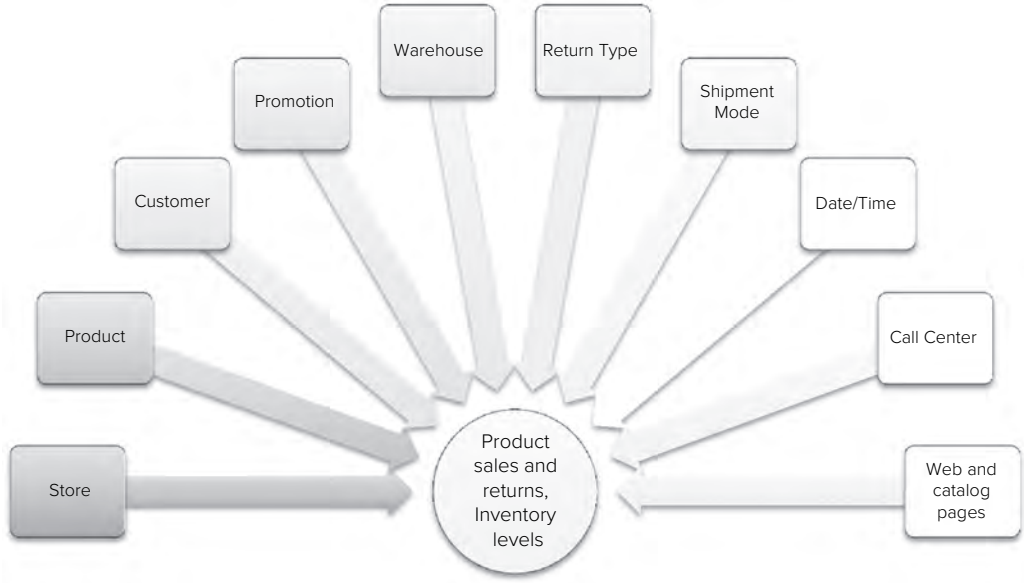


FIGURE12.19
Types of Business Entities and Historical Facts in the Retail Data Warehouse

details, addresses, and household buying demographics. For historical facts, the data warehouse maintains product sales in each channel (store, web, and catalog), returns in each channel (store, web, and catalog), and inventory levels. For each major type of historical fact, the data warehouse maintains many quantitative variables, not just basic values for the level of sales, returns, and inventory.

12.3.2 Data Warehouses in Education

Learning standards substantially influence compulsory K-12 education in many countries. In the USA, the federal government and state governments impose standards and regulations on local school districts. School districts in USA states operate

[2] Adapted from TPC Benchmark DS, Standard Specification, Transaction Processing Performance Council, June 2017.

autonomously with locally elected school boards setting policies and rules. Learning standards require local school districts to report on student performance to facilitate comparison of school districts, improve learning outcomes, and foster innovation in instruction. Individual USA states often impose reporting requirements, while the federal government requires reporting for grant eligibility.

This section describes features of the Colorado Education Data Warehouse to depict data warehouse usage in K-12 education. The Colorado Education Data Warehouse supports reporting of student assessments and growth in K-12 schools in Colorado. Assessments of student achievement provide evidence of the current status of student knowledge and understanding. However, measurement of learning requires assessment of growth in achievement over time, not just the current status of knowledge.

Colorado and other states have invested in systems to support assessment of student growth. In 1997, the Colorado legislature passed a law that required development of accountability systems for K-12 education. The Colorado legislature extended this law in 2004 and 2007 to measure student growth, not just student achievement at a point in time. Figure 12.20 depicts a timeline of accountability laws and data warehouse projects in Colorado in response to the accountability laws.

As a response to the 2007 law, the Colorado Department of Education extended its Education Data Warehouse and developed web portals to support assessments of student growth. The original Education Data Warehouse was developed beginning in 2002 to support education accountability. The student growth extension, performed from 2007 to 2009, had a budget of $6.7 million. As a result of the development since 2002, the Education Data Warehouse is in a mature state with data governance policies, processes and standards to manage the flow of data from capture to use. Data stewards provide data quality audits as part of the ongoing monitoring of data quality facilitated by master data management technology.

The Colorado Department of Education embarked on two major expansion projects in 2013. The Data Warehouse Expansion Project added data from preschool to career and extended security and accuracy in teacher and student data reporting. The Data Pipeline Project supported efficient transfer of data from school districts to the education data warehouse. The Data Pipeline reduced data redundancy, captured data with less time lag, created transaction interchanges to streamline the data collection process, and supported exchange of data on transferred students.

Colorado SchoolView™ (www.schoolview.org) is a public portal that uses the Education Data Warehouse. SchoolView supports visual analysis of student growth on the Colorado Student Assessment Program (CSAP[3]) tests for all Colorado school

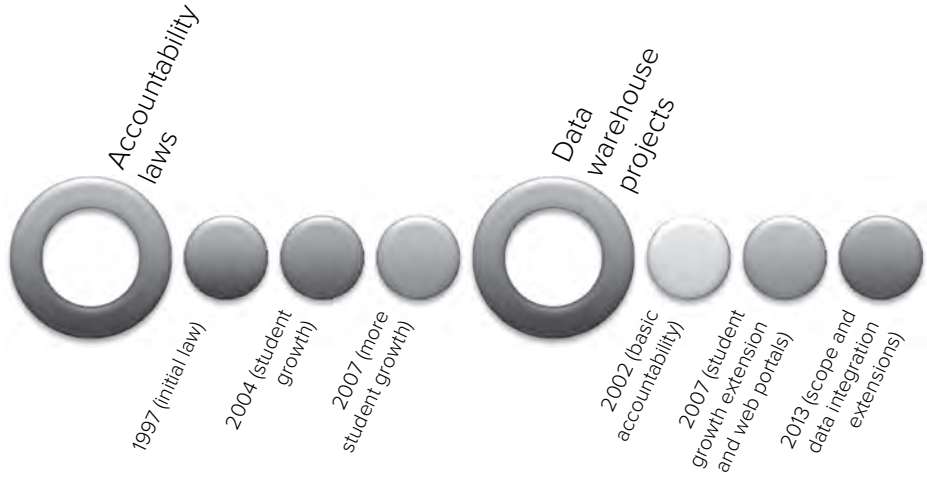


FIGURE 12.20
Timeline of Accountability Laws and Data Warehouse Projects in Colorado

[3] In 2011, the Colorado Department of Education replaced CSAP with the Traditional Colorado Assessment Program (TCAP). The official name of the testing standard is TCAP/CSAP.

districts. Users can compare median student growth in reading, writing, and math by school as depicted in Figure 12.21. In addition, users can search on the dimensions of student group, grade, and ethnicity as well as rolling up to school districts. Figure 12.22 displays visual results for the student group dimension. By selecting a bubble in the display, users can drill down to the individual member value. SchoolView provides a map interface in addition to standard searching tools for selecting schools and school districts.

The SchoolView portal serves parents, teachers, school administrators, and government agencies. Parents use the portal as a decision aid for school enrollment of their children. Colorado permits parental choice in enrollment although school district boundaries may restrict the choice somewhat. School boards at the school district level use SchoolView to make decisions about resource allocations to individual schools and possible remedial actions for individual schools. Teachers and school administrators need aggregate and individual student data to decide on program effectiveness and student performance. Government agencies use SchoolView in decisions about education policy. In particular, the USA federal "No Child Left Behind" law has requirements fulfilled by SchoolView. The public part of SchoolView serves parents and the public. Educators and government agencies have access to the private part of SchoolView.

The primary data sources for the data warehouse are achievement test (CSAP) scores. Students take CSAP tests once per year in the second half of the school year. CSAP scores are maintained by the Colorado Department of Education independent of usage in SchoolView. School districts provide data for student grades and demographic

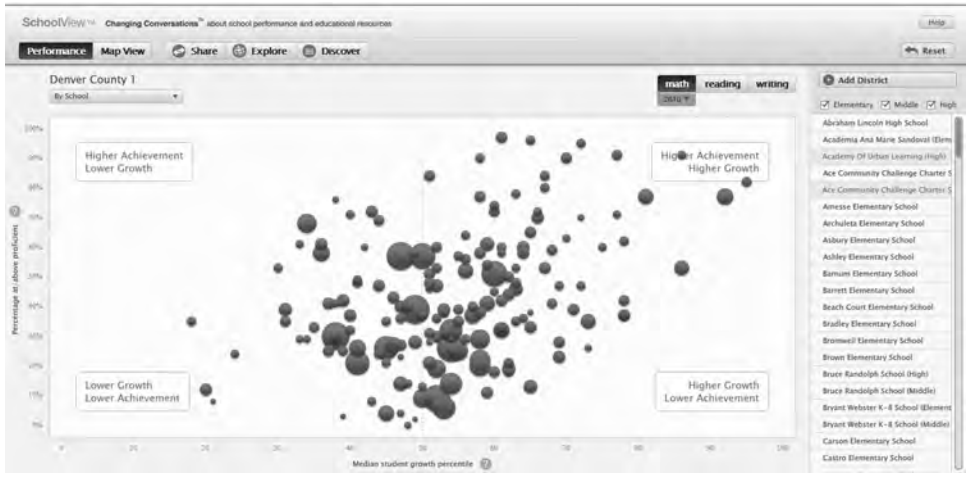


FIGURE 12.21
SchoolView Window with Visual Display of Math Results for Individual Schools

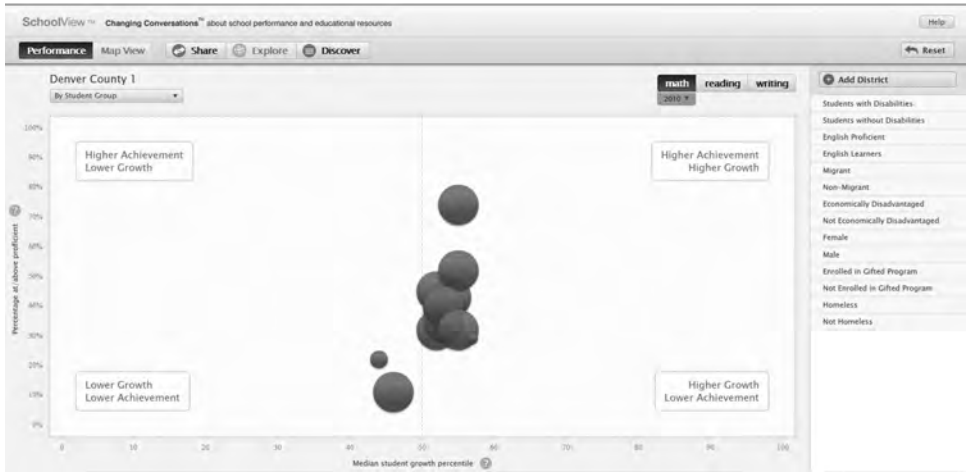


FIGURE 12.22
SchoolView Window with Visual Display of Math Results for Student Groups

attributes. Colorado has 178 school districts so the number of data sources provided by school districts is large. Figure 12.23 shows other data sources provided by higher education, public safety, corrections, early childhood development, and human services.

This background on the Education Data Warehouse should provide insight about the difficulty to develop and operate a data warehouse for highly decentralized organizations. The Education Data Warehouse has evolved over 15 years of development with initial development in 2002, followed by major extensions since 2007, along with plans for more extensions. The development effort required coordination among many areas of state and local government in Colorado. Coordination was especially difficult because each area of government involves elections with political agendas influencing requirements and funding levels. The operation of the data warehouse involved many new policies for data quality.

12.3.3 Data Warehouses in Health Care

Third-party payers, both government and private insurance, substantially influence the health care industry. Third-party payers, in conjunction with industry, government agencies, and health care organizations, have devised detailed coding standards for diagnosis, procedure, and drug events. The World Health Organization maintains the most prominent coding standard known as the International Classification of Diseases (ICD). ICD-10, the current standard in the USA, contains about 68,000 codes, a large increase from the previous version (ICD-9) containing about 13,000 codes. Third-party payers mandate usage of ICD codes (along with procedure and drug codes) by health care providers for reimbursement. Mandated usage of medical codes puts a heavy burden on health care providers, typically requiring specialized staff to assist in choosing appropriate codes.

Medical codes are important parts of electronic medical records. An electronic medical record is a digital version of a patient's medical history chart. Electronic medical records contain lab results, diagnoses, medications, treatment plans, immunization details, hospital stays, digital images, and health care provider notes. Electronic medical records provide the promise of improved patient care, better care coordination, practice efficiencies, increased patient participation, more accurate diagnoses, and reduced costs. However, health care providers bear much of the cost of data collection, but only see indirect benefits. Thus, health care providers require incentives and mandates to collect details of electronic medical records.

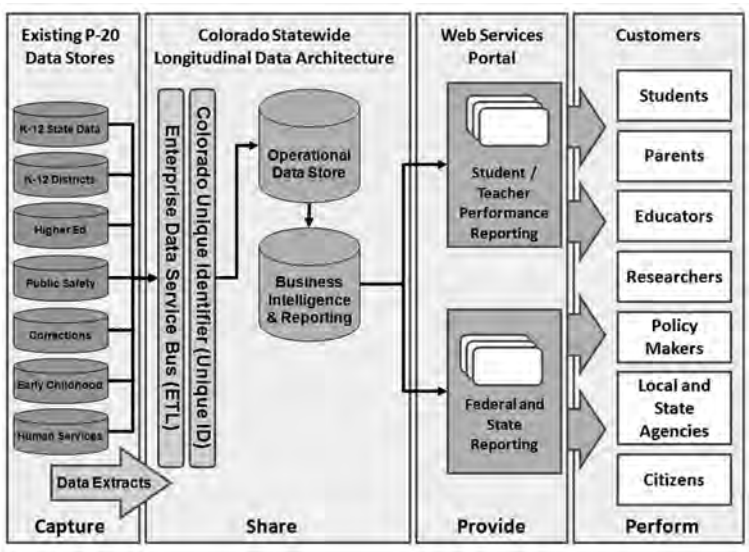


FIGURE 12.23
Enterprise Architecture for the Extended Education Data Warehouse[4]

[4] Source of Figure 12.23: 2009 Colorado SLDS Application Grant Proposal

As an outgrowth of electronic medical records, standards for data warehouses have been developed. The Observational Medical Outcomes Partnership (OMOP), a public/private partnership, developed a prominent data warehouse standard for electronic medical records. The collaborative OMOP effort developed a standard vocabulary and data model along with a suite of tools for data integration, query formulation, and data generation. Most of these tools are available through open source software licenses on the OMOP website (omop.org).

The OMOP Common Data Model provides a detailed specification for a medical data warehouse. Figure 12.24 provides an overview of the Common Data Model. The Common Data Model contains business entities for persons (typically patients), providers, cohorts, locations, organizations, care sites, payer plans, and time. The Common Data Model tracks historical facts about treatments involving drugs or procedures and event occurrences involving observations, conditions, deaths, provider visits, or exposures.

The OMOP Common Data Model and associated tools provide a foundation for development of medical data warehouses. The Scalable Architecture for Federated Translational Inquiries Network (SAFTINet), a multi-state collaboration of health care providers, developed a data warehouse conforming to a subset of the OMOP standard. SAFTINet promises economies of scale for research studies about comparative effectiveness. Comparative effectiveness research aims to reduce variation of treatments across patient populations and conditions. Studies emphasizing comparative effectiveness require pragmatic trials from diverse clinical settings and patient populations. Data warehouse usage promotes economies of scale, providing shared data across studies instead of each study collecting its own data.

To support research studies using a medical data warehouse, SAFTINet extended the OMOP tool set and populated a subset of the OMOP standard with data from clinical health care practices and USA Medicaid claims data. SAFTINet supports external users, separate from users of partner data sources as depicted in Figure 12.25. Partners voluntarily contribute data sources including electronic medical records, claims data, and administrative data. Data transformation components convert partner data sources into data marts conforming to a subset of the OMOP Common Data Model. No data integration occurs in the data transformation level. The Query System processes queries, combining data from multiple data marts if necessary. The Data Mart Bus integrates patient data across data marts if required in a query.

SAFTINet data marts conform to a subset of the OMOP Common Data model as shown in Figure 12.26. SAFTINet focuses on underserved patient populations so

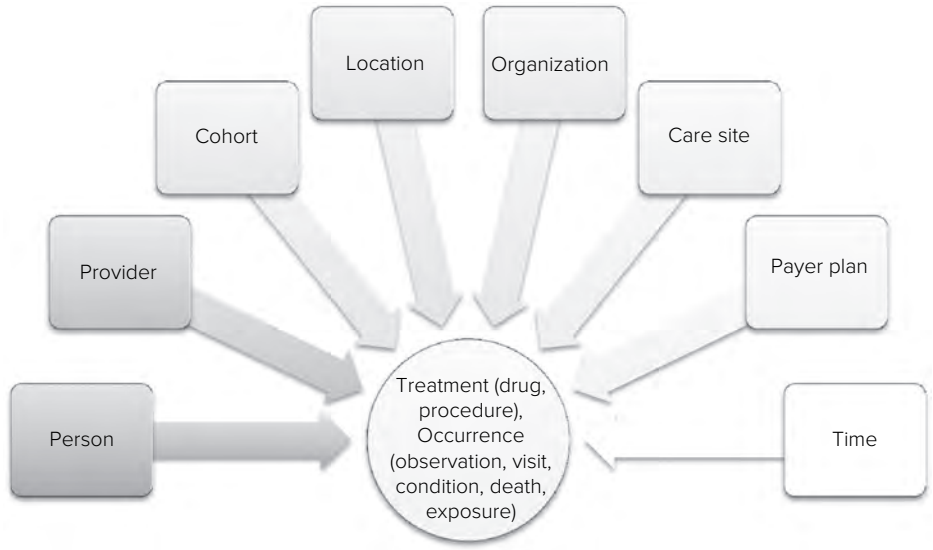


FIGURE 12.24
Overview of the OMOP Common Data Model

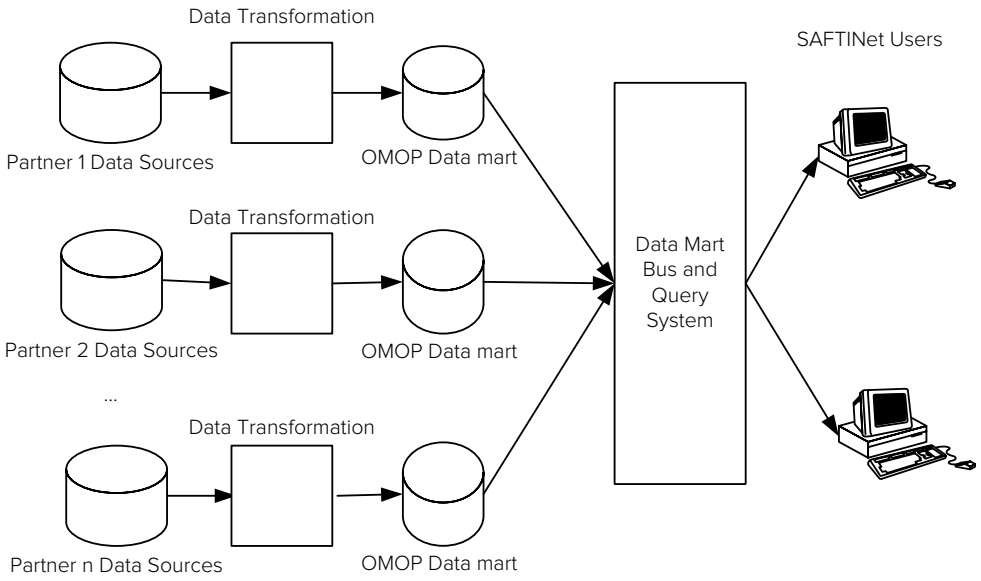


FIGURE 12.25
SAFTINet Architecture

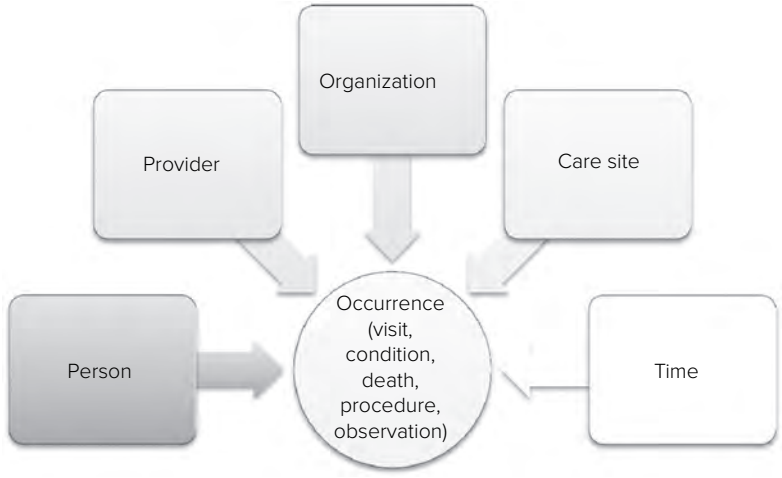


FIGURE 12.26
Overview of the SAFTINet Data Model

some parts of the Common Data Model are not relevant. In addition, SAFTINet faced substantial barriers to participation by independent health care practices and limited development resources. Thus, SAFTINet data marts use a subset of the OMOP Common Data Model due to the focus of SAFTINet, participation barriers, and limited resources.

SAFTINet has demonstrated the potential of data warehouse standards to promote health care research. SAFTINet contains diverse participation by 14 health care organizations, 100 primary care practices, 500 health care providers, and more than 400,000 patients in three USA states. To facilitate voluntary participation, SAFTINet established governance committees. Usage of SAFTINet in research has been hampered by internal review boards, strict privacy regulations, and approval of data providers. These usage restrictions are common in medical information systems so SAFTINet users face similar challenges as users of electronic medical records.

CLOSING THOUGHTS

This chapter presented conceptual background about data warehouses in preparation for detailed skills emphasized in Chapters 13 to 15. Data warehouses, logically centralized repositories of transformed data from operational databases and external data sources, have become critical infrastructure for many organizations. The first part of this chapter contrasted data requirements for transaction processing and business intelligence processing. These differing requirements influenced organizations to deploy data warehouses and DBMS vendors to develop new features for data warehouses. You learned about characteristics and applications of data warehouses supporting business intelligence requirements for organizations.

Despite importance of business intelligence requirements, organizations struggle to realize substantial value from data warehouses because of development difficulties. The second part of this chapter covered management concepts important for deploying data warehouses in organizations. You learned about development difficulties and learning curve concepts, architectures for data warehouses, and maturity concepts for realizing business value from data warehouses over time. To provide insights about data warehouse development in organizations, you learned concepts underlying a business strategy game providing simulated experience with data warehouse development.

The final part of the chapter provided examples of data warehouses used in key industries, providing a context for conceptual background in the first two parts. You learned about characteristics of the retail, education, and health care industries and data warehouse solutions to support organizations in these industries.

The concepts and data warehouse examples presented in this chapter provide a foundation for development of detailed skills emphasized in the other chapters of part 6. You will learn key skills pertaining to conceptual design of data warehouses (Chapter 13), data integration (Chapter 14), and query formulation (Chapter 15).

REVIEW CONCEPTS

- Data warehouse, a logically centralized repository of transformed data from operational databases and external data sources
- Technology and deployment limitations of operational databases for business intelligence processing: performance limitations, lack of integration and missing DBMS features
- Data needs for transaction processing versus business intelligence applications
- Characteristics of a data warehouse: subject-oriented, integrated, time-variant, and nonvolatile
- Comparison of operational databases and data warehouses on currency, detail level, design orientation, rows per retrieval, normalization level, modification level, and data representation
- Data mining, a key application of data warehouses across industries
- Development challenges with data warehouses involving substantial coordination efforts, uncertain levels of data quality, and intangible benefits
- Characteristics of learning curves and application for business value and transformation cost in data warehouse development projects
- Architectures for deploying a data warehouse: enterprise data warehouse, data mart, data mart bus, and federated data warehouse
- Enterprise data model representing the conceptual structure of a data warehouse and descriptive data to access and transform data sources
- Data mart representing a view of a data warehouse at a business unit level or business analysis area

- Architecture selection focusing on integration level and data warehouse scope
- Factors influencing architecture selection: resource constraints, information technology skills, integration need, sponsorship, strategic view, and urgency
- Staging area to provide temporary storage of transformed data before loading into a data warehouse
- Stages of the Data Warehouse Maturity Model (infant, child, teenager, adult, and sage) and difficulty of moving between some stages (infant to child and teenager to adult)
- Capability assessment to attain maturity level by reaching levels of process areas and activities
- Key features of Emerge2Maturity, a business strategy game providing a simulated experience of data warehouse development for an organization
- Historical usage of data warehouses in the retail industry
- Major data sources and types of business entities and historical facts for the data warehouse used in the TPC-DS Benchmark
- Influences on K-12 education for development of data warehouses: learning standards, independent school districts, and laws mandating reporting about student achievement
- Data sources and users of the Colorado Education Data Warehouse
- Influences on health care organizations for development of data warehouses: third-party payers, detailed coding schemes for medical events, and data collection difficulties
- Standards for medical data warehouses: Common Data Model of the Observational Medical Outcomes Partnership (OMOP)
- SAFTINet, a data warehouse supporting medical research studies, designed as a subset of the OMOP Common Data Model with voluntary participation by independent medical practices

QUESTIONS

1. What is the cause of poor performance when using operational databases for business intelligence?
2. How does lack of integration inhibit usage of operational databases for business intelligence?
3. What missing features inhibit usage of traditional DBMS technology for business intelligence?
4. Explain subject-orientated as a defining characteristic of a data warehouse.
5. Explain integrated as a defining characteristic of a data warehouse.
6. Explain time-variant as a defining characteristic of a data warehouse.
7. Explain non-volatile as a defining characteristic of a data warehouse.
8. Compare operational databases and data warehouses on currency, detail level, design orientation, rows per retrieval, normalization level, modification level, and data representation.
9. Compare operational databases and data warehouses on normalization and modification levels.
10. Compare operational databases and data warehouses on data representation.
11. How did commercial software vendors respond to missing features in DBMSs for business intelligence?
12. Why do organizations undertake data warehouse projects?

13. What requirements of data mining require extensions to capabilities required by a mature data warehouse?
14. Why has data mining emerged as a key application of data warehouses across industries?
15. What factors contribute to potentially high and unforeseen costs in data warehouse projects?
16. What are intangible benefits and how do intangible benefits affect data warehouse projects?
17. What is a learning curve for skill improvement?
18. What is a learning curve for production?
19. How is the business value learning curve for data warehouses related to the learning curve for skill attainment?
20. How is the data transformation learning curve related to the learning curve for production?
21. What are important factors in data warehouse architectures?
22. What is the enterprise data warehouse architecture?
23. What are the components of an enterprise data model?
24. How does a data warehouse differ from a data mart?
25. What is a staging area in the enterprise data warehouse architecture?
26. What is the data mart architecture?
27. What is the data mart bus architecture?
28. What is the federated data warehouse architecture?
29. What is an oper mart?
30. What are the two opposing approaches that provide the limits of data warehouse architectures?
31. What factor provides bias for data warehouses of smaller scope and lower integration level?
32. How can organizations overcome bias for data warehouses of smaller scope and lower integration level?
33. According to Choudary (2010), what factors should be high for an organization to select the enterprise data warehouse architecture?
34. According to Choudary (2010), what factors should be high for an organization to select the data mart architecture?
35. According to Choudary (2010), when should an organization select the data mart bus architecture?
36. What is the purpose of the data warehouse maturity model?
37. What is an important insight provided by the data warehouse maturity model?
38. What are the five maturity levels in the Capability Maturity Model?
39. How does an organization reach the optimizing level in the data warehouse maturity model proposed by Sen et al.?
40. What difficulties do students have to understand data warehouse development in organizations?
41. How does Emerge2Maturity address learning difficulties about data warehouse development?
42. What decisions does a player make in Emerge2Maturity?
43. How does Emerge2Maturity manage complexity about a large number of data sources?
44. How and when do learning effects occur in Emerge2Maturity?

45. What random elements are used in Emerge2Maturity?
46. Provide a brief summary of the game playing experience in Emerge2Maturity.
47. What are the business intelligence requirements in the TPC-DS Benchmark?
48. What data sources are used in the TPC-DS Benchmark?
49. What types of business entities and historical facts are used in the TPC-DS Benchmark?
50. What factors drive data warehouses used in K-12 education?
51. What data sources are used in the Colorado Education Data Warehouse?
52. What groups use the Colorado Education Data Warehouse through the SchoolView portal?
53. What data standards are mandated by third-party payers in health care?
54. What is the Observation Medical Outcomes Partnership (OMOP)?
55. What is the OMOP Common Data Model?
56. What is SAFTINet? How is SAFTINet related to OMOP?

PROBLEMS

Due to the focus on characteristics and concepts of data warehouses, this chapter only contains a small set of open-ended problems about major concepts. Other chapters in part 6 contain a much larger set of detailed problems on skill development rather than concept application.

1. For a data warehouse of a large retail firm, what architecture seems appropriate? You should use the TPC-DS Benchmark as a model for a retail data warehouse. Try to apply the architecture selection factors discussed in Section 12.2.3. If you want more details beyond provided in this chapter, you can investigate retail data warehouses used by major retailers such as Walmart, Amazon, and Target. For Walmart, the references at the end of the chapter contain a detailed case study about Walmart.
2. For the Colorado Education Data Warehouse, analyze the architecture selection factors discussed in Section 12.2.3. Do you think that school districts in Colorado have data warehouses separate from the Colorado Education Data Warehouse? How does the architecture of the Colorado Education Data Warehouse differ from the architecture of a large retail firm?
3. For SAFTINet, analyze the architecture selection factors discussed in Section 12.2.3. Do you think that health care practices and organizations participating in SAFTINet have data warehouses separate from SAFTINet? Compare the architecture selection factors of SAFTINet and the Colorado Education Data Warehouse. Would you expect more participation from independent school districts than independent health care organizations?
4. Investigate an organization in another industry. Write a brief case study with background about the industry, factors influencing data warehouse architecture selection, and details about data sources, users, and types of business entities and historical facts in the data warehouse. Analyze the architecture selection factors and explain the architecture chosen.
5. If your instructor has installed Emerge2Maturity, play some games and discuss your experience. Do you understand the relationship of features of data source categories to costs and benefits for extraction, transformation, and integration? Do you understand the sequential nature of decisions for extraction, transformation, and integration? Explain the learning effects for costs and benefits as a game progresses? Did you enjoy playing Emerge2Maturity? What insights about data warehouse development have you gained?

REFERENCES FOR FURTHER STUDY

Several references provide additional details about important parts of Chapter 12. Kimball (2003a) expounds on the data mart bus architecture presented in Section 12.2.2. Eckerson (2007) provides more details about the Data Warehouse Maturity Model covered in Section 12.2.3, while the TDWI website (tdwi.org) provides details about the Business Intelligence Maturity Model. Choudary (2010) provides details about architecture selection factors summarized in Section 12.2.3. Sen et al. (2012) describe the capability assessment model summarized in Section 12.2.3. Westerman (2000) provides a detailed case study on data warehousing development at Walmart. The case study, although somewhat dated, is a fascinating look into business practices and information technology development at Walmart, a major innovator in retail information technology. The Transaction Processing Performance Council (tpc.org) provides details about the TPC-DS Benchmark covered in Section 12.3.1. Shilling et al. (2013) presents architectural details of SAFTINet, extending the presentation in Section 12.3.3.

chapter

13

Conceptual Design of Data Warehouses

Learning Objectives

This chapter explains data representation and design practices for data warehouses. After this chapter, the student should have acquired the following knowledge and skills:

- Explain terminology and operators of the multidimensional data model for data cubes
- Apply relational data modeling patterns to multidimensional data
- Analyze data warehouse designs for summarizability problems and historical integrity
- Apply steps of the schema integration process to a small number of data sources
- Determine the grain for a data warehouse design
- Gain insights about methodologies for enterprise data warehouse development

OVERVIEW

After gaining background about concepts and management practices for data warehouses, you are ready to learn detailed skills. This chapter emphasizes data representation and design practices for conceptual design of data warehouses, important skills for data warehouse professionals.

Your learning in this chapter begins with data representation for data warehouses. Organizations typically use two representations of data warehouses for business analysis and relational DBMS implementation. The first section presents the multidimensional data model, a representation typically used by business analysts. You will learn concepts of data cube representation and manipulation. The second section covers star schemas, a representation of data cubes in a table design. You will also learn about table design extensions for historical integrity and dimension representation.

After background about data representation, this chapter emphasizes design practices to apply data representations. The third section presents design practices to identify and resolve summarizability problems. Summarizability is a key design issue for data warehouses, ensuring that query results have predictable outcomes for business analysts. You will learn about summarizability problems and patterns for dimensions and dimension-fact relationships. The fourth section provides a broader view of design practices with coverage of the schema integration process to combine schemas from

multiple data sources. You will learn the steps of the schema integration process, determination of the grain of a data warehouse design, and design methodologies for enterprise data warehouse development.

This chapter provides a foundational skill set for data warehouse professionals. Chapters 14 and 15 extend your skill set with detailed skills about data integration and query formulation for enterprise data warehouses. Collectively, the chapters in Part 6 provide breadth and depth of coverage to support career aspirations as a data warehouse or business intelligence professional.

13.1 MULTIDIMENSIONAL REPRESENTATION OF DATA

The multidimensional data model supports data representation and operations specifically tailored for business intelligence processing in data warehouses. The multidimensional data model was originally proposed as a replacement for the relational model for data warehouses. Over time, the multidimensional model has evolved into a business analyst model, complementing the relational model for data warehouse storage.

This section describes terminology and operations for the multidimensional data model. The presentation emphasizes conceptual properties, important for data warehouse design. For a query formulation perspective, Chapter 15 presents the Multidimensional Expression Language (MDX), initially developed by Microsoft, along with associated graphical tools.

13.1.1 Example of a Multidimensional Data Cube

This subsection begins with an example of a company that sells electronic products in different parts of the USA. In particular, the company markets four different printer products (mono laser, ink jet, photo, and portable) in five different states (California, Washington, Colorado, Utah, and Arizona). To store daily sales data for each product and each location in a relational database, Table 13-1 contains sample data consisting of three columns (*Product*, *Location*, and *Sales*) and 20 rows (four instances of *Product* times five instances of *Location*).

The representation of Table 13-1 can be complex and unwieldy. First, imagine that the company wishes to add a fifth product (say, color laser). To track sales by states for this new product, you need to add five rows, one each for each state. Second, note that the data in Table 13-1 represents sales data for a particular day (for example, August 10, 2017). To store the same data for all 365 days of 2017, you need to add a fourth column to store the sales date, and duplicate the 20 rows for each date 365 times to yield a total of 7,300 rows. By the same token, if you wish to store historic data for a period of 10 years, you need 73,000 rows. Each new row must contain the product, state, and date values.

Table 13-1 contains two dimensions, *Product* and *Location*, and a numeric value for unit sales. Table 13-1 can be conceptually simplified by rearranging the data in a two-dimensional format as depicted in Table 13-2.

The multidimensional representation is simple to understand and extend. For example, adding dates requires a third dimension called *Time*, resulting in a three-dimensional arrangement as shown in Figure 13.1. You can conceptually think of this three-dimensional table as a book consisting of 365 pages, each page storing sales data by product and state for a specific date of the year. In addition, the multidimensional representation is more compact because the row and column labels are not duplicated as in Table 13-1.

The multidimensional representation also provides a convenient representation of summary totals. Each dimension in a data cube can accommodate totals (row totals, column totals, depth totals, and overall totals) that a user can identify easily. For example, to add row totals to Table 13-2, a *Totals* column can be added with one value per row as shown in Table 13-3. In the relational representation as depicted in Table 13-1,

Product	Location	Sales
Mono Laser	California	80
Mono Laser	Utah	40
Mono Laser	Arizona	70
Mono Laser	Washington	75
Mono Laser	Colorado	65
Ink Jet	California	110
Ink Jet	Utah	90
Ink Jet	Arizona	55
Ink Jet	Washington	85
Ink Jet	Colorado	45
Photo	California	60
Photo	Utah	50
Photo	Arizona	60
Photo	Washington	45
Photo	Colorado	85
Portable	California	25
Portable	Utah	30
Portable	Arizona	35
Portable	Washington	45
Portable	Colorado	60

TABLE 13-1
Relational Representation of Sales Data

	Product			
Location	Mono Laser	Ink Jet	Photo	Portable
California	80	110	60	25
Utah	40	90	50	30
Arizona	70	55	60	35
Washington	75	85	45	45
Colorado	65	45	85	60

TABLE 13-2
Two-Dimensional Representation of Sales Data

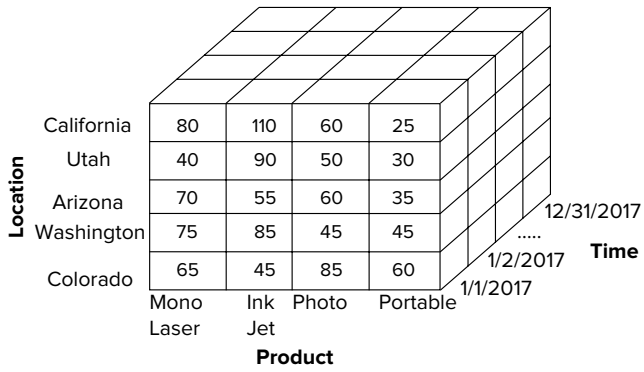


FIGURE 13.1
A Three-Dimensional Data Cube

totals must be added by using null values for column values. For example, to represent the total California sales for all products, the row <–, California, 275> should be added to Table 13-1 where – indicates all products.

TABLE 13-3
Multidimensional Representation of Sales Data with Row Totals

	Product				
Location	**Mono Laser**	**Ink Jet**	**Photo**	**Portable**	**Totals**
California	80	110	60	25	275
Utah	40	90	50	30	210
Arizona	70	55	60	35	220
Washington	75	85	45	45	250
Colorado	65	45	85	60	255

In addition to advantages in usability, the multidimensional representation supports increased retrieval speed. Direct storage of multidimensional data obviates the need to convert from a table representation to a multidimensional representation. However, the multidimensional representation can suffer from excessive storage because many cells can remain empty. Even with compression techniques, large multidimensional tables can consume considerably more storage space than corresponding relational tables.

In summary, a multidimensional representation provides intuitive appeal for business analysts. As the number of dimensions increases, business analysts find a multidimensional representation easy to understand and visualize as compared to a relational representation. Because the multidimensional representation matches the needs of business analysts, this representation is widely used in business intelligence tools even when relational tables provide physical storage.

13.1.2 Multidimensional Terminology

Data Cube
a multidimensional format in which cells contain numeric data called measures organized by subjects called dimensions. A data cube is sometimes known as a hypercube because conceptually it can have an indefinite number of dimensions.

A **data cube** or hypercube generalizes the two-dimensional (Table 13-2) and three-dimensional (Figure 13.1) representations shown in the previous subsection. A data cube consists of cells containing measures (numeric values such as the unit sales amounts) and dimensions to label or group numeric data (e.g., *Product, Location*, and *Time*). Each dimension contains values known as members. For instance, the *Location* dimension has five members (California, Washington, Utah, Arizona, and Colorado) in Table 13-3. Both dimensions and measures can be stored or derived. For example, purchase date is a stored dimension with purchase year, month, and day as derived dimensions.

Dimension Details Dimensions can have hierarchies composed of levels. For instance, the *Location* dimension may have a hierarchy composed of the levels country, state, and city. Likewise, the *Time* dimension can have a hierarchy composed of year, quarter, month, and date. Hierarchies can be used to drill down from higher levels of detail (e.g., country) to lower levels (e.g., state and city) and to roll-up in the reverse direction. Although hierarchies are not essential, they allow a convenient and efficient representation. Without hierarchies, the *Location* dimension must contain the most detailed level (city). However, computing aggregates across dimensions in this representation can be difficult. Alternatively, the *Location* dimension can be divided into separate dimensions for country, state, and city resulting in a larger data cube.

For flexibility, dimensions can have multiple hierarchies. In a dimension with multiple hierarchies, usually at least one level is shared. For example, the *Location* dimension can have one hierarchy with levels for country, state, and city, and a second hierarchy with levels for country, state, and postal code. The *Time* dimension can have one hierarchy with levels for year, quarter, and date and a second hierarchy with levels year, week, and day of the year. Multiple hierarchies allow alternative organizations for a dimension.

Another dimension feature is the ragged hierarchy for a self-referencing relationship among members of the same level. For example, a manager dimension could have

a ragged hierarchy to display relationships among managers and subordinates. An analyst manipulating a data cube may want to expand or contract the manager dimension according to relationships among managers and subordinates.

The selection of dimensions has an influence on the sparsity of a data cube. Sparsity indicates the extent of empty cells in a data cube. Sparsity can be a problem if two or more dimensions are related. For example, if certain products are sold only in selected states, cells are empty. If a large number of cells are empty, the data cube can waste space and be slow to process. Special compression techniques can be used to reduce the size of sparse data cubes.

Measure Details Cells in a data cube contain measures such as the sales values in Figure 13.1. Measures support numeric operations such as simple arithmetic, statistical calculations, and optimization methods. A cell may contain one or more measures. For example, the number of units can be another measure for the sales data cube. The number of nonempty cells in a multidimensional cube should equal the number of rows in the corresponding relational table. For example, Table 13-2 contains 20 nonempty cells corresponding to 20 rows in Table 13-1.

Derived measures can be stored in a data cube or computed from other measures at run-time. Measures that can be derived from other measures in the same cell typically would not be stored. For example, total dollar sales can be calculated as total unit sales times the unit price measures in a cell. Summary measures derived from a collection of cells may be stored or computed depending on the number of cells and the cost of accessing the cells for the computation.

The **aggregation property** indicates allowable summary operations for measures. Additive measures can be summarized across all dimensions using addition. Common additive measures include sales, cost, and profit. For example, the sales measure in Figure 13.1 can be meaningfully summed across locations, products, and time periods. Semi-additive measures can be summarized in some dimensions but not all dimensions, typically not in the time dimension. Periodic measurements such as account balances and inventory levels are semi-additive. For example, account balance can be summed across product and location dimensions but not across time. However, account balances can be averaged across time because the average operation allocates account balances to dimension members. Non-additive measures cannot be summarized in any dimension. Historical facts involving individual entities such as a unit price for inventory items are non-additive. Some non-additive measures can be converted to additive or semi-additive. For example, extended price (unit price * quantity) is additive although unit price is not additive.

Aggregation Property indicates allowable summary operations for measures. Business analysts who do not understand the allowable operations may perform operations that have no meaning. The aggregation property for a dimension can be additive (summarized on all dimensions using addition), semi-additive (summarized on some dimensions using addition), or non-additive (cannot be summarized on any dimension using addition).

Other Data Cube Examples As this section has indicated, data cubes can extend beyond the three-dimensional example shown in Figure 13.1. Table 13-4 lists common data cubes to support human resource management and financial analysis. The dimensions with slashes indicate hierarchical dimensions. The time and location dimensions are also hierarchical, but possible levels are not listed since the levels can be organization specific.

13.1.3 Time-Series Data

Time is one of the most common dimensions in a data warehouse, useful for identifying trends, making forecasts, and so forth. A time series provides storage of all historic data in one cell, instead of specifying a separate time dimension. The structure of a measure becomes more complex with a time series, but the number of dimensions is reduced. In addition, many statistical functions can operate directly on time-series data.

A time series is an array data type with a number of special properties as listed below. The array supports a collection of values, one for each time period. Examples of time-series measures include weekly sales amounts, daily stock closing prices, and yearly employee salaries. The following list shows typical properties for a time series:

TABLE 13-4
Data Cubes to Support Human Resource Management and Financial Analysis

Data Cube	Typical Dimensions	Typical Measures
Turnover analysis	Company/line of business/department, location, salary range, position classification, time	Head counts for hires, transfers, terminations, and retirements
Employee utilization	Company/line of business/department, location, salary range, position classification, time	Full time equivalent (FTE) hours, normal FTE hours, overtime FTE hours
Asset analysis	Asset type, years in service band, time, account, company/line of business/department, location	Cost, net book value, market value
Vendor analysis	Vendor, location, account, time, business unit	Total invoice amount

- *Data Type:* This property denotes the kind of data stored in the data points. The data type is usually numeric such as floating point numbers, fixed decimal numbers, or integers.
- *Start Date:* This property denotes the starting date of the first data point, for example, 1/1/2017.
- *Calendar:* This property contains the calendar year appropriate for the time series, for example, 2017 fiscal year. An extensive knowledge of calendar rules, such as determining leap years and holidays embedded in a calendar, reduces effort in data warehouse development.
- *Periodicity:* This property specifies the interval between data points. Periodicity can be daily, weekly, monthly, quarterly, yearly (calendar or fiscal years), hourly, 15-minute intervals, 4-4-5 accounting periods, custom periodicity, and so on.
- *Conversion:* This property specifies conversion of unit data into aggregate data. For instance, aggregating daily sales into weekly sales requires summation, while aggregating daily stock prices into weekly prices requires an averaging operation.

13.1.4 Data Cube Operators

A number of business intelligence operators have been proposed for data cubes. This section discusses the most commonly used operators. Most business analysis tools support additional operators along with convenient graphical interfaces for all operators. Chapter 15 provides details of graphical tools for data cube manipulation.

Slice Because a data cube can contain a large number of dimensions, users often need to focus on a subset of the dimensions to gain insights. The slice operator retrieves a subset of a data cube similar to the restrict operator of relational algebra. In a slice operation, one or more dimensions are set to specific values and the remaining data cube is displayed. For example, Figure 13.2 shows the data cube resulting from the slice operation on the data cube in Figure 13.1 where Time = 1/1/2017 and the other two dimensions (*Location* and *Product*) are shown.

FIGURE 13.2
Example Slice Operation

	Product			
Location	Mono Laser	Ink Jet	Photo	Portable
California	80	110	60	25
Utah	40	90	50	30
Arizona	70	55	60	35
Washington	75	85	45	45
Colorado	65	45	85	60

(Location × Product Slice for Time = 1/1/2017)

A variation of the slice operator allows a decision maker to summarize across members rather than to focus on just one member. The slice-summarize operator replaces one or more dimensions with summary calculations. The summary calculation often indicates the total value across members or the central tendency of the dimension such as the average or median value. For example, Figure 13.3 shows the result of a slice-summarize operation where the *Product* dimension is replaced by the sum of sales across all products. A new column called *Total Sales* can be added to store overall product sales for the entire year.

Dice Because individual dimensions can contain a large number of members, users need to focus on a subset of members to gain insights. The dice operator replaces a dimension with a subset of values of the dimension. For example, Figure 13.4 shows the result of a dice operation to display sales for the State of Utah for January 1, 2017. A dice operation typically follows a slice operation and returns a subset of the values displayed in the preceding slice. It helps focus attention on one or more rows or columns of numbers from a slice.

Drill-Down Users often want to navigate among the levels of hierarchical dimensions. The drill-down operator allows users to navigate from a more general level to a more specific level. For example, Figure 13.5 shows a drill-down operation on the State of Utah of the *Location* dimension. The plus sign by Utah indicates a drill-down operation.

Roll-Up Roll-up (also called drill-up) is the opposite of drill-down. Roll-up involves moving from a specific level to a more general level of a hierarchical dimension. For example, a decision maker may roll-up sales data from daily to quarterly level for end-of-quarter reporting needs. In the printer sales example, Figure 13.2 shows a roll-up of the State of Utah from Figure 13.5.

Pivot The pivot operator supports rearrangement of the dimensions in a data cube. For example, in Figure 13.1, the position of the *Product* and the *Location* dimensions can be reversed so that *Product* appears on the rows and *Location* on the columns. The pivot operator allows a data cube to be presented in an appealing visual order.

The pivot operator is typically used on data cubes of more than two dimensions. On data cubes of more than two dimensions, multiple dimensions appear in the row and/or column area because more than two dimensions cannot be displayed in other ways. For example, to display a data cube with *Location*, *Product*, and *Time* dimensions,

	Time			
Location	**1/1/2017**	**1/2/2017**	**...**	**Total Sales**
California	400	670	...	16,250
Utah	340	190	...	11,107
Arizona	270	255	...	21,500
Washington	175	285	...	20,900
Colorado	165	245	...	21,336

FIGURE 13.3
Example Slice-Summarize Operation

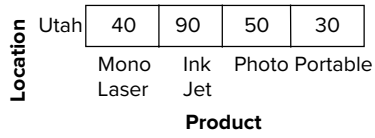


FIGURE 13.4
Example Dice Operation

FIGURE 13.5
Drill-Down Operation for the State of Utah in Figure 13.2

	Product			
Location	**Mono Laser**	**Ink Jet**	**Photo**	**Portable**
California	80	110	60	25
+ Utah				
Salt Lake	20	20	10	15
Park City	5	30	10	5
Ogden	15	40	30	10
Arizona	70	55	60	35
Washington	75	85	45	45
Colorado	65	45	85	60

TABLE 13-5
Summary of Data Cube Operators

Operator	Purpose	Description
Slice	Focus attention on a subset of dimensions	Replace a dimension with a single member value or with a summary of its measure values
Dice	Focus attention on a subset of member values	Replace a dimension with a subset of members
Drill-down	Obtain more detail about a dimension	Navigate from a more general level to a more specific level of a hierarchical dimension
Roll-up	Summarize details about a dimension	Navigate from a more specific level to a more general level of a hierarchical dimension
Pivot	Allow a data cube to be presented in a visually appealing order	Rearrange the dimensions in a data cube

the *Time* dimension can be displayed in the row area inside the *Location* dimension. A pivot operation could rearrange the data cube so that the *Location* dimension displays inside the *Time* dimension.

Summary of Operators To help you recall the data cube operators, Table 13-5 summarizes the purpose of each operator. These operators are conceptual, not actually available for business analysts. Chapter 15 presents the MDX language and pivot table tools to demonstrate extended operators in business intelligence tools. Business analysts typically use pivot table tools to manipulate data cubes. Pivot table tools provide a simple, graphical interface for performing the operators described in this section.

13.2 RELATIONAL DATA MODELING PATTERNS FOR DATA WAREHOUSES

The multidimensional data model described in the previous section was originally implemented by special-purpose storage engines for data cubes. These multidimensional storage engines support definition, manipulation, and optimization of large data cubes. Because of the commercial dominance of relational database technology, it was only a matter of time before relational DBMSs provided support for multidimensional data. Over two decades, major DBMS vendors have invested heavily in research and development to support multidimensional data. Because of the investment level and the market power of the relational DBMS vendors, most data warehouses now use relational DBMSs for primary storage of data warehouses.

Because of the importance of relational DBMS usage for data warehouses, this section presents relational data modeling patterns for multidimensional data. The first subsection explains schema patterns based on the star schema, fundamental to

relational database design for data warehouses. The next subsection shows application of schema patterns in data warehouses for retail, education, and health care. The third subsection explains time representation and historical integrity, important in query results. Because most relational DBMSs lack dimension representation, the final subsection demonstrates the Oracle CREATE DIMENSION statement, a proprietary statement beyond standard relational data modeling.

13.2.1 Schema Patterns

When using a relational database for a data warehouse, new data modeling patterns represent multidimensional data. A **star schema** is a data modeling representation of multidimensional data cubes. In a relational database, a star schema diagram looks like a star with one large central table, called the fact table, at the center of the star that is linked to multiple dimension tables in 1-M relationships in a radial pattern. The fact table stores numeric data (facts), such as sales results, while the dimension tables store descriptive data corresponding to individual dimensions of the data cube such as product, location, and time. A 1-M relationship exists from each dimension table to the fact table.

Star Schema
a data modeling representation for multidimensional databases. In a relational database, a star schema has a fact table in the center related to multiple dimension tables in 1-M relationships.

Fact tables are classified based on the types of measures stored in the tables. A transaction table contains additive measures. Typical transaction tables store measures about sales, web activity, and purchases. A snapshot table provides a periodic view of an asset level. Typical snapshot tables store semi-additive measures about inventory levels, accounts receivable balances, and accounts payable balances. A factless table records event occurrences such as attendance, room reservations, and hiring. Typically, factless tables contain foreign keys without any measures. This classification is somewhat fluid as a fact table may be a combination of these types.

Figure 13.6 shows an ERD star schema for a sales cube extending the cube presented in Section 13.1. This ERD consists of four dimension entity types, *Item*, *Customer*, *Store*, and *TimeDim*, along with one (transaction) fact entity type called Sales. When converted to a table design, the *Sales* table has foreign keys to each dimension table (*Item, Customer, Store*, and *TimeDim*). The *Item* entity type provides data for the *Product* dimension shown in the Section 13.1 examples, while the *Store* entity type provides data for the *Location* dimension. In some designs, the fact entity type depends on the related dimension entity types for its primary key. Since fact tables can have many relationships, it is generally preferred to have an artificial identifier rather than a large, combined primary key.

The store sales ERD in Figure 13.6 provides fine-grain detail for a data warehouse. The store sales ERD provides detail to the individual customer, store, and item. This level of detail is not necessary to support the sales data cubes presented in Section 13.1. However, a fine-grained level provides flexibility to support unanticipated analysis as

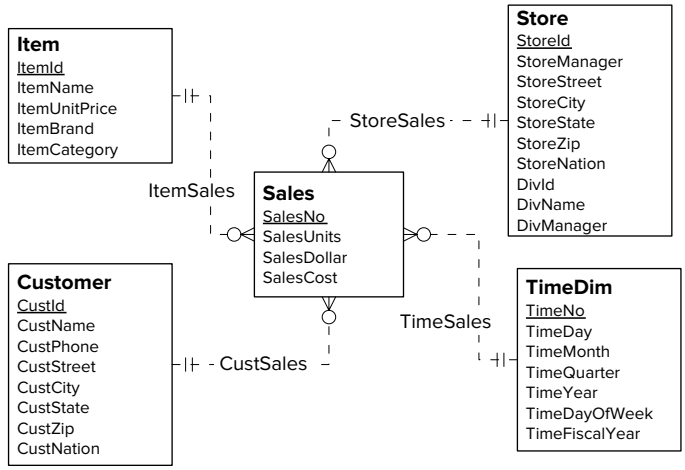


FIGURE 13.6
ERD Star Schema for the Store Sales Example

well as data mining applications. This fine-grained level may replicate data in operational databases although the data warehouse representation may differ substantially because of the subject orientation of the data warehouse and the cleaning and integration performed on the source data.

Variations to the Star Schema The star schema in Figure 13.6 represents only a single business process for sales tracking. Additional star schemas may be required for other processes such as shipping and purchasing. For related business processes that share some of the dimension tables, a star schema can be extended into a **constellation schema** with multiple fact entity types, as shown in Figure 13.7. When converted to a table design, the *Inventory* entity type becomes a fact table and 1-M relationships become foreign keys in the fact table. The *Inventory* entity type adds a number of measures including the quantity on hand of an item, the cost of an item, and the quantity returned. All dimension tables are shared among both fact tables except for the *Supplier* and *Customer* tables.

Constellation Schema
a data modeling representation for multidimensional databases. In a relational database, a constellation schema contains multiple fact tables in the center related to dimension tables. Typically, the fact tables share some dimension tables.

Snowflake Schema
a data modeling representation for multidimensional databases. In a relational database, a snowflake schema has multiple levels of dimension tables related to one or more fact tables. You should consider the snowflake schema instead of the star schema for small dimension tables that are not in 3NF.

Fact tables are usually normalized while dimension tables are often not in third normal form. For example, the *Store* entity type in Figures 13.6 and 13.7 is not in 3NF because *DivId* determines *DivName* and *DivManager*. Normalizing dimension tables to avoid storage anomalies is generally not necessary because they are usually stable and small. The nature of a data warehouse indicates that dimension tables should be designed for retrieval, not update. Retrieval performance is improved by eliminating the join operations that would be needed to combine fully normalized dimension tables.

When the dimension tables are small, denormalization provides only a small gain in retrieval performance. Thus, it is common to see small dimension tables normalized as shown in Figure 13.8. This variation is known as the **snowflake schema** because multiple levels of dimension tables surround the fact table. For the *Customer* and *Item* tables, full normalization may not be a good idea because these tables can contain a large number of rows.

The star schema and its variations require 1-M relationships from dimension tables to a fact table. The usage of 1-M relationships simplifies query formulation and supports optimization techniques discussed in Chapter 15. In some cases, M-N relationships may seem necessary between dimension and fact tables. Section 13.3 provides details about resolution of M-N relationships between fact and dimension tables.

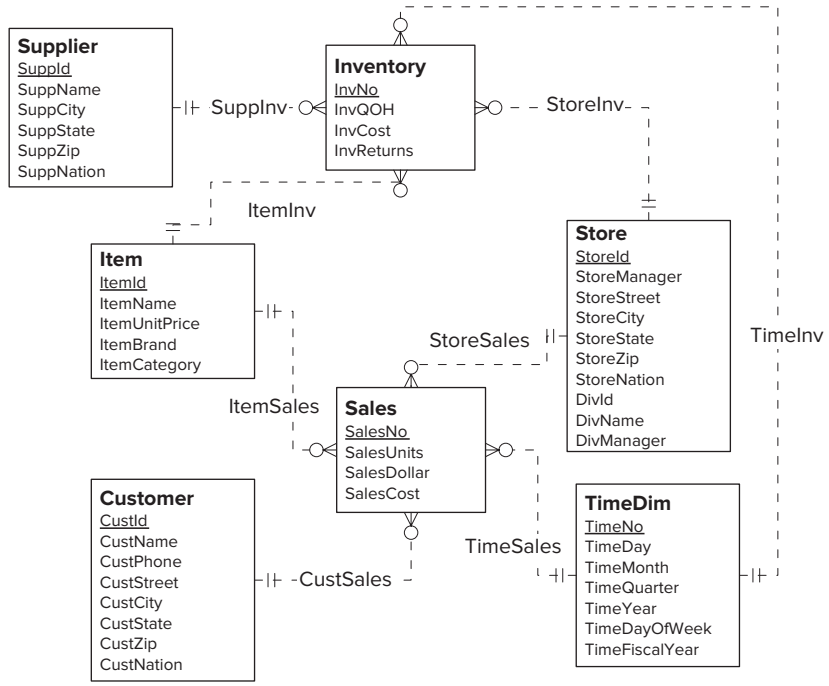


FIGURE 13.7
ERD Constellation Schema for the Sales-Inventory Example

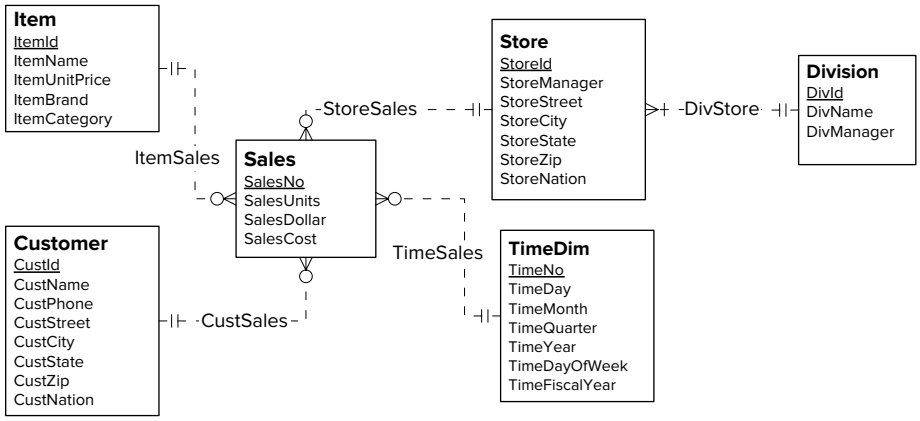


FIGURE 13.8
ERD Snowflake Schema for the Store Sales Example

13.2.2 Example Table Designs for Data Warehouses

Table designs for enterprise data warehouses usually follow the patterns in the previous section although substantial variations sometimes occur in practice. To demonstrate schema patterns in enterprise data warehouses, this section shows table designs for data warehouses in retail, education, and health care. These examples extend the presentation in Chapter 12 (section 12.3).

Table Design for the TPC-DS Benchmark™ The decision support benchmark (TPC-DS) of the Transaction Processing Performance Council (TPC) provides a moderate-sized schema using the patterns in the previous section. The TPC-DS Benchmark contains 7 schema patterns and 17 dimension tables as summarized in Table 13-6. Each column in Table 13-6 shows a schema pattern with a single fact table and a number of dimension tables. Every schema pattern contains the Item and Date_Dim dimension tables. Every schema pattern except Inventory contains dimension tables related to customers (Customer, Customer_Address, Household_Demographics, and Income_Band) as well as Time_Dim. The schema patterns involving returns (Store_Returns, Web_Returns, and Catalog_Returns) contain the Reason dimension table. The schema patterns involving sales (Store_Sales, Catalog_Sales, and Web_Sales) contain the Promotion dimension table.

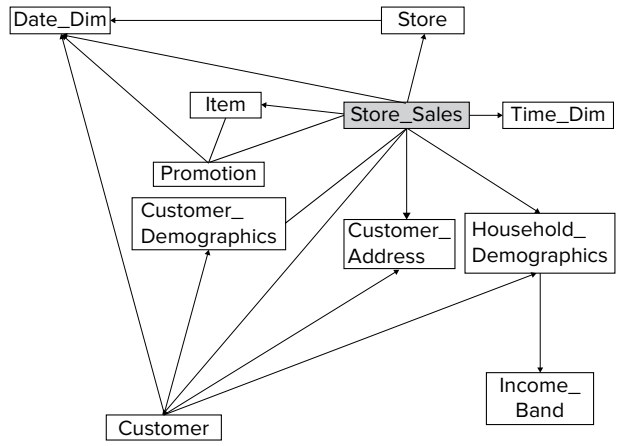
Most of the schema patterns in the TPC-DS design use the snowflake pattern. The Store Sales (Figure 13.9) and Web Sales (Figure 13.10) show snowflake designs for customer related dimension tables (Customer_Address, Household_Demographics, Customer_Demographics, and Income_Band). Note that arrows in the ERDs indicate 1-M relationships from the parent dimension table (such as Store) to the child table (such as Store_Sales). The 1-M relationship from Income_Band to Household_Demographics indicates a second level snowflake. Both schema diagrams contain relationship cycles, a variation of the schema patterns in the previous section. For example, the Store Sales schema contains the cycle from Store_Sales to Customer, Customer to Customer_Address, and Store_Sales to Customer_Address.

Table Design for the Colorado Education Data Warehouse The Colorado Education Data Warehouse, initially presented in Chapter 12.3.2, involves a complex collection of nine constellation schemas. The constellation schemas contain 94 dimension tables and 32 fact tables. The fact tables have some level of denormalization as fact tables contain both key and code values instead of just key values. The constellation schemas have some level of snowflaking as ten dimension tables are referenced in other dimension tables. The schema diagram uses nine pages in Microsoft Visio with each page containing a constellation schema with many connections among fact and dimension tables.

TABLE 13-6
Summary of the Table Design for the TPC-DS Benchmark

Dimension Tables	Fact Tables						
	Store_ Sales	Store_ Returns	Catalog_ Sales	Catalog_ Returns	Web_ Sales	Web_ Returns	Inventory
Store	√	√					
Item	√	√	√	√	√	√	√
Date_Dim	√	√	√	√	√	√	√
Time_Dim	√	√	√	√	√	√	
Promotion	√		√		√		
Customer_Demographics	√	√	√		√	√	
Customer_Address	√	√	√	√	√	√	
Household_Demographics	√	√	√	√	√	√	
Customer	√	√	√	√	√	√	
Income_Band	√	√	√	√	√	√	
Reason		√		√		√	
Call_Center			√	√			
Catalog_Page			√	√			
Ship_Mode			√	√	√	√	
Warehouse			√	√	√	√	√
Web_Site					√	√	
Web_Page					√	√	

FIGURE 13.9
Store Sales Schema Diagram for the TPC-DS Benchmark[1]



One of the simplest schemas contains the fact table for standardized test (CSAP) scores along with connections to 27 dimension tables as listed in Table 13-7. The dimension tables demonstrate most of the concepts in earlier parts of this chapter. Most dimension tables contain flat dimensions, typically a code value. For example, dimension tables for ethnicity, homeless, migrant status, and gifted contain a single, flat dimension. Some dimension tables contain hierarchical dimensions and multiple dimensions. For example, the school dimension has location columns that form hierarchical dimensions and other columns comprising flat dimensions. Each dimension table uses two date columns (beginning and ending effective dates) to provide historical integrity as explained in the next subsection.

[1] Figures 13.9 and 13.10, from the TPC Benchmark™ DS Standard Specification, Version 2.5.0, are copyrighted by the Transaction Processing Performance Council.

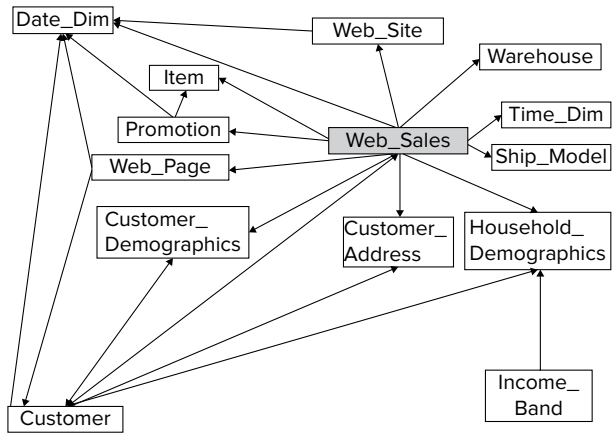


FIGURE 13.10
Web Sales Schema Diagram for the TPC-DS Benchmark

DIM_504_PLAN	DIM_ESL	DIM_IEP
DIM_ACCOMMODATION	DIM_ETHNICITY	DIM_LANGUAGE_BACKGROUND
DIM_BILINGUAL	DIM_FARM	DIM_MIGRANT_STATUS
DIM_CBLA_STATUS_CODE	DIM_GENDER	DIM_SCHOOL_EMH
DIM_CSAP_CONTENT_PROFICIENCY	DIM_GIFTED_TALENTED	DIM_SCHOOL_YEAR
DIM_CSAP_SUBJECT	DIM_GRAD_CLASS	DIM_SCHOOL
DIM_DID_NOT_TEST	DIM_GRADE_CALC_EXEMPTION	DIM_TIME_IN_DISTRICT
DIM_DISABLING_CONDITION	DIM_GRADE	DIM_TIME_IN_SCHOOL
DIM_DISTRICT	DIM_HOMELESS	DIM_TITLE_1

TABLE 13-7
List of Dimensions in the Student Achievement Star Schema

The fact tables contain measures with a variety of aggregation properties. The major fact table for CSAP results contains additive measures (number of points, percentage points, scaled score, and growth percentile) although these measures should be shown as central tendencies (average or median) for reasonable interpretation. Summary fact tables contain event counts such as the number of partially proficient students in a test result. The finance fact tables contain semi-additive measures that can be summarized across time such as bonded debt levels.

The Education Data Warehouse is modest-sized compared to typical enterprise data warehouses. The total size of the Education Data Warehouse is about 270 GB. The fact tables have about 200 million rows with 1.6 million rows added per year to the major fact table.

Table Design for the OMOP Common Data Model As presented in Chapter 12.3.3, the Observational Medical Outcomes Partnership (OMOP), a public/private partnership, developed a prominent data warehouse standard for electronic medical records. The OMOP Common Data Model (CDM) provides a detailed specification for a medical data warehouse. Figure 13.11 shows an overview of the CDM with 39 tables organized in 5 areas. The standardized clinical tables contain details about clinical events for each person during observation periods. A visit occurrence can involve multiple procedures, drug exposures, device exposures, conditions, observations, lab results, and notes.

The other areas in the CDM support the standard clinical data. The tables for standardized health economics contain cost details about visits, procedures, drugs, and devices, dependent on the health care delivery system for a patient. The tables for the health system document providers, care sites, and locations involved with patient care. The tables in the derived elements area store derived details about cohorts, doses, drugs, and conditions. A cohort can be derived from persons, providers, or visits. The

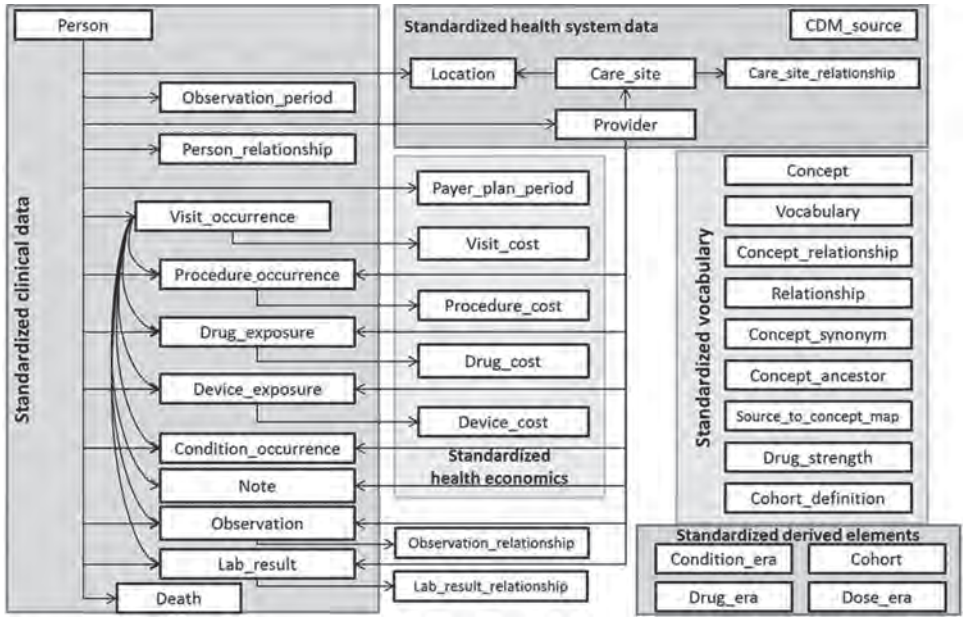


FIGURE 13.11
Overview of the OMOP Common Data Model[2]

tables in the standardized vocabulary support detailed concepts used in CDM fact tables. These tables are constant for each instantiation of the CDM.

The OMOP CDM contains variation from the standard schema patterns. As shown in Figure 13.12, the ERD[3] for the standardized clinical data contain M-N relationships rather than just 1-M relationships in the standard schema patterns. Person is the dimension table with 1-M relationships to two major fact tables (Visit_Occurrence and Specimen). The other fact tables have 1-M relationships with Visit_Occurrence and Person. For example, 1-M relationships exist from Person and Visit_Occurrence to Measurement and Observation. Measurement and Observation are associative entity types representing M-N relationships. Thus, the ERD for the standardized clinical data complicate the standard schema patterns with M-N relationships.

13.2.3 Time Representation and Historical Integrity

Time representation is a crucial issue for data warehouses because most queries use time in conditions. The principal usage of time is to record the occurrence of facts. The simplest representation is a timestamp data type for a column in a fact table. In place of a timestamp column, many data warehouses use a foreign key to a time dimension table as shown in previous subsections. Using a time dimension table supports convenient representation of organization-specific calendar features such as holidays, fiscal years, and week numbers that are not represented in timestamp data types. The granularity of the time dimension table is usually in days. If time of day is also required for a fact table, it can be added as a column in the fact table to augment the foreign key to the time table.

Most fact tables involve time represented as a foreign key to the time table with augmentation for time of day if required. For fact tables involving international operations, two time representations (time table foreign keys along with optional time of day columns) can be used to record the time at source and destination locations. A variation identified by Kimball (2003) is the accumulating fact table that records the status of multiple events rather than one event. For example, a fact table containing a

[2] Figures 13.11 and 13.12 are copyrighted by the Observational Health Data Sciences and Informatics (www.ohdsi.org).

[3] This ERD notation displays a 1-M relationship with the key symbol by the parent entity type and infinity symbol by the child entity type.

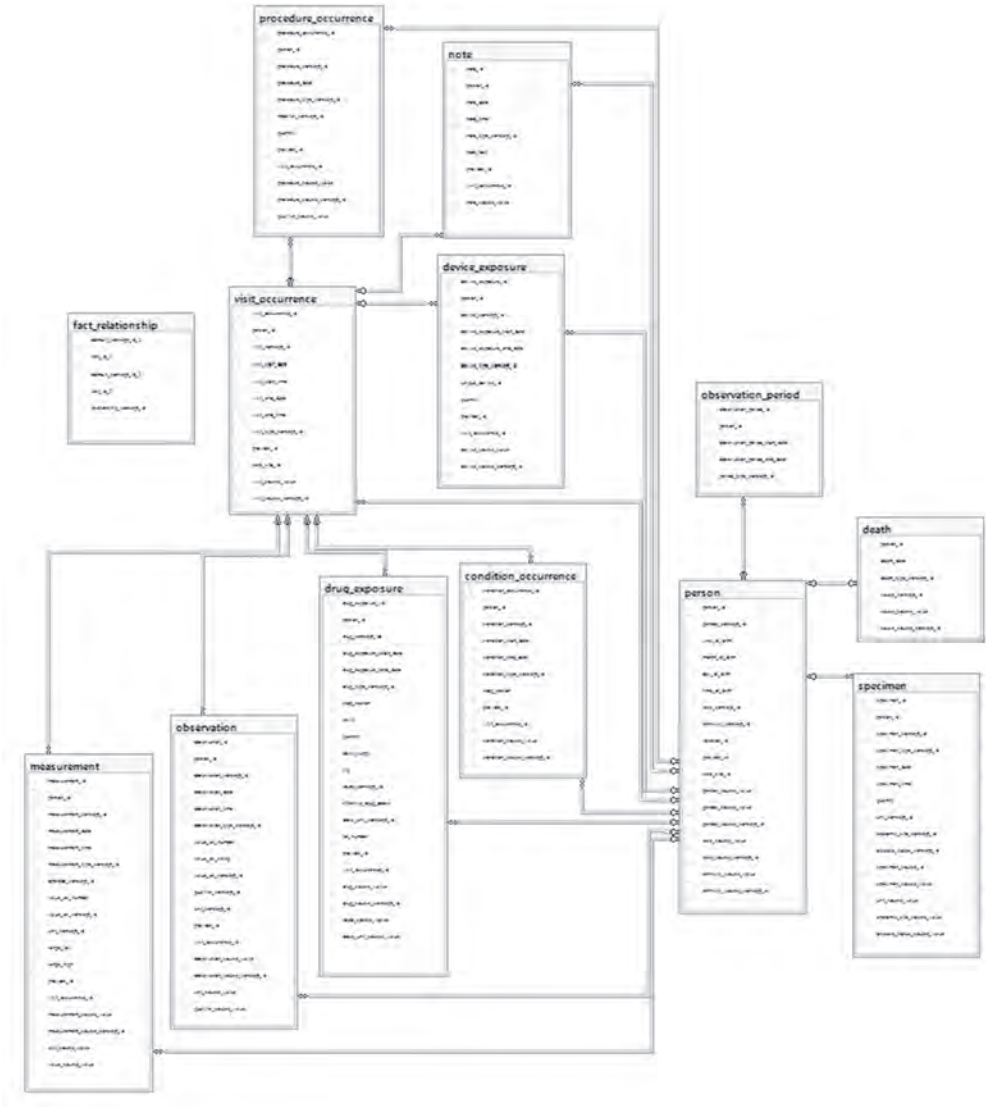


FIGURE 13.12
ERD for the Standardized Clinical Data Tables

snapshot of order processing would include order date, shipment date, delivery date, payment date, and so on. Each event occurrence column can be represented by a foreign key to the time table along with a time of day column if needed.

For dimension tables, time representation involves the level of historical integrity, an issue for updates to dimension tables. When a dimension row is updated, related fact table rows are no longer historically accurate. For example, if the city column of a customer row changes, the related sales rows are no longer historically accurate. To preserve historical integrity, the related sales rows should point to an older version of the customer row. Kimball (April 1996) presents three alternatives for historical integrity:

- Type I: overwrite old values with the changed data. This method provides no historical integrity.
- Type II: use a version number to augment the primary key of a dimension table. For each change to a dimension row, insert a row in the dimension table with a larger version number. For example, to handle the change to the city column, there is a new row in the *Customer* table with the same customer number but a larger version number than the previous row. Besides the version number, additional columns are needed to record the beginning effective date and ending effective date for each historical column.

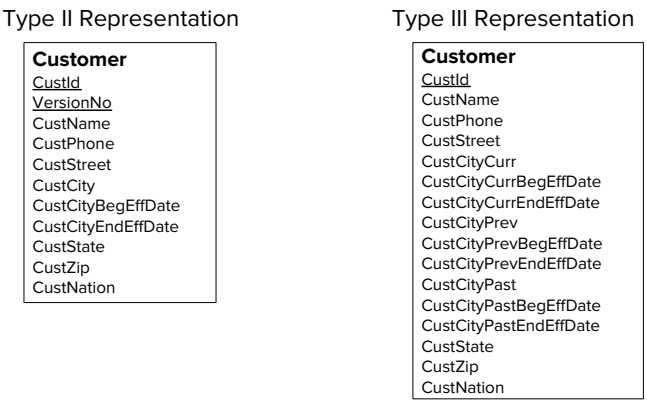


FIGURE 13.13
Alternatives for Historical Dimensional Integrity of *CustCity*

- Type III: use additional columns to maintain a fixed history. For example, to maintain a history of the current city and the two previous city changes, three city columns (*CustCityCurr*, *CustCityPrev*, *CustCityPast*) can be stored in the *Customer* table along with associated six date columns (two date columns per historical value column) to record the effective dates.

Figure 13.13 shows Type II and Type III alternatives for the *CustCity* column. The Type II alternative involves multiple rows for the same customer, but the entire history is represented. The Type III alternative involves just a single row for each customer, but only a limited history can be represented.

13.2.4 Extensions for Dimension Representation

The SQL CREATE TABLE statement lacks explicit representation of hierarchical dimensions. Thus, the star schema and its variations do not provide explicit representation of hierarchical relationships of levels of a dimension. Because dimension definition is important to support data cube operations as well as optimization techniques for query rewriting, some relational DBMS vendors have created proprietary SQL extensions for dimensions. This section reviews the Oracle CREATE DIMENSION statement to indicate the types of extensions that can be found in relational DBMSs.

The Oracle CREATE DIMENSION statement[4] supports the specification of levels, hierarchies, and constraints for a dimension. The first part of a dimension declaration involves the specification of levels. For flat (nonhierarchical) dimensions, only a single level exists in a dimension. However, most dimensions involve multiple levels as depicted in Example 13.1 for the *StoreDim* dimension. Each level corresponds to one column from the *Store* source table.

Example 13.1

Oracle CREATE DIMENSION Statement for the *StoreDim* Dimension with the Specification of Levels

```
CREATE DIMENSION StoreDim
 LEVEL StoreId    IS Store.StoreId
 LEVEL City       IS Store.StoreCity
 LEVEL State      IS Store.StoreState
 LEVEL Zip        IS Store.StoreZip
 LEVEL Nation     IS Store.StoreNation ;
```

[4] Do not put blank lines in CREATE DIMENSION statements. The Oracle SQL compiler generates error messages when encountering blank lines in CREATE DIMENSION statements.

The next part of a CREATE DIMENSION statement involves the specification of hierarchies. The Oracle CREATE DIMENSION statement supports dimensions with multiple hierarchies as shown in Example 13.2. Specification of a hierarchy proceeds from the most detailed level to the most general level. The CHILD OF keywords indicate the direct hierarchical relationships in a dimension.

Example 13.2

Oracle CREATE DIMENSION Statement for the *StoreDim* Dimension with the Specification of Levels and Hierarchies

```
CREATE DIMENSION StoreDim
 LEVEL StoreId    IS Store.StoreId
 LEVEL City       IS Store.StoreCity
 LEVEL State      IS Store.StoreState
 LEVEL Zip        IS Store.StoreZip
 LEVEL Nation     IS Store.StoreNation
 HIERARCHY CityRollup (
  StoreId CHILD OF
  City    CHILD OF
  State   CHILD OF
  Nation )
HIERARCHY ZipRollup (
  StoreId CHILD OF
  Zip     CHILD OF
  State   CHILD OF
  Nation );
```

The Oracle CREATE DIMENSION statement supports dimensions with levels from multiple source tables. This feature applies to normalized dimension tables in snowflake schemas. Example 13.3 augments Example 13.2 with the inclusion of an additional level (*DivId*) along with an additional hierarchy containing the new level. In the level specification, the *DivId* level references the *Division* table. In the *DivisionRollup* hierarchy, the JOIN KEY clause indicates a join between the *Store* and the *Division* tables. The JOIN KEY clause, at the end of a hierarchy specification, applies to a hierarchy with levels from more than one source table.

Example 13.3

Oracle CREATE DIMENSION Statement for the *StoreDim* Dimension with the Usage of Multiple Source Tables

```
CREATE DIMENSION StoreDim
 LEVEL StoreId    IS Store.StoreId
 LEVEL City       IS Store.StoreCity
 LEVEL State      IS Store.StoreState
 LEVEL Zip        IS Store.StoreZip
 LEVEL Nation     IS Store.StoreNation
 LEVEL DivId      IS Division.DivId
```

```
 HIERARCHY CityRollup (
  StoreId CHILD OF
  City    CHILD OF
  State   CHILD OF
  Nation )
HIERARCHY ZipRollup (
  StoreId CHILD OF
  Zip     CHILD OF
  State   CHILD OF
  Nation )
HIERARCHY DivisionRollup (
  StoreId CHILD OF
  DivId
  JOIN KEY Store.DivId REFERENCES DivId );
```

The final part of a CREATE DIMENSION statement involves constraint specification. The ATTRIBUTE clause defines functional dependency relationships involving dimension levels and related columns in dimension tables. Example 13.4 shows ATTRIBUTE clauses for the related columns in the *Division* table.

Example 13.4

Oracle CREATE DIMENSION Statement for the *StoreDim* Dimension with the Usage of ATTRIBUTE Clauses for Constraints

```
CREATE DIMENSION StoreDim
 LEVEL StoreId    IS Store.StoreId
 LEVEL City       IS Store.StoreCity
 LEVEL State      IS Store.StoreState
 LEVEL Zip        IS Store.StoreZip
 LEVEL Nation     IS Store.StoreNation
 LEVEL DivId      IS Division.DivId
 HIERARCHY CityRollup (
  StoreId CHILD OF
  City    CHILD OF
  State   CHILD OF
  Nation )
HIERARCHY ZipRollup (
  StoreId CHILD OF
  Zip     CHILD OF
  State   CHILD OF
  Nation )
HIERARCHY DivisionRollup (
  StoreId CHILD OF
  DivId
  JOIN KEY Store.DivId REFERENCES DivId )
ATTRIBUTE DivId DETERMINES Division.DivName
ATTRIBUTE DivId DETERMINES Division.DivManager ;
```

In Example 13.4, the DETERMINES clauses are redundant with the primary key constraint for the *Division* table. The DETERMINES clauses are shown in Example 13.4 to reinforce constraints supported by the primary key declarations. DETERMINES clauses are required for constraints not corresponding to primary key constraints to enable query optimizations. For example, if each zip code is associated with one state, a DETERMINES clause should be used to enable optimizations involving the zip code and state columns.

13.3 SUMMARIZABILITY PROBLEMS AND PATTERNS

Summarizability involves aggregation/disaggregation between a coarse level of detail and finer levels of details. Summary operations are common in data warehouse queries. Summary operations occur in drill down and rollup operations on a data cube and join operations combining fact and dimension tables.

Violations of summarizability conditions detract from the usability of a data warehouse. The most serious summarizability violations produce erroneous results. Even if results are not incorrect, violations of summarizability conditions can lead to user confusion. Violations of summarizability conditions can also restrict the ability to use optimizations that improve query performance.

The first subsection focuses on the summarizability problems involving dimension tables containing hierarchical dimensions. The second subsection covers summarizability problems involving join operations between fact and dimension tables.

13.3.1 Dimension-Fact Summarizability Problems and Patterns

This subsection presents three **dimension summarizability problems** beginning with problems involving the drill-down operator. Drill-down incompleteness involves inconsistency between totals shown in drill-down operations. Drilling from a parent (coarser) level to a child (finer) level shows a smaller total indicating that measures attributed to parent members have not been allocated to child members. In Figure 13.14, the parent college level drills down to the department child level with a smaller total. The enrollment in the business college is omitted in the department level because the business college does not have departments. This inconsistency could cause user confusion and erroneous decision making.

Dimension Summarizability Problems
inconsistent results that occur in summary operations involving relationships between entity types representing dimension levels. The inconsistent results involve different totals when summing values at the child and parent levels in a dimension hierarchy.

Roll-up incompleteness reverses drill-down incompleteness. Rolling up from a child (finer) level to a parent (coarser) level shows a smaller total indicating that measure values attributed to child members have not been allocated to parent members. In Figure 13.15, the product child level rolls up to the category parent level with a smaller total. The sales of napkin products are omitted in the category level because napkin products are not food or drink. This inconsistency could cause user confusion and erroneous decision making.

The non strict dimension problem involves M-N relationships between dimension levels, typically exceptions to 1-M relationships. For example, the time dimension can have M-N relationships between levels as depicted in Figure 13.16. The

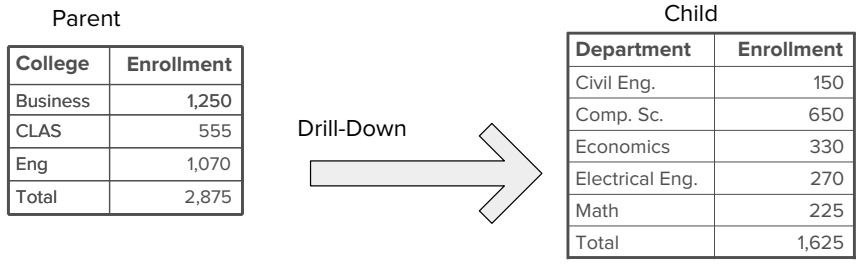


FIGURE 13.14
Drill-Down Incompleteness Example

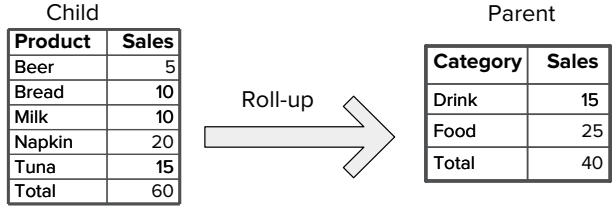


FIGURE 13.15
Roll-up Incompleteness Example

FIGURE 13.16
Example of a Non Strict Dimension Problem

Child

Week	Sales
1-2017	5
2-2017	10
3-2017	10
4-2017	10
5-2017	20
6-2017	10
7-2017	10
8-2017	10
9-2017	10
Total	95

Roll-up

Parent

Month	Sales
Jan-2017	37
Feb-2017	53
Total	90

weeks of the year can overlap months leading to different totals for weeks and months. In Figure 13.16, January and February involve almost nine complete weeks. The week total (95) is more than the two month total (90). Users may perceive the difference as an inconsistency if they expect a 1-M relationship between month and weeks.

Dimension Summarizability Patterns To solidify your understanding of dimension summarizability, you should generalize beyond the examples presented in the previous subsection. Schema patterns provide a tool to generalize beyond examples. The summarizability patterns involve values for the minimum and maximum cardinalities. You should be able to recognize schema patterns that provide summarizability as well as patterns involving summarizability problems.

Summarizable schema patterns eliminate all three dimension summarizability problems. As shown in Figure 13.17, the regular dimension pattern involves a minimum cardinalilty of 1 for both the parent and child levels of a dimension hierarchy. The parent's minimum cardinality of 1 eliminates drill-down incompleteness, while the child's minimum cardinality of 1 eliminates rollup incompleteness. Although the unusual dimension pattern with a maximum cardinality of 1 for the parent eliminates summarizability problems, it is not typical in practice. The examples in Figure 13.18 involving Year-Month (a) and Division-Brand (b) are typical examples of regular summarizability patterns. In the year-month example, month refers to

Dimension Summarizability Pattern: a schema pattern that ensures consistent results in summary operations involving the parent and child entity type in the relationship. Relationship cardinalities determine the consistency of the summary operations. The regular and unusual patterns are the two dimension summarizability patterns.

FIGURE 13.17
Schema Patterns for Summarizable Dimensions

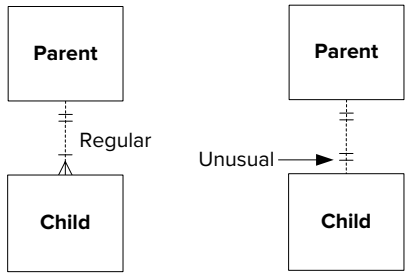
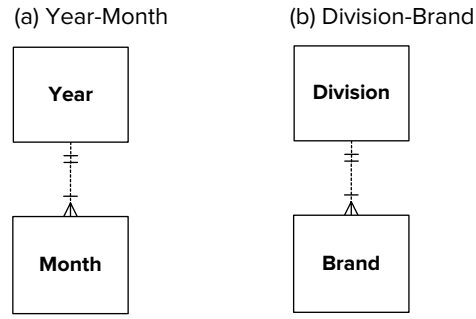


FIGURE 13.18
Examples of Schema Patterns for Summarizable Dimensions



the month within a specific year such as August 2017. The relationship between *Division* and *Store* in Figure 13.8 is another example of a regular summarizability pattern.

The values of minimum and maximum cardinalities determine schema patterns for the three dimension summarizability problems. The drill-down incomplete problem involves a minimum cardinality of 0 for the parent entity type as shown in Figure 13.19. The roll-up incomplete problem involves a child's minimum cardinality of 0. The non-strictness problem involves a M-N relationship. The examples in Figure 13.20 depict ERDs for the College-Department, Category-Product, and Month-WeekofYear examples presented in Figures 13.14 to 13.16 as small tables.

Resolving the dimension summarizability problems is typically not difficult conceptually although the solutions may complicate the data integration process. For drill-down incompleteness, unallocated parent members should be related to a default child member. For example, colleges without departments should be represented by a new child member such as business college enrollments should be allocated to the "unallocated business" member in the department level. For roll-up incompleteness, a new default parent member should be used for child members without a parent. For example, a new "non food, non beverage" category can be added to the parent level to relate to unassociated child members such as napkin. For the non strict dimension problem, the simplest solution is to place the levels in different hierarchies. For example, the weekofyear and month levels should be placed in different date hierarchies. When the levels cannot be placed in different hierarchies, a major parent can be used in place of multiple, related parent members. For example, a product related to multiple categories can be related just to a major category. Another possible solution is to group related parent members into parent groups so that each child member is related to a single parent group.

To help you recall the conditions for dimension summarizability completeness and incompleteness, Table 13-8 lists the conditions for each pattern.

13.3.2 Dimension-Fact Summarizability Problems and Patterns

Relationships between dimension and fact tables dominate relational representation of data warehouses as explained in Section 13.2. Thus, it is important that join operations between fact and dimension tables show consistent summaries of measures.

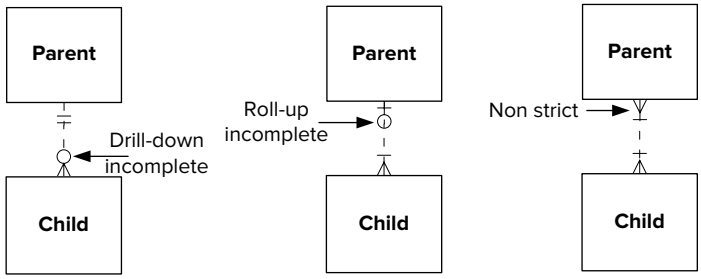


FIGURE 13.19
Schema Patterns for Dimension Summarizability Problems

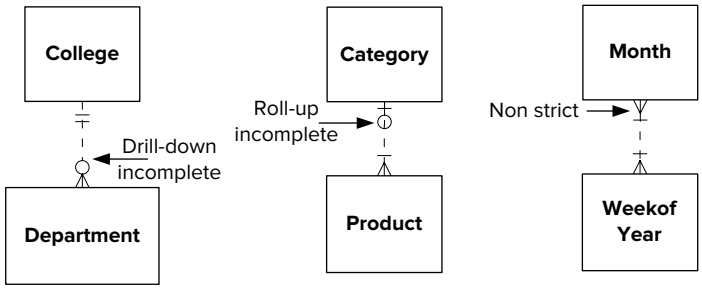


FIGURE 13.20
Examples of Schema Patterns for Dimension Summarizability Problems

TABLE 13-8
Summary of the Dimension Summarizability Conditions

Summarizability Pattern	Conditions
Drill-down complete	Parent minimum cardinality = 1
Drill-down incomplete	Parent minimum cardinality = 0
Roll-up complete	Child minimum cardinality = 1
Roll-up incomplete	Child minimum cardinality = 0
Non strict	Child maximum cardinality = M
Regular	Parent min, max cardinality = (1, M) Child min, max cardinality = (1, 1)
Unusual	Parent max cardinality = 1

Incomplete dimension-fact relationships involve fact entities that do not have a related parent entity in a dimension-fact relationship. Inconsistencies caused by incomplete relationships are manifest in join operations with summary calculations. Figure 13.21 demonstrates inconsistent totals between (a) sales summarized by both customer and month and (b) sales summarized by month only. The sales summarized by both customer and month are inconsistent with the sales summarized by month due to anonymous customers. Some sales have been recorded without a known customer due to customer requirements for anonymity. A business analyst may become confused because of the inconsistent totals.

The non strict dimension-fact relationship problem involves double counting measure values due to an M-N relationship between dimension and fact tables. In Figure 13.22, double counting of sales occurs when summarizing sales by individual

Dimension-Fact Relationship Summarizability Problems: inconsistent results that occur in summary operations involving relationships between dimension and fact entity types. The inconsistent results involve different totals when summing values involving different dimension-fact relationships.

FIGURE 13.21
Incomplete Dimension-Fact Relationship Example

(a) Summarized by Customer and Month

Customer	Month	Sales
Cust-1	Jan-2017	10
Cust-2	Jan-2017	5
Cust-3	Feb-2017	15
Total		30

(b) Summarized by Month Only

Month	Sales
Jan-2017	25
Feb-2017	15
Total	40

FIGURE 13.22
Non Strict Dimension-Fact Relationship Example

(a) Unit sales by salesperson

Salesperson	Date	UnitSales
SP1	10-Feb-2017	10
SP2	10-Feb-2017	10
SP3	11-Feb-2017	15
SP4	12-Feb-2017	20
Total		55

(b) Shared unit sales by salesperson

Salesperson	Date	UnitSales
SP1, SP2	10-Feb-2017	10
SP3	11-Feb-2017	15
SP4	12-Feb-2017	20
Total		45

salesperson and date. The sale on February 10, 2017 involves both SP1 and SP2. The individual sales total (55) is greater than the shared sales (45) because the shared sale is counted twice in Figure 13.22(a). The M-N relationship between the salesperson dimension table and sales fact table leads to double counting of the sales total, possibly causing erroneous decision making and user confusion.

Dimension-Fact Relationship Summarizability Pattern a schema pattern that ensures consistent results in summary operations involving relationships between dimension and fact entity types. Relationship cardinalities determine the consistency of the summary operations. The regular and unusual patterns are the two dimension-fact relationship summarizability patterns.

Schema patterns for dimension-fact relationships support a more general understanding of summarizability problems. Figure 13.23 depicts patterns for summarizable dimension-fact relationships. In the regular dimension-fact pattern, a dimension entity may not be related to any fact entity. The optional relationship supports sparsity in which some dimension members do not have related fact entities. In Figure 13.23, some products have not been sold so the relationship is optional. The unusual dimension-fact pattern involves a mandatory relationship for the dimension entity type. In Figure 13.24, every date must have a related sale. The mandatory relationship for the dimension entity type is often not enforced because of difficulties in the data integration process.

Non summarizable dimension-fact relationship patterns deviate from the summarizable patterns in the cardinalities for the fact entity type. The incomplete dimension-fact relationship pattern involves a minimum cardinality of 0 for the fact entity type as shown in Figure 13.25. As an example, the *Purchase* entity type in Figure 13.26 has a minimum cardinality of 0 indicating an incomplete dimension-fact relationship. The non strict dimension-fact relationship involves a minimum cardinality of M for the fact entity type as shown in Figure 13.25. As an example, the *Sales* entity type has a maximum cardinality of M in Figure 13.26.

Resolving incomplete dimension-fact relationships is conceptually simple although the resolution may complicate the data integration process. Unrelated fact entities should be connected to a default dimension entity if a connection to an actual

FIGURE 13.23
Schema Patterns for Summarizable Dimension-Fact Relationships

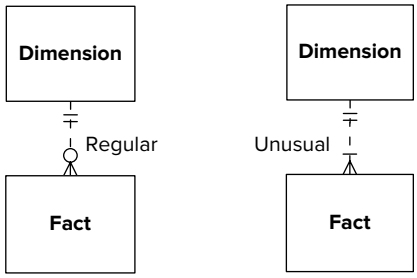


FIGURE 13.24
Schema Examples of Summarizable Dimension-Fact Relationships

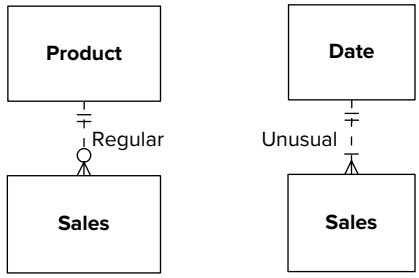


FIGURE 13.25
Schema Patterns for Non Summarizable Dimension-Fact Relationships

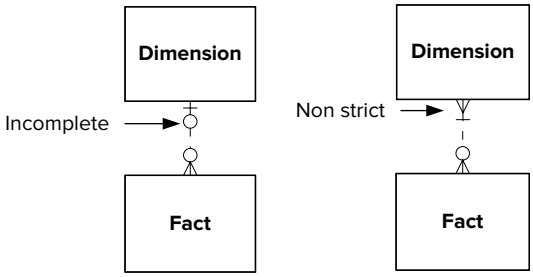


FIGURE 13.26
Schema Examples of Non Summarizable Dimension-Fact Relationships

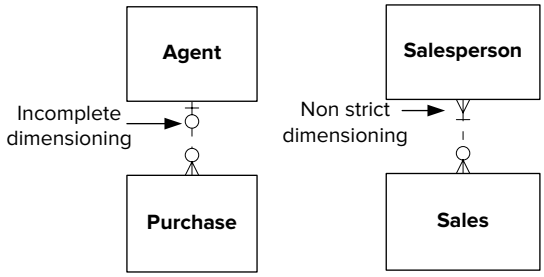


TABLE 13-9
Summary of Dimension-Fact Relationship Summarizability Conditions

Summarizability Pattern	Conditions
Complete dimension-fact relationship	Fact minimum cardinality = 1
Incomplete dimension-fact relationship	Fact minimum cardinality = 0
Strict dimension-fact relationship	Fact maximum cardinality = 1
Non strict dimension-fact relationship	Fact maximum cardinality = M
Regular dimension-fact relationship	Dimension min, max cardinality = (0, M) Fact min, max cardinality = (1, 1)
Unusual dimension-fact relationship	Dimension minimum cardinality = 1

dimension entity cannot be made. For example, anonymous sales should be connected to a default anonymous customer in the customer entity type.

Resolving non strict dimension-fact relationships can be more complex than resolution of incomplete relationships. Sometimes the source data has exceptions that involve M-N relationships, not 1-M relationships. For example, if the *Sales* fact table is derived from customer invoices, some invoices may involve multiple customers such as roommates or spouses. Two ways to resolve non strict dimension-fact relationships are explained in the following list.

- If there are a small, fixed number of possible related entities, a simple adjustment can be made to a fact entity type. Multiple relationships can be added between dimension and fact entity types to allow for more than one related entity. For example, the *Sales* entity type in Figure 13.6 can have an additional relationship to identify an optional second customer on an invoice.
- If there can be groups of related entities, the representation is more difficult. An entity type representing a group of related entities can be added with an associative entity type that connects the other entity types. For example to represent customer groups in Figure 13.6, a customer group entity type can be added along with 1-M relationships to connect the customer group, *Sales*, and *Customer* entity types.

Resolving M-N relationships involves a compromise for simplified query formulation and data integration procedures. Some data warehouse designs retain M-N relationships to prioritize a faithful representation rather simplified query formulation and integration procedures. The OMOP CDM in Section 13.2.2 shows a data warehouse design preserving M-N relationships. The major part of the data warehouse design for clinical patient data (Figure 13.12) contains several associative entity types representing M-N relationships.

To help you recall the conditions for summarizability completeness and strictness of dimension-fact relationships, Table 13-9 lists the conditions for each pattern.

13.4 SCHEMA INTEGRATION AND DESIGN METHODOLOGIES

Data warehouse professionals use schema patterns and summarizability concepts as important tools in data warehouse design. To apply them in data warehouse design, these tools should be used in a schema integration process and overall design methodology. This section presents a process for schema integration and design methodologies for enterprise data warehouse design. The schema integration process specifies steps for a integrating a small number of data sources into a conceptual data warehouse design. A design methodology specifies a strategy to apply the schema integration process to a large number data sources for an enterprise data warehouse design. Together, the schema integration process and design methodology support enterprise data warehouse design using schema patterns and summarizability concepts as important tools.

13.4.1 Schema Integration Process

The schema integration process uses specifications about business intelligence needs and data source documentation to integrate a small number of data sources into a conceptual design of a data warehouse. An organization's requirements for business intelligence indicate important decisions, measures, and analysis methods used by business analysts. Some of the requirements can be wish lists, not currently used by business analysts. Data sources provide raw material to satisfy business intelligence requirements. The schema integration process uses ERDs, data dictionary documentation, and sample data about data sources.

Using documentation of data sources and specifications of business needs as input, the schema integration process involves a sequence of steps to integrate a small number of data sources as depicted in Figure 13.27. In the first step, you specify dimensions and measures including important properties presented in Section 13.1. In the second step, you determine the most appropriate grain for each dimension and make estimated size calculations about the grains. A grain indicates the level of detail for dimensions such as individual customers and stores. Estimated size calculations about fact tables allow planning about hardware and software capacity for a data warehouse. In the third step, you create a table design for the data warehouse that supports the dimensional model and data sources. In the fourth step, you identify summarizability problems and provide resolutions using patterns presented in Section 13.3. Finally, in the fifth step, you provide mappings for the data sources and populate data warehouse tables using sample data from the data sources. A mapping is an initial specification for the data integration process. Chapter 14 presents details of data integration to extend and implement an initial mapping specification.

The schema integration process may involve backtracking and iteration through the steps. The order of steps indicates a logical progression allowing backtracking and iteration especially as business intelligence requirements evolve as an organization conducts the process.

Mini Case Study for Schema Integration This section depicts details of the schema integration process using a mini case study involving two data sources for a retail firm. The Purchase database (Figure 13.28) supports purchase transactions

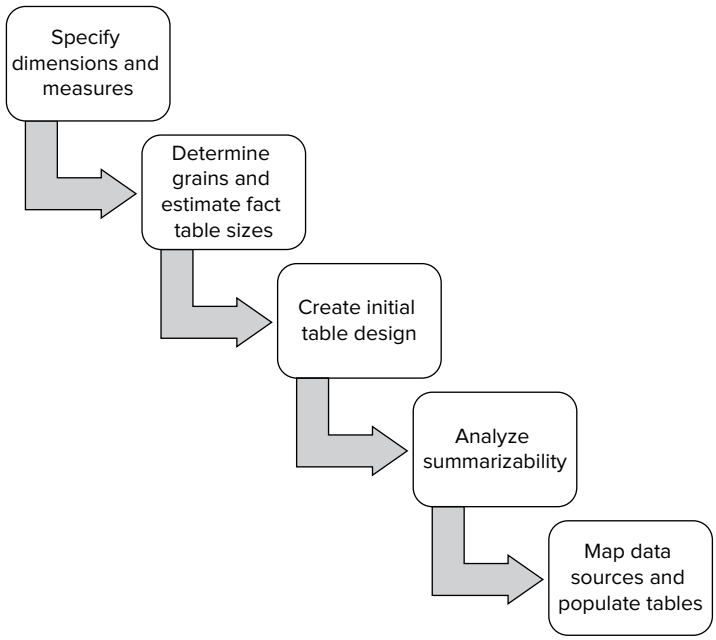


FIGURE 13.27
Steps of the Schema Integration Process

to replenish retail inventory. A purchase consists of a heading with the purchase number, date, payment method, delivery date, and supplier. A purchase contains a collection of products with the quantity and unit cost recorded on purchase lines along with links to the product and purchase heading. Each product has one preferred supplier. However, a purchase can use a non-preferred supplier if necessary. For additional insight about the tables, Appendix 13.A contains sample rows for each table.

Individual stores of the retail firm also maintain an inventory of custom products ordered from local suppliers. Individual store order products using purchase worksheets for custom products as depicted in Table 13-10. Inventory practices for custom products are informal. New products are typically purchased when a manager senses new demand for local items.

The data warehouse tracks inventory balances over time, a type of snapshot. Snapshots are typical in applications in which balances are involved, such as account balances in financial services, enrollment in courses, reservations in hospitality and travel, and head count in personnel management. Snapshots cannot be aggregated over time correctly as summation of quantities and values over time is not meaningful.

The basic measures for inventory tracking are quantity on hand and inventory value. Inventory valuation can be complex as many accounting methods exist to value inventory. For this problem, the purchase price or unit cost of the inventory can be used for valuation. The data warehouse should support detailed tracking of inventory to the individual product, purchased by date, and supplier. Here are typical computations for analyzing and tracking inventory balances using the quantity on hand and inventory valuation measures.

- The average quantities and stock values in each time period
- The opening and closing balances for each time period
- The change in inventory levels between consecutive periods and parallel periods
- The minimum and maximum inventory levels in a time period
- The relative contribution of the stocked item to the overall stock value

FIGURE 13.28
ERD for Small Purchase Database

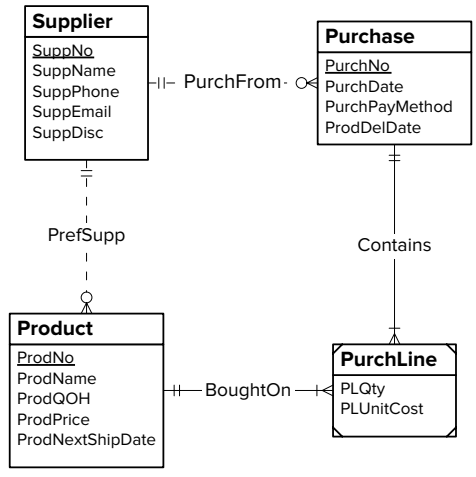


TABLE 13-10
Sample Worksheet for Custom Inventory

ProdCode	ProdDesc	Supp	Qty	Unit Price	PurDate	Amount
CPC1	Pencils	Omart	20	$2.00	13-Feb-2017	$40.00
CPC2	Paper	Smart	10	$3.50	14-Feb-2017	$35.00
CPC3	Folders	Pmart	20	$1.50	11-Feb-2017	$30.00

Selected Solution Details for the Mini Case Study The requirements of this mini case study involve three important decisions that involve some extensions of material in previous sections. This mini case study requires calculations about estimated grain sizes to gain insight about storage requirements of finer grains such as the individual customer and store. Simplicity is an important principle of data warehouse design. One area for simplification is fact table choice. Often an operational database has transactions with multiple levels of detail such as the *Purchase* and *PurchLine* entity types. Data warehouse designs are dominated by fact tables with a single level. Populating sample data warehouse tables from source data provides insight about data integration requirements and clarifies omissions in a data warehouse design.

Appendix 13.B contains full details of the solution so the remainder of this section presents details of grain determination and mappings, aspects not covered in other parts of this chapter. For each dimension, a data warehouse designer should determine alternative grains or levels of details to estimate impacts on fact table sizes. The level of detail of each related dimension determines fact table size and sparsity. Sparsity appears in data cubes as empty cells. In a fact table, sparsity involves combinations of dimension values without a related fact table row. Fine grains such as individual customers, days, and products dominate data warehouse designs because of analysis flexibility. However, fine grains cause large fact table sizes, high level of sparsity, and substantial computing resources. Coarser grains such as customer postal codes, product categories, and weeks reduce flexibility of analysis especially on data mining applications. However, coarser grains reduce fact table sizes, sparsity, and computing resources.

For fact table size estimation, the approach depends on matching fact tables to source tables as depicted in Figure 13.29. If you can match a fact table to source tables, you should determine sparsity using statistics about the size of a source table. For example, if the fact table corresponds to the *PurchLine* table, you can use statistics about the *PurchLine* size (number of rows) to compute sparsity. To compute sparsity, you first compute the fill ratio, the number of non-empty cells to total cells. Fill ratio is the number of rows in the source table divided by the product of the sizes of each dimension table. Sparsity is computed as 1 minus the fill ratio.

If you cannot match a fact table to existing source tables (such as for coarse grains), you should estimate sparsity to calculate the fact table size. Determining fact table size may be necessary if statistics about a source table are not reliable or likely to change substantially. To compute fact table size, you first calculate the fill ratio as the 1 minus estimated sparsity since sparsity and fill ratio sum to 1. Fact table size is computed as the product of the size of each dimension table multiplied by the fill ratio.

Mapping sample data in data sources to data warehouse tables involves associations and additional data as depicted in Figure 13.30. Most of the mapping is associations among source and data warehouse columns. There can be conversions such as

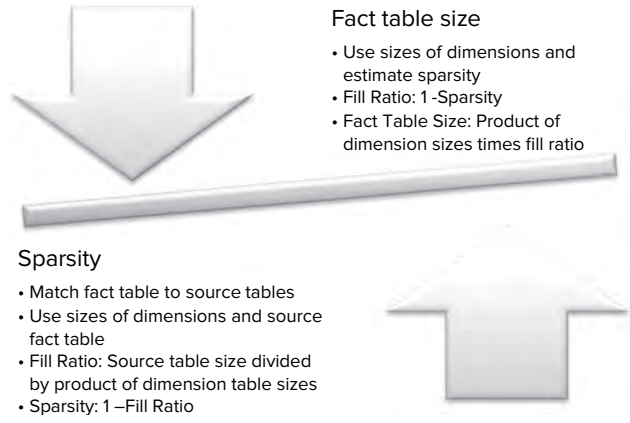


FIGURE 13.29
Grain Determination

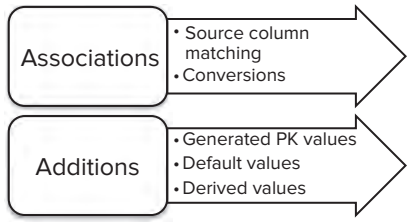


FIGURE 13.30
Data Source Mapping

units of measure but details of conversions are not important for this problem. Chapter 14 covers details of conversions with data integration tools so the specification here just involves the need for conversion.

Additional data is typically necessary to specify a complete mapping. Typical additions are generated primary key values, default values for missing values, and derived values. A typical derived value is a data source indicator to specify the data source for a row. In this case, a data source indicator can specify if the source is the Purchases database or custom products worksheet.

13.4.2 Data Warehouse Design Methodologies

When designing an enterprise data warehouse, an organization requires a methodology to apply the principles and practices of data modeling and schema integration. Without an appropriate methodology, the best principles and practices will likely fail to produce a data warehouse with high value for an organization. This subsection reviews a range of proposed methodologies to provide background and insights for successful development of enterprise data warehouses.

Design methodologies for data warehouses differ from methodologies for transaction databases because data warehouses are primarily repositories of secondary data. Operational databases capture data at its source such as from an ATM, web shopping cart, and manufacturing plant. In contrast, data warehouses transform data from primary sources (operational databases and external data sources) and then store and summarize the transformed data.

Because of differences in data characteristics and usage, data warehouses have different design artifacts than operational databases. Data warehouse design methodologies support the development of data models, data integration procedures, and data marts for enterprise data warehouses. The data models for data warehouses have different patterns than data models for operational databases. Data integration procedures are essential for data warehouses but typically not important for operational databases. Data marts have different characteristics than views for operational databases.

The methodologies presented in this subsection have received some prominence although numerous other approaches also have been proposed. Data warehouse design methodologies differ by emphasis on the supply of data sources, the demand for business intelligence services, and the possible level of automation in the development process. Automation may have an important role because data sources already exist as raw materials for data warehouse design. However, no commercial products have been developed to support automation in data warehouse design methodologies.

Demand-driven data warehouse design methodology emphasizes the identification of data marts to capture intended usage of a data warehouse. The demand-driven methodology has three phases for (1) identifying data marts (subsets of user requirements), (2) building a matrix relating data marts and dimensions, and (3) designing fact tables.

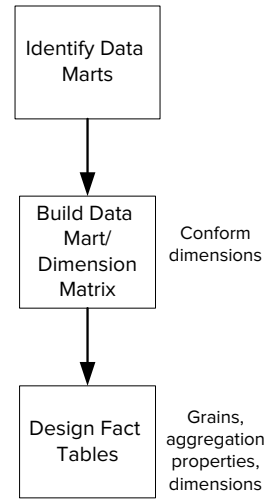
Demand Driven Methodology The **demand-driven data warehouse design methodology** (also known as the requirements driven approach), first proposed by Kimball et al. (1998), is one of the earliest data warehouse design methodologies. The demand-driven methodology emphasizes the identification of data marts to capture intended usage of a data warehouse as depicted in Figure 13.31. A data mart is defined as a collection of related facts important for a group of data warehouse users. When this methodology was proposed, the data mart architecture was common for data

warehouses. Because of its emphasis on subsets of user requirements, the demand-driven approach has some similarity to the view driven approach to database design[5].

After identifying data marts, possible dimensions for each data mart are listed. Dimensions, standardized across data marts, are known as conformed dimensions. A matrix relating conformed dimensions and data marts is developed to refine the initial data mart specification.

The final step involves specification of fact tables with an emphasis on the grain of fact tables. Typical grains are individual transactions, snapshots (points in time), and line items on documents. The grain is usually determined by the primary dimensions. After specifying the grain of a fact table, all dimensions are specified. The details of each dimension are then specified including the hierarchical levels. In the last part, measures for each fact table are specified including the measure properties such as aggregation.

FIGURE 13.31
Steps in the Demand-Driven Data Warehouse Design Methodology



Supply Driven Methodology The **supply-driven data warehouse design methodology** (Moody and Kortink, 2000) emphasizes the analysis of existing data sources. Entity types in ERDs of existing data sources are analyzed to provide a starting point for the data warehouse design as shown in Figure 13.32. The supply-driven methodology seems amenable to automation although automated tools to support the methodology have not been reported.

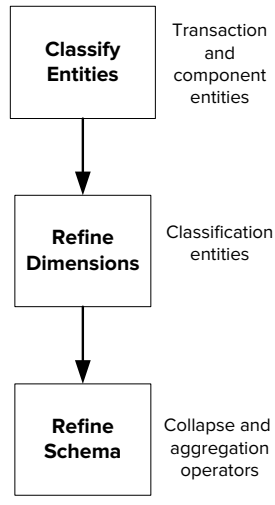
Supply-driven data warehouse design methodology emphasizes the analysis of existing data sources. Entities in ERDs of existing data sources are analyzed to provide a starting point for the data warehouse design. The supply-driven methodology has three phases for (1) classify entities, (2) refine dimensions, and (3) refine the schema.

In the first step, the supply-driven approach classifies entity types in existing ERDs as a prelude to develop a star schema or variation for the data warehouse. Entity types containing event data at a point in time are classified as transaction entity types. Guidelines in the methodology indicate that transaction entity types typically have numeric data that can be summarized. Typical events involve sales, purchases, reservations, hiring, and so on. Event entity types will typically become fact tables in a star schema. Entity types related to events in 1-M relationships are classified as component entity types. Component entity types typically become dimension tables in a star schema. In total, the first step provides a set of initial star schemas or a constellation schema if dimensions are conformed.

The second step of the supply-driven methodology refines dimensions. Entity types related to component entity types are labeled as classification entity types. Dimension hierarchies are formed by classification and component entity types. Each sequence of classification and component entity types, joined by 1-M relationships in the same direction, becomes a dimension hierarchy.

The third step of the methodology refines the star schemas using two operators. The collapse operator denormalizes dimension entity types to reduce snowflaking. For example, the collapse operator applied to the Division-Store relationship in Figure 13.8 combines the *Division* and *Store* entity types eliminating the snowflake design. The aggregation operator makes the grain coarser in transaction entity types. Aggregation of a fact table may require modifications to the primary dimension tables to make the dimension tables consistent with the grain of the fact table.

FIGURE 13.32
Steps in the Supply-Driven Data Warehouse Design Methodology



Hybrid Methodology The **hybrid data warehouse design methodology** (Bonifati et al., 2001) combines the demand and supply methodologies. The hybrid methodology involves a demand-driven stage, a supply-driven stage, and then a third stage to integrate the demand and supply-driven stages. The demand and supply stages can be done independently as shown in Figure 13.33. The overall emphasis in the hybrid approach is to balance the demand and supply aspects of data warehouse design possibly aided by automated tools.

The demand-driven stage collects requirements using the Goal-Question-Metrics (GQM) paradigm. The GQM provides forms and interview guidelines to help define a

[5] Chapter 12 of the sixth edition covers the view driven approach for database design. Although this chapter has been removed in the later editions, you can find this chapter on the textbook's website.

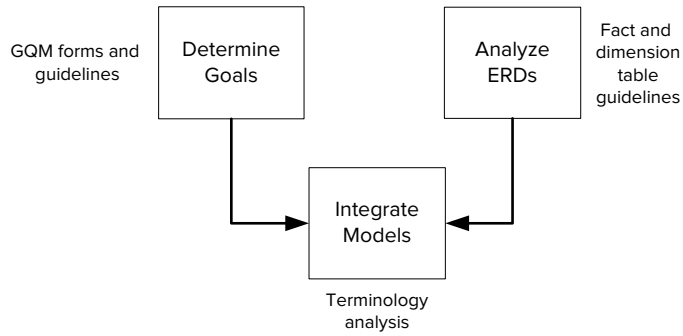


FIGURE 13.33
Steps in the Hybrid Data Warehouse Design Methodology

Hybrid data warehouse design methodology: combines the demand and supply methodologies. The hybrid methodology involves a demand-driven stage, a supply-driven stage, and then a third stage to integrate the demand and supply-driven stages. The demand and supply stages can be done independently. The overall emphasis in the hybrid approach is to balance the demand and supply aspects of data warehouse design.

set of goals for the data warehouse. The methodology provides some informal guidelines to derive measures and dimensions from the goals.

The second step of the hybrid methodology involves analysis of existing ERDs. This step may be partially automated although tool development has not been reported. The methodology provides guidelines to identify fact and dimension tables in existing ERDs. Potential fact tables are identified based on the number of additive attributes. Dimension tables are involved in 1-M relationships with fact tables.

The third step of the hybrid methodology integrates the dimensional model in the demand stage and the star schema in the supply stage. The methodology provides guidelines to convert both models to a common vocabulary using terminology analysis. After conversion to a common vocabulary, the methodology provides a process to match the demand and supply models.

CLOSING THOUGHTS

This chapter provided detailed coverage of conceptual design concepts and practices for enterprise data warehouses. To design a data warehouse, most organizations use both the multidimensional data model and the relational model. Most organizations use a relational DBMS for primary storage of a data warehouse and business intelligence tools use the multidimensional model as a representation for business analysts.

For the multidimensional data model, this chapter presented terminology associated with data cubes and operators to manipulate data cubes. For the relational data model, this chapter described data modeling patterns (the star schema and its variations), design rules for summarizability, time representation in relational data warehouse designs, and extensions for representation of hierarchical dimensions in SQL. To depict complexity of enterprise data warehouse designs in organizations, this chapter demonstrated data warehouse designs in retail, education, and health care.

The final part of the chapter emphasized design of enterprise data warehouses. A schema integration process was presented for analyzing a small number of data sources. For designing enterprise data warehouses, several prominent design methodologies were reviewed.

The concepts and design skills emphasized in this chapter provide a foundation for data warehouse professionals. The remaining chapters in part 6 extend this foundation with skills for data integration and query formulation.

REVIEW CONCEPTS

- Multidimensional data cube: dimensions, measures, hierarchies, time-series data type
- Important dimension properties: hierarchy and sparsity

- Aggregation property for measures: additive, semi-additive, non-additive
- Data cube operators: slice, dice, drill-down, roll-up, pivot
- Star schema: fact table and related dimension tables in 1-M relationships
- Variations of the star schema: snowflake schema (multiple dimension levels) and constellation schema (multiple fact tables and shared dimension tables)
- Classifications of fact tables: transaction table, snapshot table, and factless table
- Maintaining historical dimensional integrity using a Type II representation for unlimited history and a Type III representation for limited history
- Dimension representation using the Oracle proprietary SQL statement (CREATE DIMENSION) to support data cube operations and optimization techniques
- Dimension summarizability problems: drill-down incompleteness, roll-up incompleteness, and non strict dimensions
- Dimension summarizability patterns: regular dimension pattern and unusual dimension pattern
- Fact-dimension summarizability problems: incomplete dimension-fact relationships and non strict dimension-fact relationships
- Fact-dimension summarizability patterns: regular dimension-fact pattern and unusual dimension-fact pattern
- Schema integration process for analyzing a small set of data sources
- Input for the schema integration process: documentation of data sources and specifications of business intelligence requirements
- Important decisions of schema integration: dimensional data model, grain size estimation, table design, initial mapping of data sources, and population of data warehouse tables with sample data
- Estimate sparsity by matching a fact table to source tables or estimate fact table size using sizes of dimensions and estimate of sparsity
- Simplifying a data warehouse design by combining multiple source tables into one fact table
- Data source mappings through source column matching, conversions, generated primary key values, default values, and derived values
- Demand-driven data warehouse design methodology emphasizing the identification of data marts to capture intended usage of a data warehouse
- Supply-driven data warehouse design methodology emphasizing the analysis of existing data sources
- Hybrid data warehouse design methodology balancing the demand and supply-driven methodologies

QUESTIONS

1. What are the advantages of multidimensional representation over relational representation for business analysts?
2. Explain why a dimension may have multiple hierarchies.
3. Why is it important to specify the aggregation property for each measure?
4. Provide an example of a measure with each aggregation property value (additive, semi-additive, and non-additive).
5. What are the advantages of using time-series data in a cell instead of time as a dimension?
6. How is slicing a data cube different from dicing?

7. What are the differences between drilling-down and rolling-up a data cube dimension?
8. How is a pivot operation useful for multidimensional databases?
9. Explain the significance of sparsity in a data cube.
10. What is a star schema?
11. What are the differences between fact tables and dimension tables?
12. How does a snowflake schema differ from a star schema?
13. What is a constellation schema?
14. What is the relationship between the types of fact tables and aggregation properties for measures?
15. What is an accumulating fact table?
16. In the schema patterns for the TPC-DS Benchmark, what fact tables involve sales?
17. In the schema patterns for the TPC-DS Benchmark, what fact tables involve returns?
18. What variation to standard schema patterns is shown in the TPC-DS Benchmark?
19. Describe the schema for standardized test scores in the Colorado Education Data Warehouse.
20. Describe the main area of the OMOP Common Data Model.
21. Describe the variation from the standard schema patterns in the OMOP Common Data Model.
22. How is time represented in a fact table?
23. What is the difference between Type II and Type III representations for historical dimension integrity?
24. What is the purpose of the Oracle CREATE DIMENSION statement?
25. What is the difference between drill-down incompleteness and roll-up incompleteness?
26. For the non strict dimension problem, will totals summarized at the child level always be larger than totals summarized in the related parent level?
27. Why is the unusual dimension summarizability pattern called unusual?
28. List important cardinalities of the regular summarizability dimension pattern, the drill-down incomplete pattern, and the roll-up imcomplete pattern.
29. Briefly explain the incomplete dimension fact relationship problem including the schema pattern for the problem.
30. Briefly explain the difference between the non strict dimension problem and the non strict dimension-fact relationship problem.
31. What is the difference between the regular and unusual patterns for summarizable dimension-fact relationships?
32. Briefly explain two ways to resolve non strict dimension-fact relationships.
33. Briefly explain resolution of the incompleteness problems for dimension summarizability.
34. Briefly explain resolution of the incomplete dimension-fact relationship problem.
35. What is the relationship between the schema integration process and design methodologies for enterprise data warehouse development?
36. What inputs does the schema integration process use?
37. Does the schema integration process allow backtracking and iteration among the steps?

38. What is a grain in the schema integration process?
39. What is a mapping in the schema integration process? How is a mapping related to the data integration process?
40. How can you simplify fact table design for a data warehouse schema?
41. Why should you populate sample data warehouse tables in the schema integration process?
42. What are two alternative approaches for grain size estimation?
43. How do you compute the sparsity of a fact table?
44. How do you compute a fact table size starting with an estimate of sparsity?
45. What is involved with mapping sample data in data sources to data warehouse tables?
46. What additional data is necessary to specify a complete mapping in the data integration process?
47. Why are new design methodologies needed for data warehouse design instead of using approaches for design of operational databases?
48. Briefly explain the demand-driven data warehouse design methodology.
49. Briefly explain the supply-driven data warehouse design methodology.
50. Briefly explain the hybrid data warehouse design methodology.

PROBLEMS

The problems provide practice with data cube definition and operations, schema design patterns, resolution of summarizability problems, and CREATE DIMENSION statements. The problems use a database for an automobile insurance provider (Figure 13.P1) to support policy transactions (create and maintain customer policies)

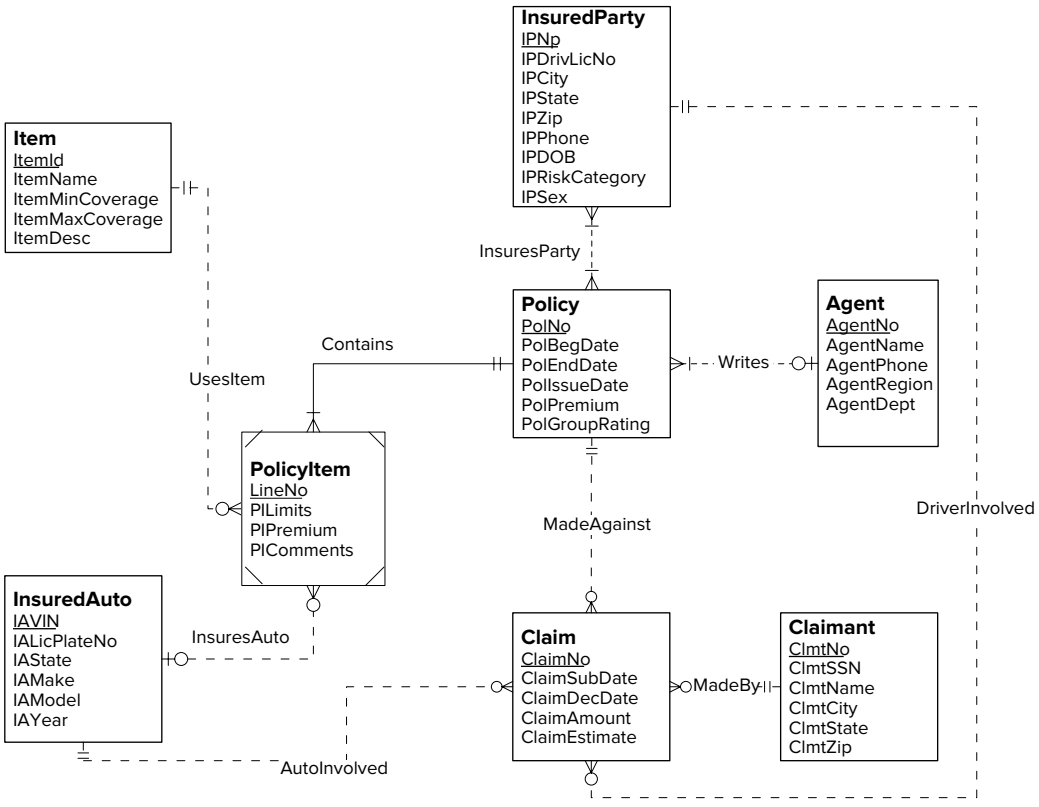


FIGURE 13.P1
ERD for Auto Insurance Policies and Claims

and claims transactions (claims made by other parties). This database design is simplified to provide reasonable practice problems with the concepts in Chapter 13. The policy transactions utilize the entity types *Item*, *Agent*, *InsuredParty*, *Policy*, *InsuredAuto*, and *PolicyItem*, while the claims transactions use the entity types *InsuredParty*, *Claimant*, *InsuredAuto*, *Policy*, and *Claim*. For each entity type, you should assume the data types of your own choice. The cardinalities for the *InsuresParty* relationship indicate that a policy can involve an entire family, not just individuals.

1. Identify dimensions and measures in a data cube for automobile policy analysis. Indicate the aggregation property of each measure.
2. Identify the finest level grain of the fact table to support automobile policy analysis. Justify your decision.
3. Consider a relationship between a dimension for insured parties and the fact table in problem 2. Identify a summarizability problem involving this relationship. Propose an alternative to resolve the summarizability problem.
4. Consider a relationship between a dimension for agents and the fact table in problem 2. Identify a summarizability problem involving this relationship. Propose an alternative to resolve the problem.
5. Consider relationships between dimension tables for items and insured autos and the fact table in problem 2. Identify any summarizability problems in the relationships. Propose an alternative representation for each summarizability problem that you identified.
6. Design a time dimension table and one or more relationships to the fact table in problem 2.
7. Design a star or snowflake schema to support the dimensions and measures in the data cube from problem 1. Your schema should use resolutions to summarizability problems that you proposed in earlier problems.
8. For each dimension table in the schema for problem 7, list the independent and hierarchical dimensions. Analyze each hierarchical dimension for summarizability problems. Propose alternative representations to resolve any summarizability problems.
9. Identify dimensions and measures in a data cube for claims analysis. Indicate the aggregation property of each measure.
10. Identify the finest level grain of the fact table to support automobile claim analysis. Justify your decision.
11. Consider a relationship between a dimension for claimants and the fact table in problem 10. Identify any summarizability problems involving this relationship. Propose an alternative to resolve the problem if a problem exists.
12. Consider a relationship between a dimension for insured autos and the fact table in problem 10. Identify any summarizability problems involving this relationship. Propose an alternative to resolve the problem if a problem exists.
13. Consider a relationship between a dimension for policies and the fact table in problem 10. Identify any summarizability problems involving this relationship. Propose an alternative to resolve the problem if a problem exists.
14. Consider a relationship between a dimension for insured party and the fact table in problem 10. Identify any summarizability problems involving this relationship. Propose an alternative to resolve the problem if a problem exists.
15. Design a time dimension table and one or more relationships to the fact table in problem 10.
16. Design a star or snowflake schema to support the dimensions and measures in the data cube for problem 9. Your schema should use resolutions to summarizability problems that you proposed in earlier problems.

17. For each dimension table in the schema for problem 16, list the independent and hierarchical dimensions. Analyze each hierarchical dimension for summarizability problems. Propose alternative representations to resolve any summarizability problems.
18. Combine the star or snowflake schemas for policies (problem 7) and claims (problem 16) into a constellation schema. Identify common dimension tables that can be shared in the constellation schema.
19. For the *InsuredPartyDim* table, discuss the stability of the columns in the table. What columns would typically change together? What columns would history be desirable?
20. For the *InsuredAutoDim* table, discuss the stability of the columns in the table. What columns would typically change together? What columns would history be desirable?
21. Modify the *InsuredPartyDim* table for a history of the *IPRiskCategory* column. Provide a type II representation and a type III representation with current and previous risk values.
22. Modify the *InsuredPartyDim* table for a limited history of the *IPCity*, *IPState*, and *IPZip* columns. The limited history should record the current and previous values and change dates for the combination of columns.
23. Describe the data cube resulting from the operation to slice the policy data cube by a certain agent.
24. Describe the data cube resulting from the operation to dice the data cube result of the slice operation in problem 9 by insured parties having zip codes in a specified state.
25. Begin with a data cube with four dimensions (*InsuredParty*, *InsuredAuto*, *Item*, and *Agent*) and one measure (policy amount) in the cells. From this data cube, describe the operation to generate a new data cube with three dimensions (*InsuredParty*, *Item*, and *Agent*) and one measure (average auto policy amount).
26. This problem involves the addition of external telematics data into the auto insurance data warehouse design. Telematics involves advanced sensors installed in a car or a smartphone to collect data on an individual car usage including time, location, types of roads, and driving behavior such as speed. The external data source is an XML file with the following structure. The XML file only contains data for drivers insured by the auto insurance company. For this problem, identify the dimensions and measures involved in the XML file. Think carefully about the grain for the measures as the XML file is very large with time intervals every 5 to 10 minutes per driving experience.
    - Header record: VIN, license plate number, state, driver license number, insurance policy number.
    - Detail record (multiple records for the header record): date, start time, end time, start GPS coordinates, end GPS coordinates, driving distance, road type, average driving speed, and average speed limit.
27. Design a star or snowflake schema to support the dimensions and measures in the data cube from problem 26. Utilize the dimensions in the schemas from problem 18 as much as possible. You can assume that the data integration process would add data missing from the XML file for some dimensions.

The final two problems provide practice with the Oracle CREATE DIMENSION statement. These two problems need the *SSCustomer* and *SSTimeDim* tables. These tables are part of the Store Sales database used in Chapter 15. Here are the CREATE TABLE statements for these tables. Prefix of SS in table names avoids schema name conflicts.

```
CREATE TABLE SSCustomer
 (CustId      CHAR(8) NOT NULL,
  CustName    VARCHAR2(30) NOT NULL,
  CustPhone   VARCHAR2(15) NOT NULL,
  CustStreet  VARCHAR2(50) NOT NULL,
  CustCity    VARCHAR2(30) NOT NULL,
  CustState   VARCHAR2(20) NOT NULL,
  CustZip     VARCHAR2(10) NOT NULL,
  CustNation  VARCHAR2(20) NOT NULL,
 CONSTRAINT PKSSCustomer PRIMARY KEY (CustId)  );
CREATE TABLE SSTimeDim
 (TimeNo          INTEGER NOT NULL,
  TimeDay         INTEGER NOT NULL,
  TimeMonth       INTEGER NOT NULL,
  TimeQuarter     INTEGER NOT NULL,
  TimeYear        INTEGER NOT NULL,
  TimeDayofWeek   INTEGER NOT NULL,
  TimeFiscalYear INTEGER NOT NULL,
CONSTRAINT PKSSTimeDim PRIMARY KEY (TimeNo) );
```

28. Write an Oracle CREATE DIMENSION statement for a customer dimension consisting of the customer identifier, the city, the state, the zip, and the country. Define two hierarchies grouping the customer identifier with the city, the state, and the nation and the customer identifier with the zip, the state, and the nation. You should review the material in Chapter 13.2.4 about the Oracle CREATE DIMENSION statement.
29. Write an Oracle CREATE DIMENSION statement for a time dimension consisting of the time identifier, the day, the month, the quarter, the year, the fiscal year, and the day of the week. Define three hierarchies grouping the time identifier, the day, the month, the quarter, and the year, the time identifier and the fiscal year, and the time identifier and the day of the week. You should review the material in Chapter 13.2.4 about the Oracle CREATE DIMENSION statement.

PRACTICE MINI CASE STUDY FOR SCHEMA INTEGRATION

The practice mini case study provides additional experience with the schema integration process. You should not attempt the practice mini case study until you understand the basic mini case study in Section 13.4.1 (with details in Appendix 13A) and its solution (Appendix 13B) in Chapter 13.

This practice mini case study contains two data sources with sample data along with a statement of business intelligence needs. Using the data sources and business intelligence needs, you will specify a dimensional model with properties of dimensions and measures, estimate the grain size, create a schema design for the data warehouse that integrates the data sources, identify summarizability problems in the design, and populate data warehouse tables from sample rows in the data sources.

Data Sources

Fitness Unlimited is a leading provider of exercise centers with a variety of fitness programs and membership options. Fitness Unlimited maintains a retail database to track sales of services and merchandise. In the ERD for the retail database (Figure 13.CS1), a sale contains a heading (*Sale*) with sales date and a collection of merchandise recorded in the M-N relationship *Contains*. Service purchases are recorded in the *ServPurchase* entity type with 1-M relationships from ServiceCategory and Member. Typical services are lessons, premium equipment usage,

and social events. The *MemTypeOf* relationship is optional for members because guest members can use a fitness center and purchase merchandise and services on a short term basis without having a paid membership. Tables with sample rows are shown after Figure 13.CS1.

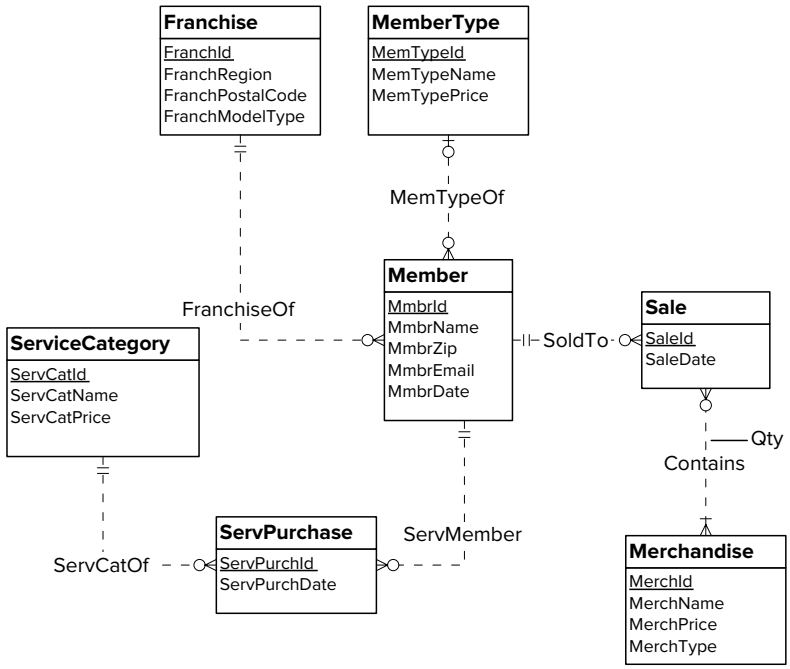


FIGURE 13.CS1
ERD for Retail Fitness Database

Franchise			
FranchId	**FranchRegion**	**FranchPostalCode**	**FranchModelType**
F1	Northwest	98011	Full
F2	Mountain	80111	Medium
F3	Central	45236	Limited

MemberType		
MemTypeId	**MemTypeName**	**MemTypePrice**
M1	Platinum	$1,000
M2	Gold	$800
M3	Value	$300

ServiceCategory		
ServCatId	**ServCatName**	**ServCatPrice**
SC1	Ball machine	$15
SC2	Private lesson	$75
SC3	Adult class	$150
SC4	Child class	$125

Merchandise			
MerchId	**MerchName**	**MerchPrice**	**MerchType**
MC1	Wilson balls	$3	Balls
MC2	Wilson racket	$200	Racket
MC3	Adidas shoes	$100	Shoes
MC4	Racket stringing	$40	Racket

Member						
MmbrId	**MmbrName**	**MmbrZip**	**MemTypeId**	**MmbrDate**	**FranchId**	**MmbrEmail**
1111	Joe	98011	M1	1-Feb-2009	F1	joe@serv1.com
2222	Mary	80112	M2	1-Jan-2010	F2	mary@serv2.com
3333	Sue	45327	M3	3-Mar-2011	F3	sue@serv3.com
4444	George	45236			F3	george@serv4.com

Sale		
SaleId	**SaleDate**	**MmbrId**
1111	10-Feb-2017	1111
2222	13-Feb-2017	2222
3333	13-Feb-2017	2222
4444	14-Feb-2017	3333

Contains		
MerchId	**SaleId**	**Qty**
MC1	1111	2
MC2	1111	1
MC4	2222	1
MC3	3333	1
MC4	4444	1

ServicePurchase			
ServPurchId	**ServPurchDate**	**MmbrId**	**ServCatId**
1111	13-Feb-2017	1111	SC1
2222	14-Feb-2017	2222	SC2
4444	15-Feb-2017	4444	SC3

Franchises also sell special events to corporate customers and other organizations. Since special event promotions and sales are not standard among franchises, spreadsheets are typically used to track special events. The franchise sales database was never extended to accommodate special event sales. The Special Events Worksheet shows a typical format for tracking special event sales by a franchise. Most franchises use a similar spreadsheet.

Special Events Worksheet					
Corporate Customer Id	**Corporate Customer Name And Location**	**Event Type Code**	**Event Name**	**Event Date**	**Amount**
CC1	First Data, Greenwood Village	L-A	Adult Social	13-Feb-2017	$1,000
CC2	DU Tennis, Denver	L-B	Pioneer Social	14-Feb-2017	$500
CC3	Creek High School, Greenwood Village	L-C	Team Practice	21-Feb-2017	$200

Data Source Size Estimates

To estimate grain size, you should use these estimates about cardinalities of tables and unique values of some columns.

- Franchise rows: 350
- Franchise postal codes: 200
- MemberType rows: 10
- Merchandise rows: 500
- MerchType values: 30
- ServCategory rows: 20
- Member rows: 50,000
- Member zip codes: 500
- Sale rows: 150,000 per year
- Contains rows: 450,000 per year
- ServicePurchase rows: 100,000 rows per year
- SpecialEvents Worksheet rows: 300 per year per franchise with 200 franchises using this spreadsheet
- 150 unique customers per special event worksheet

Business Intelligence Requirements

The data warehouse should support analysis of merchandise sales and service purchases by franchise, merchandise or service type, and customer over time. For merchandise, sales amount is computed as quantity times selling price. For services purchases, each unit sale is recorded separately so only the service price at the time of purchase is recorded. For customer, merchandise sales should be tracked by zip code, membership date, and member type. For franchise, merchandise sales should be tracked by franchise region, postal code, and model type.

The corporate sales office wants a high level of flexibility for sales analysis. For data mining analysis, the sale office needs detail by individual customer, product or service, franchise, and date. For typical reporting applications, the sales office needs detail by customer location, franchise location, product or service type, and week.

Schema Integration Requirements

You should design a star schema (or variation) to support revenue analysis. You should pay close attention to the grain of the fact table, the major part of the star schema diagram. As part of the design, you should identify all relevant dimensions with hierarchies specified. In your documentation, you should identify summarizability problems in your star schema and indicate mapping from data sources into tables.

You should populate your data warehouse tables based on the data in the operational tables and spreadsheet. You do not need to insert the data into your tables. You can just show table listings in your solution document. Your sample rows should include all revenue events in the range February 10, 2017 to February 21, 2017.

1. You should identify dimensions, map dimensions to data sources, and specify dimension hierarchies. For each dimension, you should identify its data sources

and attributes in each data source. For hierarchical dimensions, you should indicate the levels from broad to narrow.

2. You should specify measures, related data sources, and measure aggregation properties.
3. Identify the grain in your dimensional design using the business needs as a guideline. You should then indicate relative storage requirements for the grain using statistics for the data sources. Using the cardinality estimates provided, you should determine either the fact table size or sparsity and then compute the unknown grain size variable. For example, you should compute sparsity if the fact table size is given.
4. Extend your analysis to design a star schema (or variation) to support inventory analysis. For each table, you should define the table name, primary key, and columns. You do not need to write complete CREATE TABLE statements.
5. Identify potential summarizability problems in your star schema and indicate preferred resolutions of the summarizability problems. For incomplete dimension-fact relationships, you should also indicate if columns in a dimension table allow null values.
6. You should populate your data warehouse tables based on the data in the sample tables and spreadsheet. You do not need to write SQL INSERT statements or insert data into database tables. You can just show table listings in your solution document. You should indicate mappings from data sources into tables. For example, a mapping may involve generating new primary key values for a data warehouse table or using a default value for a missing value.

Solution Quality

Quality is rather subjective in data warehouse designs, but some elements are less subjective. You should address these quality items in the appropriate part of your solution.

- Schema pattern: You should use a recognized schema pattern: star, constellation, or snowflake schema.
- Fact table selection: You should study fact table selection in the solution for the practice mini case for inspiration. Typically, the fact table combines a two level solution in a source schema into a single fact table. For example, an order heading and order detail are usually combined into a fact table recording the order details with dimension relationships to capture the order heading.
- Missing data in populated tables: You should ensure that your populated tables include all revenue events shown in both data sources. The best check on your schema design is to map sample rows from the data sources to the data warehouses.
- Simplicity: Typically, a schema design for a data warehouse simplifies the schemas of the underlying data sources. Simplification can involve combining some elements of data sources in decisions about dimensions and fact tables.

REFERENCES FOR FURTHER STUDY

Several references provide additional details about important parts of Chapter 13. Kimball (1996, 2003) provides more details about historical integrity covered in Section 13.2.3. The survey of summarizability concepts by Mazon et al. (2009) augments Section 13.3 on summarizability problems and patterns. The survey of methodologies for data warehouse design by Romero and Abello (2009) augments Section 13.4.2 on design methodologies. For more details about data warehouse design methodologies, you should consult Kimball et al. (1998) about the demand-driven approach, Moody and Kortink (2000) about the supply-driven approach, and Bonifati et al. (2001) about the hybrid approach.

chapter

14

Data Integration Concepts and Practices

Learning Objectives

This chapter extends design skills covered in Chapter 13 with concepts and practices about data integration. After this chapter, the student should have acquired the following knowledge and skills:

- Explain the types of data sources, data quality issues, and data cleaning tasks involved in data integration
- Gain insight about managing the complex processes of refreshing and populating a data warehouse
- Apply regular expressions to parse text fields using simple patterns
- Explain techniques for correcting and standardizing values along with the process of entity matching
- Describe the features of data integration tools for maintaining a data warehouse
- Apply a data integration tool on a task of moderate complexity
- Write Oracle SQL statements for merging change data and inserting change data into multiple fact tables

OVERVIEW

After acquiring background and detailed skills about conceptual design of data warehouses through your study of Chapter 13, you are ready to learn about data integration, a unique and vital part of data warehouse processing. Chapter 14 extends your design background with concepts and detailed skills about data integration. Together, Chapters 13 and 14 provide skills and knowledge essential to careers and work assignments involving data warehouses.

Chapter 14 emphasizes concepts and practices for data integration, fundamental to obtain business value from a data warehouse. A data integration workflow maps source data to populate data warehouse tables. You will first learn management concepts about data integration involving characteristics of change data, workflows for refresh processes, and management of refresh processing. After this conceptual background, you will learn about data cleaning techniques for parsing, standardizing values, and entity matching. In the final part of the chapter, you will learn about architectures and features of data integration tools, features of commercial products for data integration, and specialized SQL statements for data integration tasks.

14.1 DATA INTEGRATION CONCEPTS

Data integration adds value to disparate data sources that contribute to enterprise data warehouses. Data integration workflows seek to provide a single source of truth for decision making. Integrating data sources involves challenges of large volumes of data, legacy source systems, lack of standards for formats, units of measure, and integrity rules, different update frequencies, missing data, and lack of common identifiers. Data integration is a critical success factor for data warehouse projects. Many projects have failed due to unexpected difficulties in populating and maintaining a data warehouse. Organizations must make substantial investments in effort, hardware, and software to overcome challenges of data integration.

Data integration involves initially populating a data warehouse and periodically refreshing a data warehouse as data sources change. Determining data to load in a warehouse involves matching business intelligence needs to the realities of available data. Reconciling the differences among data sources is a significant challenge especially considering that source systems typically cannot be changed. As data sources change, a data warehouse should be refreshed in a timely manner to support business intelligence needs. Since data sources change at different rates, the determination of the time and content to refresh can be a significant challenge. Because of these challenges, data integration can involve significant investments in hardware, software, and personnel to achieve a satisfactory solution.

This section presents concepts of data integration. The first part describes the kinds of data sources available for populating a data warehouse. The second part describes workflow specification for maintaining a data warehouse. The final part discusses management of the process of periodic refresh of a data warehouse and initial loading of source data.

14.1.1 Sources of Data

Accessing source data presents challenges in dealing with a variety of formats and constraints on source systems. External source systems usually cannot be changed. Internal source systems usually have restrictions and high costs about changes to accommodate requirements of a data warehouse. Even if a source system can be changed, budget constraints may allow only minor changes. Source data may be stored in legacy format or modern format. Legacy format generally precludes retrieval using nonprocedural languages such as SQL. Modern format means that the source data can be accessed through a relational database or Web pages. Unless stored with formal meta data, Web pages can be difficult to parse and nonstandard across websites. Formal meta data usually involves XML data along with an XML schema to provide interpretation of the XML data.

Change data from source systems provides the basis to update a data warehouse. Change data comprises new source data (insertions) and modifications to existing source data (updates and deletions). Further, change data can affect both fact and dimension tables. The most common change data involves insertions of new facts. Insertions and updates of dimension data are less common but are still important to capture.

Change data can be classified by source system requirements and processing level as depicted in Figure 14.1 and summarized in Table 14-1. Source system requirements involve modifications to source systems to acquire change data. Typical changes to source systems are new columns such as timestamps required for queryable change data and trigger code required for cooperative change data. Since source systems are difficult to change, queryable and cooperative change data may not be available.

Processing level involves resource consumption and development required for data integration procedures. Processing of logs and snapshot change data involve substantial computing resources. The amount of processing for a log varies so its processing requirements may be larger than snapshot change data. If a source system does not already generate logs, it is unlikely that log change data can be available.

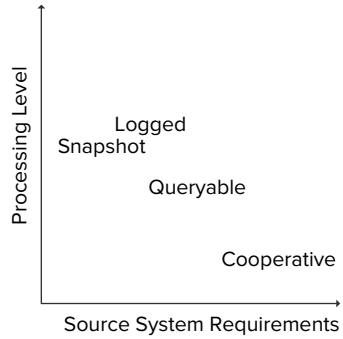


FIGURE 14.1
Classification of Change Data

Change Type	Description	Evaluation
Cooperative	Source system notification using triggers	Requires modifications to source systems
Logged	Source system activity captured in logs	Readily available but significant processing to extract useful data
Queryable	Source system queries using timestamps	Requires timestamps in data sources and non-legacy source systems
Snapshot	Periodic dumps of source data augmented with difference operations	Resource intensive for difference operations; no source system requirements so useful for legacy data

TABLE 14-1
Summary of Change Data Classification

After understanding classification of change data, you should understand details about each type of change data. Cooperative change data involves notification from a source system about changes. The notification typically occurs at transaction completion time using a trigger as depicted in Figure 14.2. A trigger can input change data immediately into a data warehouse or queue it for later input with other changes. Because cooperative change data involves modifications to both a source system and a data warehouse, it has traditionally been the least common format for change data. However, as data warehouse projects mature and legacy systems are redeveloped, cooperative change data should become more common.

Logged change data involves files that record changes or other user activity. For example, a transaction log contains every change made by a transaction and a web log contains page access histories (called clickstreams) by website visitors. Logged change data usually involves no changes to a source system as logs are readily available for most source systems.

Figure 14.3 shows an example of a web log. Substantial processing during data integration is required for web logs to decompose text even though web logs follow several standard formats. Since web logs record page visits, a transformation procedure needs substantial processing to link related log records.

As its name implies, queryable change data comes directly from a data source via a query as depicted in Figure 14.4. Queryable change data requires time stamping in a data source. Since data sources contain timestamps only for selected data, an organization must augment queryable change data with other kinds of change data.

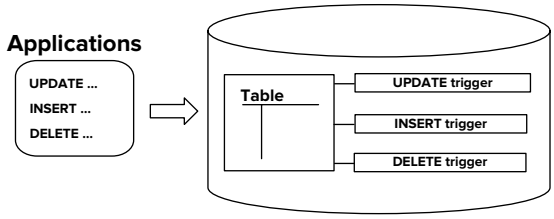


FIGURE 14.2
Processing of Cooperative Change Data

FIGURE 14.3
Example of Logged Change Data

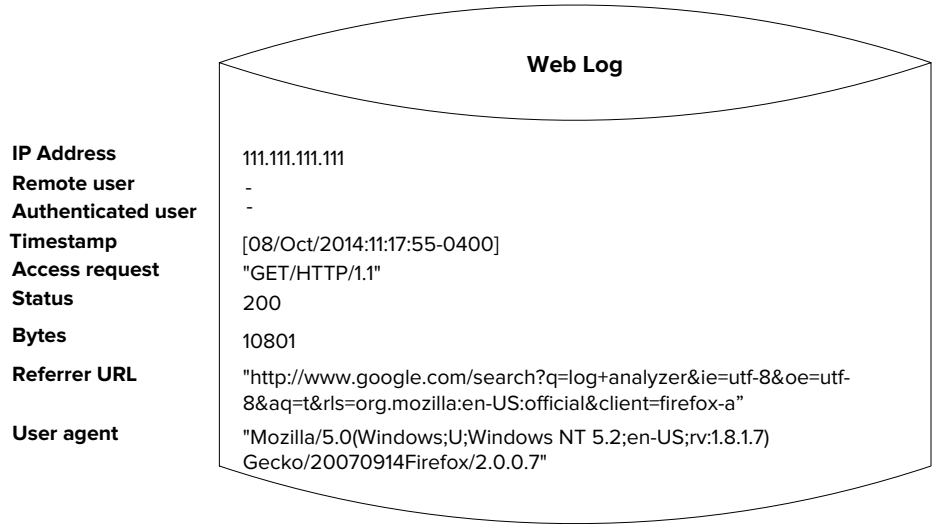
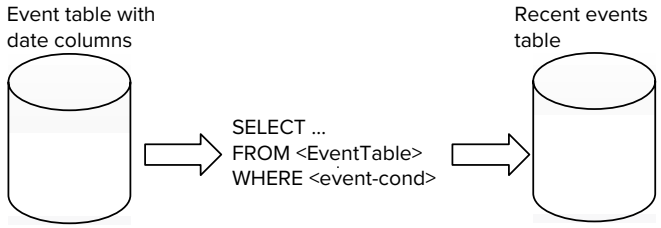


FIGURE 14.4
Processing of Queryable Change Data



An organization typically uses queryable change data for fact tables containing columns such as order date, shipment date, and hire date that are stored in operational databases.

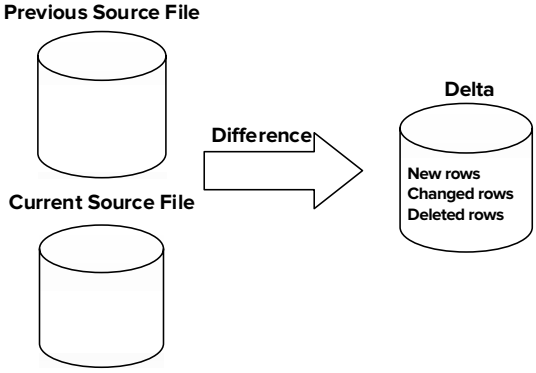
Snapshot change data involves periodic dumps of source data. To derive snapshot change data, Figure 14.5 shows a difference operation using the two most recent source files to derive the result, called a delta. To generate a delta, transformation code compares source files to identify new rows, changed rows, and deleted rows.

Snapshots have historically been the most common form of change data because of applicability. Snapshots are the only form of change data without requirements on a source system. Because computing a snapshot can be resource intensive, constraints may exist about the time and frequency of retrieving a snapshot.

14.1.2 Workflow for Maintaining a Data Warehouse

Maintaining a data warehouse involves a variety of tasks that manipulate change data from source systems. Figure 14.6 presents a generic workflow that organizes the tasks.

FIGURE 14.5
Processing of Snapshot Change Data



The preparation phase manipulates change data from individual source systems. Extraction involves the retrieval of data from an individual source system. Transportation involves movement of the extracted data to a staging area. Cleaning involves a variety of tasks to standardize and improve the quality of the extracted data. Auditing involves recording results of the cleaning process, performing completeness and reasonableness checks, and handling exceptions.

The integration phase merges the separate, cleaned sources into one source. Merging can involve the removal of inconsistencies among the source data. Audit processes record results of the merging process, performs completeness and reasonableness checks, and handles exceptions.

The update phase propagates integrated change data to various parts of a data warehouse including fact and dimension tables, materialized views, stored data cubes, and data marts. After propagation, notification can be sent to user groups and administrators.

The preparation and integration phases should resolve data quality problems as summarized in Table 14-2. Data from legacy systems are typically dirty, meaning that they may not conform to enterprise-wide data quality standards. If directly loaded, dirty data may lead to poor decision making. To resolve data quality problems, the auditing task should include exception handling. Exceptions can be noted in a log file and then manually addressed. Over time, exceptions should decrease as data quality standards are improved on internal data sources.

In addition to exception handling, the auditing task should include completeness checks and reasonableness checks. A completeness check counts the number of reporting units to ensure that all have reported during a given period. A reasonableness check determines whether key facts fall in predetermined bounds and are a realistic extrapolation of previous history. Exceptions may require reconciliation by business analysts before propagation to a data warehouse.

The maintenance process varies among data sources. The workflow should be customized to fit the requirements of each data source. For example, auditing may be minimized for high-quality data sources. Data integration tools, depicted in Section 14.3, support graphical workflow modeling to allow customization of workflows by organizations.

In addition to periodic maintenance, data integration involves the initial population of a data warehouse as depicted in Figure 14.7. The initial loading process is more open ended than refresh processing. Time requirements for discovering and resolving data quality problems can be difficult to estimate. Profiling tools can facilitate discovery of data quality problems. Data quality problems are usually resolved through data integration procedures. If owners of source data cooperate, resolution can involve

FIGURE 14.6
Generic Workflow for Data Warehouse Maintenance

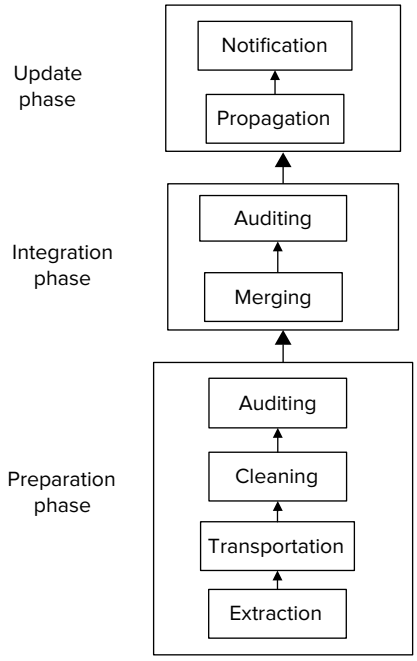


TABLE 14-2
Typical Data Quality Problems

Multiple identifiers: some data sources may use different primary keys for the same entity such as different customer numbers
Multiple names: the same field may be represented using different field names
Different units: measures and dimensions may have different units and granularities.
Missing values: data may not exist in some databases; to compensate for missing values, different default values may be used across data sources
Orphaned transactions: some transactions may be missing important parts such as an order without a customer
Non-standard text data: some data sources may combine multiple columns into a single text column such as addresses containing multiple components. In addition, format of address components can vary across data sources.
Conflicting data: some data sources may have conflicting data such as different customer addresses
Different update times: some data sources may perform updates at different intervals

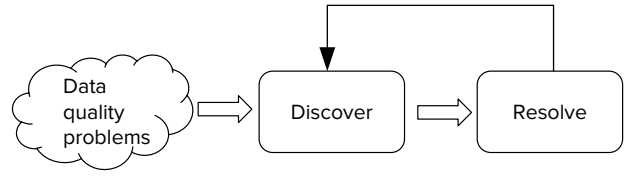


FIGURE 14.7
Overview of the Loading Process for a Data Warehouse

changes to source systems. The initial population process should be performed for each major extension of a data warehouse.

14.1.3 Managing the Refresh Process

Refreshing a data warehouse is a complex process that involves management of time differences between updating of data sources and updating of the related data warehouse objects (tables, materialized views, data cubes, data marts). In Figure 14.8, valid time lag is the difference between the occurrence of an event in the real world (valid time) and the storage of the event in an operational database (transaction time). Load time lag is the difference between transaction time and the storage of an event in a data warehouse (load time). For internal data sources, there may be some control over valid time lag. For external data sources, an organization usually has no control over valid time lag. Thus, a data warehouse administrator has most control over load time lag.

The framework in Figure 14.8 implies that data sources can change independently leading to different change rates for fact and dimension tables. Fact tables generally record completed events such as orders, shipments, and purchases with links to related dimensions. For example, inserting a row in a sales fact table requires foreign keys that reference dimension tables for customers, stores, time, and items. However, updates and insertions to related dimension tables may occur at different times than fact events. For example, a customer may move or an item may change in price at different times than orders, shipments, or inventory purchases. Due to multiple change rates, a data warehouse administrator should manage load time lag separately for dimension tables and fact tables.

Managing the refresh process involves careful balancing of factors influencing the primary objective as depicted in Figure 14.9. The primary objective in managing the refresh process is to determine the refresh frequency and detailed refresh schedules for each data source. The optimal refresh frequency maximizes the net refresh benefit

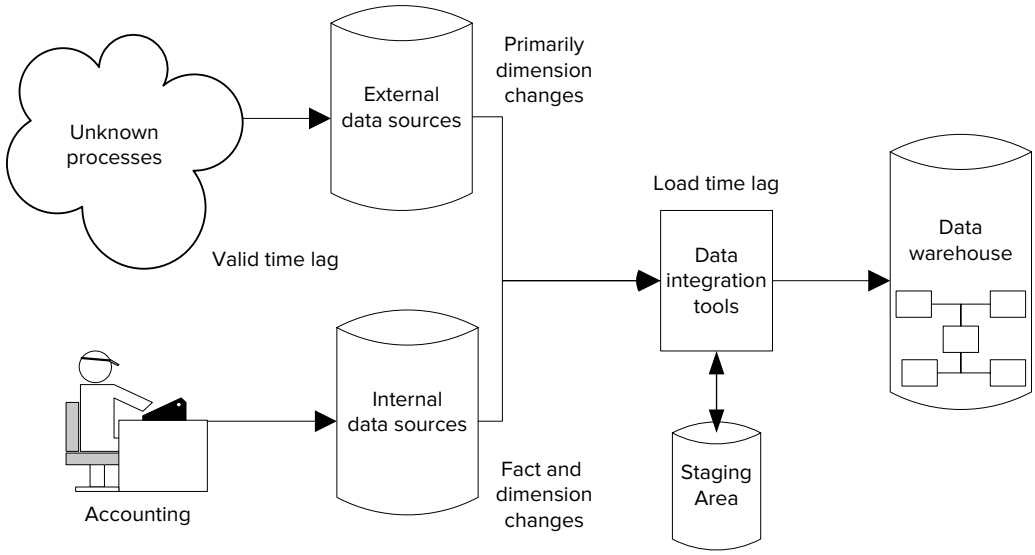


FIGURE 14.8
Overview of the Data Warehouse Refresh Process

TABLE 14-3
Summary of Refresh Constraints

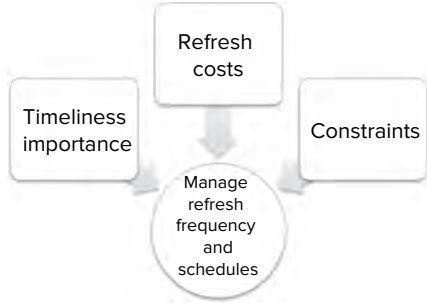
Constraint Type	Description
Source access	Restrictions on the time and frequency of extracting change data
Integration	Restrictions that require concurrent reconciliation of change data
Completeness/consistency	Restrictions that require loading of change data in the same refresh period
Availability	Load scheduling restrictions due to resource issues including storage capacity, online availability, and server usage

while satisfying important constraints. The net refresh benefit is the value of data timeliness minus the cost of refresh.

The value of data timeliness depends on the sensitivity of decision making to the currency of data. Some decisions are very time sensitive such as inventory decisions for the product mix in stores. Other decisions are not so time sensitive such as store location decisions.

The cost to refresh a data warehouse includes both computing and human resources. Computing resources are necessary for all tasks in a maintenance workflow. Human resources may be necessary in the auditing tasks during the preparation and integration phases. The level of data quality in source data also affects the level of human resources required. The development effort to use data integration tools and write custom software is not part of refresh cost unless development cost occurs with each refresh.

FIGURE 14.9
Factors Influencing Refresh Process Objective



An important distinction involves the fixed cost and the variable cost of refresh. Large fixed cost encourages less frequent refresh because fixed cost occurs with each refresh. Fixed cost may include startup and shutdown effort as well as fixed resource rental.

Along with balancing the value of timeliness against the cost of refresh, a data warehouse administrator must satisfy constraints on the refresh process as summarized in Table 14-3. Constraints on either a data warehouse or a source system may restrict frequent refresh. Source access constraints can be due to legacy technology with restricted scalability for internal data sources or coordination problems for external data sources. Integration constraints often involve identification of common entities such as customers and transactions across data sources. Completeness/consistency constraints can involve maintenance of the same time period in change data or inclusion of change data from each data source for completeness. Data warehouse availability often involves conflicts between online availability and warehouse loading.

14.2 DATA CLEANING TECHNIQUES

Data cleaning is an important part of a data integration workflow to resolve the data quality problems summarized in Table 14-2. This section explains common data cleaning techniques and demonstrates their usage to resolve data quality problems. Data integration tools, presented in the next section, support nonprocedural specification for many data cleaning techniques. This section covers parsing of multipurpose text fields, correcting and standardizing values for missing and inconsistent values, and matching entities among data sources.

14.2.1 String Parsing with Regular Expressions

Parsing decomposes complex objects into their constituent parts. For data integration, parsing is important for decomposing multipurpose text data into individual fields. Parsing of physical addresses, phone numbers, and email addresses are typical transformations for marketing data warehouses. To facilitate target marketing analysis, these composite fields should be decomposed into standard parts. For example, data

FIGURE 14.10
Parsing Name and Address into Constituent Fields

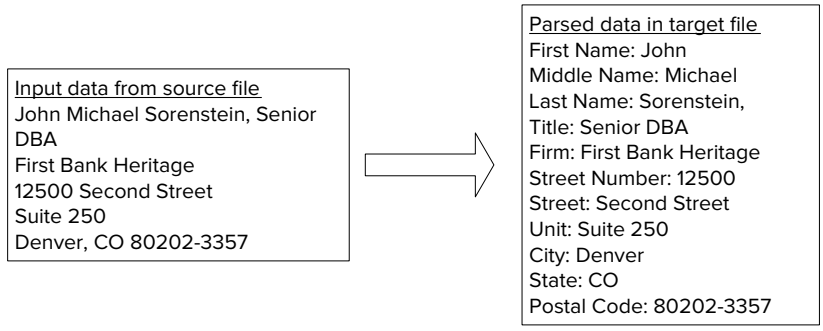
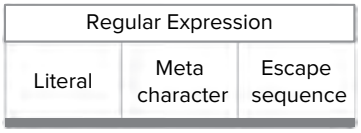


FIGURE 14.11
Regular Expression Components



sources that contain addresses in a single field typically require parsing into standard fields such as the street number, street, city, state, country, and postal code. Figure 14.10 demonstrates parsing of a customer's name and address into constituent fields.

Regular expressions are pattern specifications that are important elements of parsing in data integration tasks. Chapter 4 provided a brief introduction to inexact matching in SQL SELECT statements although this subsection provides more details because of the importance of regular expressions in data integration. As depicted in Figure 14.11, a regular expression (or regex for short) contains literals, metacharacters, and escape characters together that define a search pattern. A literal is a character to match exactly. A metacharacter is one or more special characters having a unique meaning such as the character ^ (circumflex or caret). An escape sequence removes the special meaning of a metacharacter so that it is matched as a literal. In a regular expression an escape sequence involves placing the metacharacter \ (backslash) in front of a metacharacter.

Regular Expression
a pattern specification that is important for parsing of multipurpose text fields in data integration tasks. A regular expression (or regex for short) contains literals (exactly matching characters), metacharacters (special meaning characters), and escape characters (remove special meaning of metacharacters).

To perform pattern matching, a user provides a regular expression known as the search expression and a target string as depicted in Figure 14.12. The search expression specifies the pattern to locate in the target string. In this example, the search expression contains 7 meta characters (^ (caret or circumflex), [(left square bracket),] (right square bracket), +, -, \ (backslash), and $), 6 literal characters (a, z, c, o, m, .), and one escape sequence (\.) to turn off the special meaning of the period symbol. The match result shows the part of the target string matching the search expression.

Pattern matching with literals is simple although not important by itself. Table 14-4 contains simple regular expressions using only literals. Note that literals match the first appearance of the target string. The quotation marks are not part of the regular expressions or target strings. A regular expression engine compares a regular expression to a pattern, providing a true/false response along with an optional indication of the first matching character.

FIGURE 14.12
Pattern Matching Example

Search Expression			Target String	Match Result
	^[a-z]+\.com$		abc.com	abc.com
Meta characters	*Literals*	*Escacpe sequence*		
^	c	\.		
[	o			
]	m			
+	a			
-	z			
\	.			
$				

Search Expression	Target String	Evaluation
"a"	"John Michael"	True; matches the 9th position
"a"	"Senior DBA"	False; match is case sensitive
"Co"	"Denver, CO"	False; "o" not matched because of case sensitivity
"Suite 2"	"Suite 250"	True; match begins in 1st position; spaces are literals

TABLE 14-4
Regular Expression Examples with Literals Only

Metacharacters, characters with special meaning within a search expression, provide the power of regular expressions. To help you understand prominent metacharacters, Figure 14.13 provides a classification and Table 14-5 provides brief explanations. The iteration or quantifier metacharacters, ?, *, +, and {}, support matches on consecutive characters. The range metacharacters, [], match a single character from a range of specified characters. The position metacharacters or anchors, ^, $, and . (period character), support matching in specified places in a target string. The alteration metacharacter, |, supports optional parts of search patterns. Table 14-6 provides examples for each metacharacter along with comments about the examples.

The backslash (\) escape character removes the meaning of a metacharacter allowing a metacharacter to be used as a literal. For example, the regular expression "abc*" matches the search string "ddd abc*" but not "fabc". The backslash only has the escape meaning before a metacharacter. A backslash also combines with certain literal characters to designate character classes, predefined sets of characters useful in patterns. For example "\d" refers to the digits 0 to 9 and "\w" refers to word characters (letters, digits, and underscore).

For complex matching requirements, a regular expression can match different subparts or groups in a target string. Groups facilitate parsing because parts of a string can be matched and used for later processing. For example, a date string can be parsed into year, month, and day components using groups. The parentheses metacharacters () provide matching on groups in a search string. Matching with groups provides a list

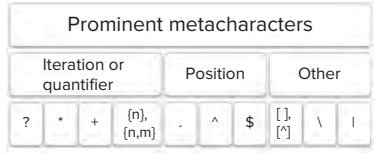


FIGURE 14.13
Classification of Prominent Metacharacters

Metacharacter	Type	Meaning
?	Iteration	Matches preceding character 0 or 1 time
*	Iteration	Matches preceding character 0 or more times
+	Iteration	Matches preceding character 1 or more times
{n}	Iteration	Matches preceding character exactly n times
{n,m}	Iteration	Matches preceding character at least n times and at most m times
[]	Range	Matches enclosed character one time
^	Position	Matches the following search string at the beginning of the search string; using ^ only has meaning as the first character in a regular expression; ^ used inside [ ] has a different meaning
^	Range	Negation of search pattern if ^ is inside [ ]
$	Position	Matches the preceding search pattern at the end of the search string; $ only has meaning as the last character in a regular expression.
.	Position	Matches any character except a newline character at the specified position only
|	Alteration	Matches either pattern to the left or right of the | character.

TABLE 14-5
Meaning of Important Metacharacters

TABLE 14-6
Regular Expression Examples with Metacharacters

Search Expression	Target Strings	Evaluation
“colou?r”	“color”, “colour”	Matches both search strings; Matches preceding character 0 times in first search string
“tre*”	“tree”, “tread”, “trough”	Matches all three search strings; Matches preceding character 0 times in third search string
“tre+”	“tree”, “tread”, “trough”	Does not match the third search string; The + metacharacter requires matching on at least one character.
“[abcd]”	“dog”, “fond” , “pen”	Matches first two strings but not the third string; Matches strings that contain any enclosed character
“[0-9]{3}-[0-9]{4}”	“123-4567”, “1234-567”	Matches first string but not the second string; Regular expression uses iteration and range metacharacters.
“ba{2,3}b”	“baab”, “baaab”, “bab”, “baaaab”	Matches first two strings but not the last two strings; four consecutive “a” characters do not match in the last search string.
“^win”	“erwin”, “window”	Does not match the first search string because “win” does not appear in the beginning of the search string
“win$”	“erwin”, “window”	Does not match the second search string because “win” does not appear at the end of the search string
“[^0-9]+”	“123”, “abc”, “a456”	Matches the second and third search strings; ^ negates the string pattern inside the [] matching any non-digit.
“abc.e*”	“fabc”, “fabcd”, “fabcee”	Matches the second and third search strings; Regular expression requires a character following “abc”.
“dog|cat|frog”	“a dog”, “cat friend”, “frogman”	Matches all three search strings because of the alteration metacharacters in the pattern

of matched groups. For example, the search expression “a(b*)c” matched against the target string “abbc” produces two matched groups: (Group 0) “abbc” matching the entire regular expression and (Group 1) “bb” matching the regular expression group “(b*)”. As additional features, a pattern can have more than one group and groups can be nested. For example, the search expression “(d(e*))*(c+)” contains group 1 with the pattern “(d(e*))”, group 2 with the pattern “(e*)”, and group 3 with the pattern “(c+)”. In addition, the entire pattern is group 0.

This introduction only scratches the surface of the power and complexity of regular expressions. There are other important aspects such as conditional matching and additional grouping features. Data integration tools typically support regular expressions along with a replacement function to substitute specified contents for matching parts of a target string. Libraries of regular expressions support parsing of common fields such as names, addresses, phone numbers, and email addresses. Developing high quality regular expressions is tedious and difficult due to complexity and exceptions in text data such as addresses.

Regular expressions support context-fee parsing when the meaning of a symbol does not depend on its relationship to other symbols in text. Natural language processing provides parsing and understanding natural language text that is context-sensitive. Sentiment analysis has emerged as an important area of natural language processing for business intelligence. Sentiment analysis involves identification and categorization of opinions in text such as recommendations posted in product reviews. Due to varying business intelligence requirements, sentiment analysis is typically performed in

text mining, a business intelligence application using a data warehouse rather than a part of data integration.

14.2.2 Correcting and Standardizing Values

Correcting values involves resolution of missing and conflicting values. Reasonable resolution requires understanding the reason for missing values. Missing values in summarizability problems (incomplete drill-downs, incomplete roll-ups, and incomplete dimension-fact relationships) can usually be resolved through default values as indicated in Chapter 13. Missing values in these cases usually involve unallocated relationships. For example, a department is not allocated to a division, a division lacks departments, or a sale involves an anonymous customer. In these situations, default values for unattached entities are appropriate. Missing values inapplicable to an entity can often be resolved through default values. For example, unmarried students and customers will not have values for attributes related to spouses. The default value could be the same value as the corresponding value for the entity. For example, student age and spouse age would be set to the same value for unmarried students.

Missing values that are unknown rather than inapplicable are more difficult to resolve. For example, missing dates of birth, addresses (or parts of an address), and grade point averages are more difficult to resolve. One approach to unknown values involves typical values. For unknown numeric values, a median or average value can be used. For unknown non numeric values, the mode (most frequent value) can be used. Typical values can be refined through conditional probability calculations. The typical value would the one most likely given other attribute values of the entity. More complex approaches will predict missing values using data mining algorithms. These approaches extend the conditional probability approach by additional decision criteria using a large sample.

Detailed investigations possibly conducted using search services can resolve some cases of unknown values and conflicting values. Figure 14.14 demonstrates the result of an investigation to determine the middle name and street address in an employee record. A map and knowledge about the location of a building was used to obtain the street address. Other employee files were used for the missing middle name. For conflicting values, the more recent value is preferred although a data source may lack timestamps to determine the update time. Without a timestamp, the more credible source may be preferred.

Standardization involves business rules to transform values into preferred representations. Both standard and custom business rules can be developed. Standardization is typically applied to units of measure and abbreviations. To augment searching, business rules can provide alternative values in the results of data warehouse queries. Data standardization services can be purchased for names, addresses, and product details although customization may be necessary.

The approaches described in this subsection are reactive, attempting to resolve problems occurring in existing data sources. Proactive approaches can be more cost effective if changes in data collection procedures can be made in different parts of

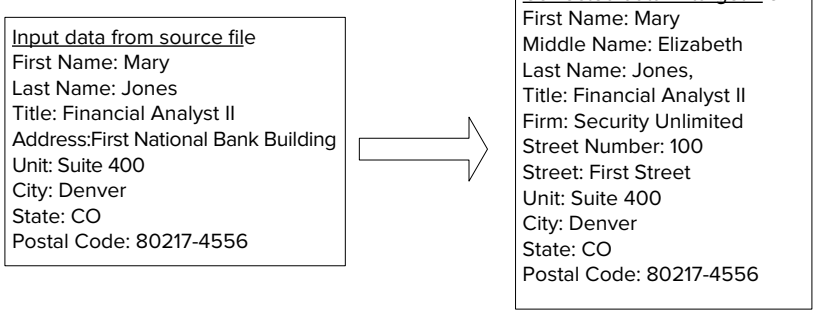


FIGURE 14.14
Completed Missing Values through an Investigation

an organization. Standards can be facilitated by XML schemas with rules about data interchange. It may even be possible to apply standards to external data sources if external users derive benefits from a data warehouse.

14.2.3 Entity Matching

Entity matching refers to identification of duplicate records when no common, reliable identifier exists. The classic application involves identification of duplicate customers in lists from different firms. Because a common identifier does not exist, duplicates must be identified from other common attributes such as names, address components, phone numbers, and ages. Because these common attributes come from different data sources, inconsistency and non standard representations may exist complicating the matching process.

Despite the difficulty of the entity matching problem, it is important in many applications. Marketing is the most prominent area as firms are often interested in expanding their customer bases. Merging of firms typically triggers a major customer matching effort. Law enforcement agencies need to link crimes to suspects and combine aliases into one suspect. Fraud detection must resolve individuals who claim benefits under different identifiers when the individuals are the same person. For example, the same person may fraudulently file multiple tax returns to receive tax credits. Other applications involve insurance industry needs to link crash and injury reports and gene identification in lab and research reports.

Entity Matching Example The simple example in Table 14-7 depicts difficulties of entity matching. The data sources do not have a common identifier to reliably match so non unique fields must be used.

Data source 1 has the maiden (pre marriage) name and work address. Data source 2 has the marital name and home address. The middle name, job title, and firm also have different values. The state and country fields have different values due to usage of abbreviations and non-abbreviations. Experience indicates these records are likely a match because of proximity of Bothell and Redmond in Washington state, matching first name with same uncommon spelling, sharing part of the last name, and matching employment after standardizing the firm and job titles. The last name difference can be explained by combining last names after marriage.

An entity matching algorithm without this domain expertise, may indicate an inconclusive match rather than a likely match. A costly investigation by a domain expert may be necessary to resolve this inconclusive match.

The example in Table 14-7 shows common fields for two data sources. Matching is more complex if data sources contain unstructured data such as text, images, and events rather than common, structured fields.

TABLE 14-7
Entity Matching Example

	Data Source	
Fields	**Data Source 1**	**Data Source 2**
First Name	Aimee	Aimee
Middle Name	Christina	C.
Last Name	Parker	Parker-Lewis
Job Title	Product Manager	Prod. Mgr.
Firm	Microsoft Corporation	Microsoft
Street	15580 NE 31st Street	16517 78th Place NE
City	Redmond	Bothell
State	WA	Washington
Postal Code	98052	98020
Country	USA	United States

Entity Matching Outcomes To obtain a more precise understanding of **entity matching**, the outcomes of comparing two cases should be understood. The outcomes of entity identification are similar to standard classification problems. The rows in the Table 14-8 represent predictions and the columns represent actual results of matching two entities for duplication. A true match involves a predicted match and an actual match allowing the two records to be combined correctly. A false match involves a predicted match but an actual non match resulting in two records combined that should have remained separate. A false non match involves a prediction of non match but an actual match resulting in two records remaining separate that should be combined. A true non match involves a prediction of non match and actual non match resulting in two separate records remaining separate. The possible match situations involve predictions with too much uncertainty. Investigation is required to resolve cases with high uncertainty in match prediction.

Entity Matching
identification of duplicate records when no common, reliable identifier exists. Because a common identifier does not exist, duplicates must be identified from other common attributes such as names, address components, phone numbers, and ages. The result of matching two records may be a true match, false match, false non match, true non match, or additional investigation.

In an economic sense, entity matching procedures should balance the benefits of consolidated entity lists (true matches and true non matches) against the costs of incorrect actions (false matches and false non matches) plus investigation costs. The costs of false matches are typically the largest as a false match eliminates a potential customer. Determining uncertainty levels leading to investigations is also important if investigation cost is substantial. Investigation costs may be labor intensive, likely more than the cost of false non matches.

Entity Matching Approaches Many approaches to entity identification have been developed but no dominant approach has emerged. Data mining tools typically include methods for entity identification. In addition, commercial services with customization to individual data source requirements can match entities but usually with relatively high cost. Before deciding on a solution, an organization should investigate data quality levels in data sources to consolidate. To improve entity identification results, investment to improve consistency and incompleteness in underlying data sources is usually worthwhile.

Entity matching algorithms use **quasi identifiers** to compensate for missing common identifiers. A quasi identifier is a collection of columns almost unique in combination. In a study published in 2000, Sweeney demonstrated that 87% of the US population can be identified by a combination of gender, birth date, and postal code. Other examples of quasi identifiers are name components, location components, profession, and race. Before an entity matching algorithm can be applied, common quasi identifiers should be determined. Poor data quality such as missing values and unknown update times complicate choices for quasi identifiers.

Quasi Identifier
a combination of columns that is almost unique for an entity. Entity matching algorithms use quasi identifiers to compensate for missing common identifiers.

For quasi identifiers with a text data type, entity matching algorithms typically use distance measures for text comparisons. Often, text fields in a quasi identifier will not match exactly so similarity assessment must be performed. Three common measures of text similarity are edit distance, N-gram distance, and phonetic distance. Edit distance is the number of deletions, insertions, or substitutions required to transform a source string into a target string. *N*-gram distance breaks text into subsequences of length *n* and then measures intersections among the *n*-grams. Phonetic distance codes words into standard consonant sounds. Phonetic distance has many applications in law enforcement to account for different name spellings. Data integration tools typically contain functions for each distance measure.

TABLE 14-8
Entity Matching Outcomes

	Actual	
Predicted	**Match**	**Non Match**
Match	True match	False match
Possible Match	Investigation	Investigation
Non Match	False non match	True non match

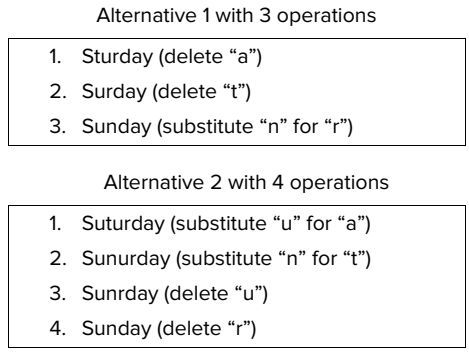


FIGURE 14.15
Edit Distance Example for "Saturday" to "Sunday"

To depict edit distance, Figure 14.15 demonstrates a simple example. In this example, the edit distance to transform "Saturday" into "Sunday" is 3 operations. This example shows two sequences of operations. The first sequence involves two deletions ("a" and "t") are followed by substitution of "n" for "r". The second sequence involves two substitutions ("u" for "a" followed by "n" for "t") and two deletions ("u" and "r"). The edit distance equals 3 as the first alternative contains fewer operations than the second alternative.

The example in Figure 14.15 shows only two sequences of operations so finding the minimal number of operations is easy. For more complex text values, a large number of sequences must be evaluated to find the minimal solution.

Customer data integration is a generalization of entity matching for marketing data warehouses. Customer data integration promotes the sharing of customer data between both front-office and back-office processes as well as with business-to-consumer and business-to-business web-enabled systems. Customer data integration provides an enterprise customer hub that serves as a central repository of customer information reconciled from multiple data sources, both internal and external. The customer data integration market is comprised of process and technology solutions for recognizing a customer at any business point while aggregating and delivering timely and accurate data about customers.

14.3 DATA INTEGRATION TOOLS

To manage complexity in data integration processes to populate and refresh a data warehouse, software products for data integration have been developed. In earlier years of data warehouse development, data integration involved tedious coding for data cleaning tasks and data source connectivity. Many project failures partly resulted from unexpected difficulties during the labor intensive process to develop data integration solutions. In addition, organizations experienced difficulty meeting performance requirements because of the resource intensive nature of refresh processing.

Data Integration Tools
software tools for extraction, transformation, and loading of change data from data sources to a data warehouse. Enterprise tools provide integrated development environments supporting a full range of data integration tasks.

This section provides broad coverage of **data integration tools** linked with practice exercises on the textbook's website for developing skills on a particular tool. This section begins with a broad coverage of common features and architectures in data integration tools. To provide more detail about features, the next three sections present major data integration tools, Talend Open Studio, Pentaho Data Integration, and the Oracle Data Integrator. The final section explains SQL statements applicable to selected data integration tasks.

14.3.1 Architectures and Features of Data Integration Tools

Software vendors realized the potential to improve software development productivity and refresh performance. Data integration software have evolved from independent tools to integrated development environments supporting a full range of data integration tasks, graphical and non-procedural specification, and code generation to minimize custom coding. Improved performance is a more recent feature of data

integration tools. Many data integration tools now provide features such as scalable parallel processing to help organizations meet demanding performance requirements of refresh processing.

Marketplace and Architectures The vibrant marketplace for data integration products and services contains third party vendors and DBMS vendors offering both proprietary and open source products along with a wide range of services. Third party vendors emphasize support for a variety of DBMS products. DBMS vendors leverage relational database support for data warehouse implementation. At the end of 2016, a Gartner report estimated software revenue from data integration tools at $2.7 billion with annual growth rate of 6.3 percent to grow to $4 billion in 2021. In the Gartner market summary for 2017, Informatica has maintained market leadership for more than a decade. As depicted in Figure 14.16, the Gartner market summary classifies firms by vision and execution. Vision involves innovation level, geographic areas, financial performance, and platform support. Execution involves market share, acceptance of new technologies, and credibility. Talend is the only firm with a base open source product and enterprise subscription service in the top quadrant. The overall marketplace is fluid with acquisitions and new product developments likely having strong influence on future market structure.

Two major architectures dominate enterprise data integration tools. The Extraction, Transformation, and Loading (ETL) architecture performs transformation before loading as shown in Figure 14.17a. Transformations and data quality checks are performed by a dedicated ETL engine before loading transformed data into target data warehouse tables. The Extraction, Loading, and Transformation (ELT) architecture performs data transformations and data quality checks after loading as depicted in Figure 14.17b. The ELT architecture relies on a relational DBMS to generate SQL statements and procedures and move data between tables.

ETL architecture supporters emphasize DBMS independence of ETL engines, while ELT architecture supporters emphasize superior optimization technology in relational DBMS engines. ETL architectures can usually support more complex operations in a single transformation than ELT architectures, but ELT architecture may use less network bandwidth. Some data integration tools support both architectures so the distinction between the architectures may blur somewhat in the future. In addition, a combination of ETL and ELT processing may provide better performance for enterprise data warehouses so the demand for both architectures should grow without either architecture dominating.

Integrated Development Environment
an essential feature of a data integration tool. Integrated development environments (IDEs) support complex software projects with a source code editor, visual specification tools, debugger, and code generator packaged in a convenient graphical interface. Since data integration involves complex software development, IDEs have become an essential feature for data integration tools.

Features Data integration tools provide essential and secondary features as depicted in Figure 14.18 and summarized in Table 14-8. **Integrated development environments** (IDEs) support complex software projects with a source code editor, visual

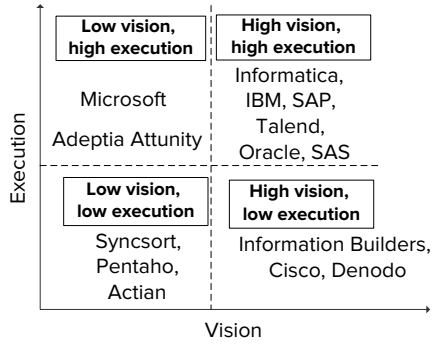


FIGURE 14.16
Summary of Gartner's 2017 Market for Data Integration Tools[1]

[1] Adapted from Beyer, M., Thoo, E., Zaidi, E, and Greenwald, R. "Gartner Magic Quadrant for Data Integration Tools," Gartner, August 2017.

FIGURE 14.17
ETL versus ELT Architectures for Data Integration

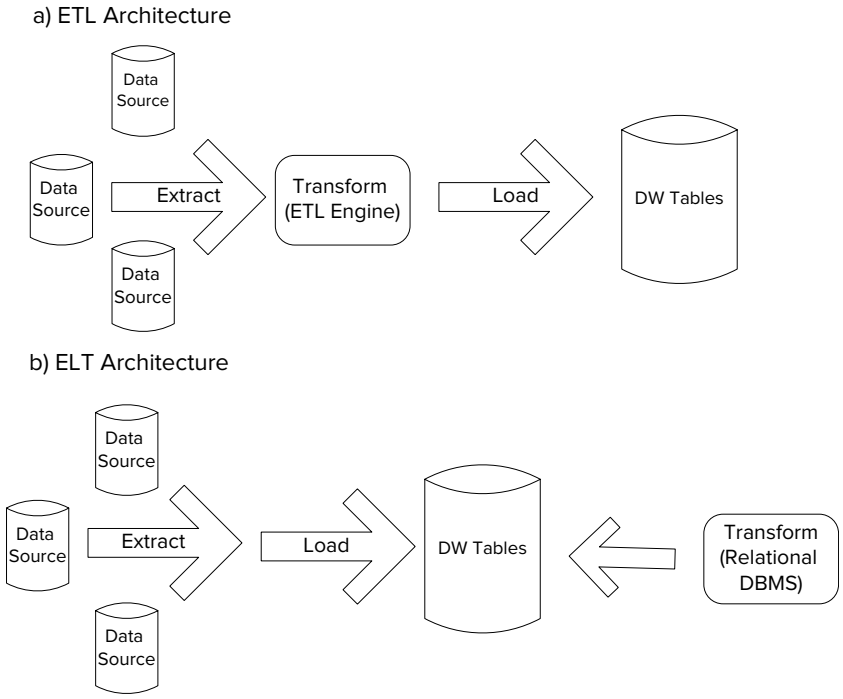
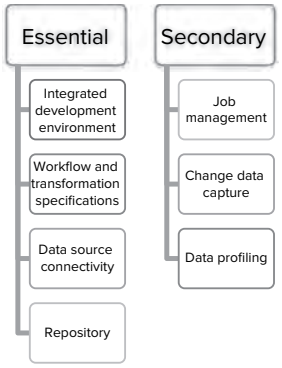


FIGURE 14.18
Essential and Secondary Features of Data Integration Tools



specification tools, debugger, and code generator packaged in a convenient graphical interface. Since data integration involves complex software development, IDEs have become essential for data integration tools. Most data integration tools use rule and graphical specifications rather than procedural coding to indicate workflow and transformations. Some tools can generate code that can be customized for more flexibility. Data source connectivity supports numerous types of data sources including files, databases, XML, and email. A repository stores workflow designs, transformations, data source connections, and other aspects of data integration tasks.

Secondary features are provided by most products, but sometimes require an extended product with an additional license. Job management involves monitoring and scheduling of workflows for execution of data integration workflows. Scheduling typically handles base schedules, repetition, conditional execution, and stepping through a workflow to identify problems. Monitoring provides logging of events, performance alerts, and reports.

TABLE 14-8
Description of Typical Features of Enterprise Data Integration Tools

Feature	Description
Workflow specification	Graphical specification of workflow designs and executions
Transformation specification	Non procedural specification of data transformations for data types, calculations, lookups, aggregations, and string manipulations
Job management	Graphical display of workflow progress; Tools for optimized scheduling of jobs using parallel processing
Data profiling	Non procedural specification to understand data characteristics and data quality levels
Change data capture	Synchronous (via triggers) and asynchronous capture of source data
Integrated development environment	Graphical development environment for all data integration functions along with team support and version control
Repository	Database for all data integration specifications, jobs, users, and data capture results
Database/file connectivity	Connection to a large collection of databases and file types

Data profiling helps a data warehouse administrator and data owners understand and improve data quality in data sources. A data profiling tool can reduce unexpected delays and surprises in populating and refreshing a data warehouse. The Microsoft SQL Server data profiling tool provides a number of functions to understand data quality.

- Descriptive statistics about the distribution of values in a column
- Null value ratio
- Uniqueness showing the number of distinct values for a column and combinations of columns
- Column relationships showing functional dependency coverage and inclusion coverage for foreign keys

Change data capture typically uses a publish and subscribe model to control change data availability and notify subscribers about change data availability. Figure 14.19 shows a publisher, the data owner, providing authorization to extract source data. Triggers can capture change data to change tables or logs. A log extraction process transforms log data and inserts it into change tables. A subscriber uses published change data in a data warehouse for decision making.

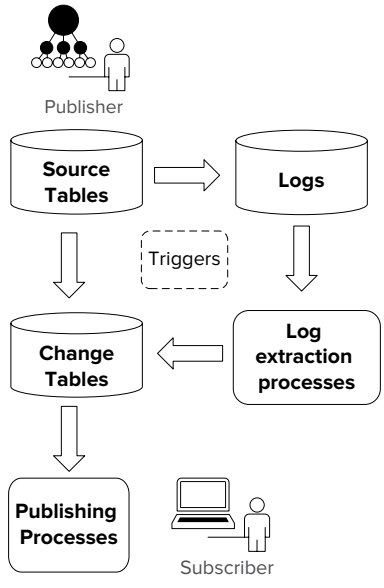
The following subsections provide more detail about essential features in Talend Open Studio, Pentaho Data Integration, and Oracle Data Integrator. Talend and Pentaho offer open source base products and extended products with a subscription service. The Oracle Data Integrator supports the ELT architecture, while Talend and Pentaho primarily support the ETL architecture.

14.3.2 Talend Open Studio

Talend provides a base open source product along with subscription services. The community edition uses a standard open source license. Talend provides subscription services for enterprise editions with extended product features and technical support. Talend offers product extensions for big data with parallel and real-time processing options, data quality with a library of data quality tests, and master data management with data profiling. The Talend Open Studio primarily supports the ETL architecture although it provides a component for the ELT architecture.

Talend Open Studio for Data Integration supports graphical job design, a palette of data transformation components, a metadata repository, job monitoring, and database connectivity. The job design component provides graphical specification to define and connect transformation components. The job execution component makes connections to data sources (databases, files, and XML data), executes data transformations, and graphically displays execution results. The repository component allows retrievals of job designs, source schemas, transformation components, and job executions.

FIGURE 14.19
Overview of Change Data Capture



Talend provides a convenient IDE for job design and execution as depicted in Figure 14.20. The repository pane contains components of a job design as well as metadata about data source schemas, code, SQL templates, and documentation. The design pane contains the outline of job components and a code viewer for components with custom code. The component palette contains a list of folders containing components. An analyst can drag a component from an open folder into the job design canvas. After placing components in the Canvas, an analyst connects them with data flows. The job pane contains job properties and execution details. The Canvas shows a job design or execution. In this snapshot, the Canvas contains a job design for an interval match with four components and three data flows. During job execution, the IDE annotates each part of a job design in the Canvas with execution details such as run times and number of objects read or written.

Table 14-9 shows icons for selected components of selected categories of Talend components. Each icon contains a short text description to clarify the

FIGURE 14.20
Integrated Development Environment for Talend Open Studio

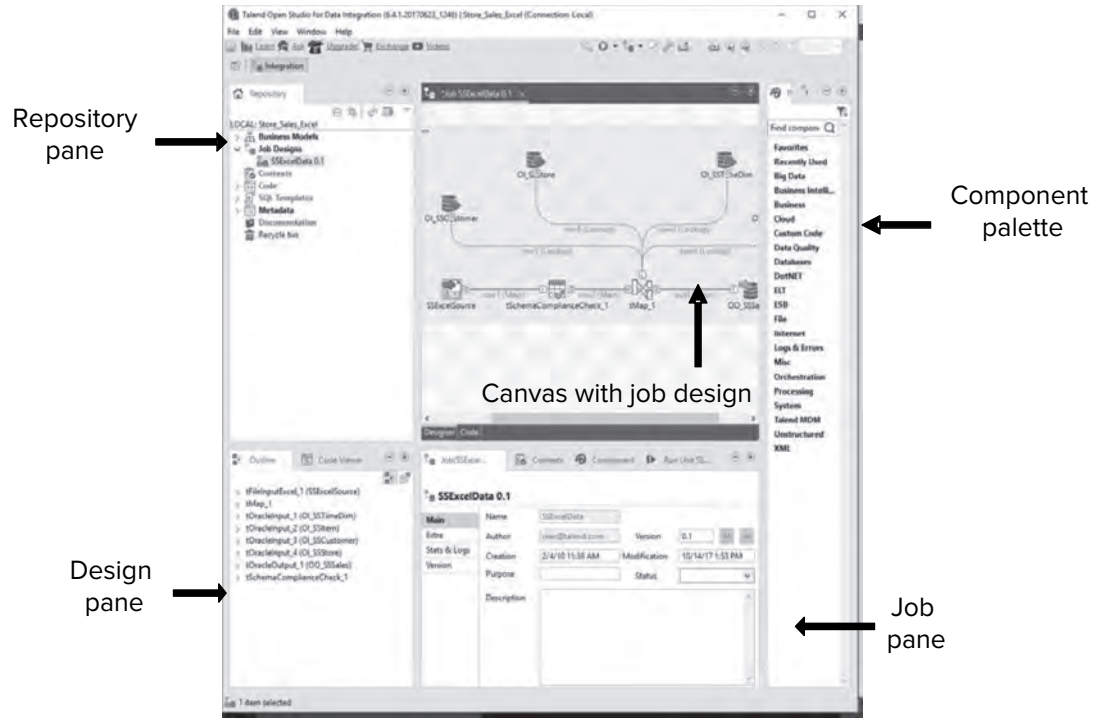


TABLE 14-9
Prominent Component Categories in Talend Open Studio

Category	Prominent Components		
Data quality	tUniqRow	tIntervalMatch	tSchemaComplianceCheck
Databases	tOracleConnection	tOracleInput	tOracleOutput
ELT	tELTOracleInput	tELTOracleOutput	tCombinedSQLFilter
Processing	tMap	tSortRow	tConvertType
Internet	tHttpRequest	tSendMail	tSocketInput
XML	tExtractXMLField	tXSDValidator	tXSLT
Big data	tGSConnection	tGSCopy	tGSDelete

meaning of the component. Each component has an associated property window for specifying its property values after it is placed on the Canvas.

To clarify job design features and components, an example job design is presented. A Talend job design involves a number of components connected to process work. In Figure 14.21, the Excel data source (*SSExcelSource*) contains rows to be loaded into the *SSSales* table, the fact table of the Store Sales data warehouse. The first component in the flow is the Excel file, a tFileInputExcel component. The file input is processed by the tSchemaComplianceCheck component, a data quality component. Null value and data type checks are performed in the tSchemaComplianceCheck component. The output of the tSchemaComplianceCheck component is further processed by the tMap component. The tMap component performs joins on four dimension tables (*SSCustomer*, *SSStore*, *SSTimeDim*, *and SSItem*) to ensure valid foreign keys. The tMap component uses four tOracleInput components to perform the joins. The output of the tMap processing is loaded into the *SSSales* table, a tOracleOutput component. A commercial job flow would also have outputs for rejected records. The job in Figure 14.21 only has outputs for accepted records. The tSchemaComplianceCheck and tMap components would both be connected to an additional output for rejected records.

Component details are specified in a non-procedural manner using property windows and graphical displays. For example, the tMap component uses a graphical

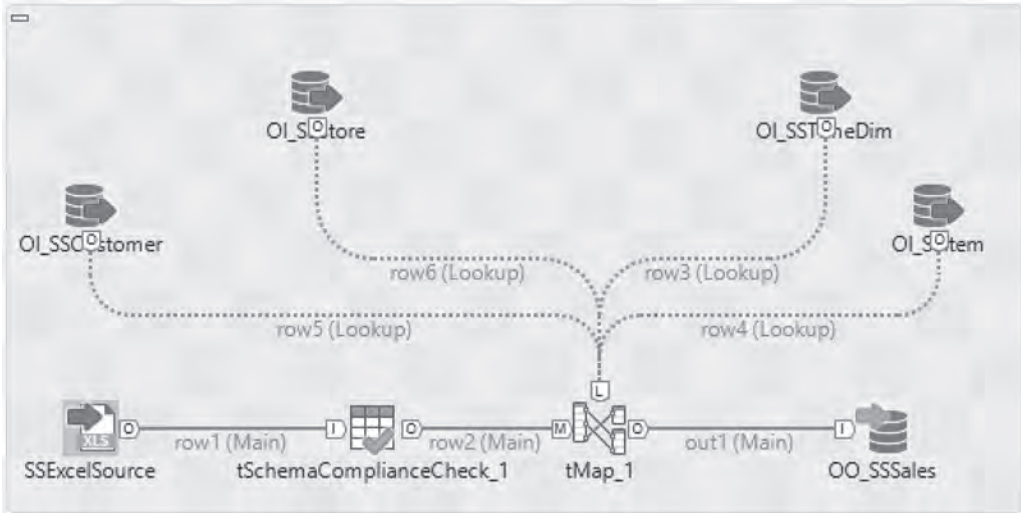


FIGURE 14.21
Example Job Design using Talend Open Studio

display to support join specification. In Figure 14.22, the fields in the Excel data source are mapped to columns in the *SSTimeDim* Oracle table. The columns in the top part (row1) of Figure 14.22 are from the input file. The columns in the bottom part (row2) of the window are the *SSTimeDim* table. A drag and drop method is used to match the columns in the data source and table.

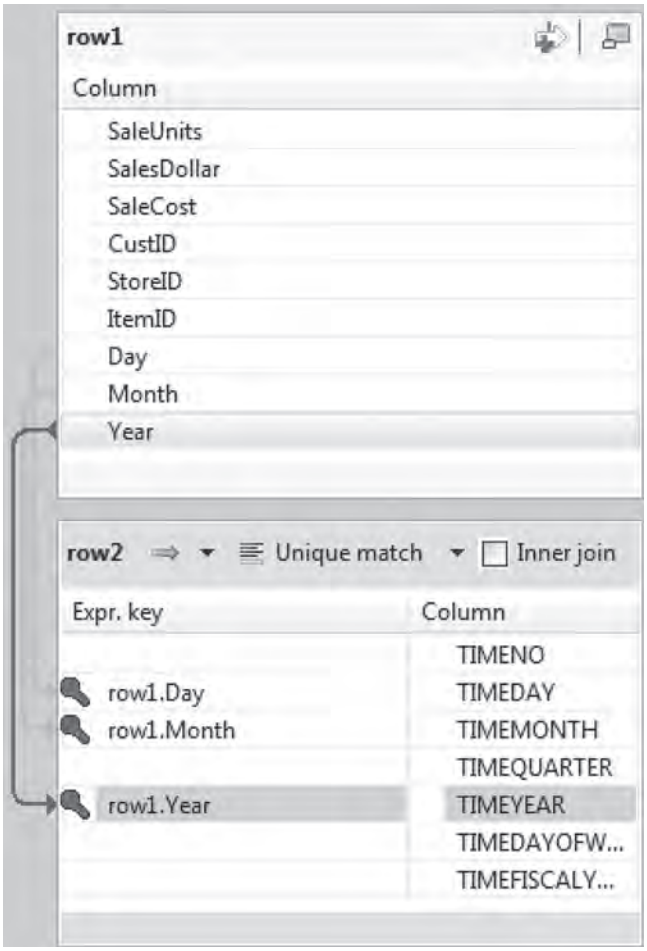


FIGURE 14.22
Talend Join Specification for Time Columns in the tMap Component

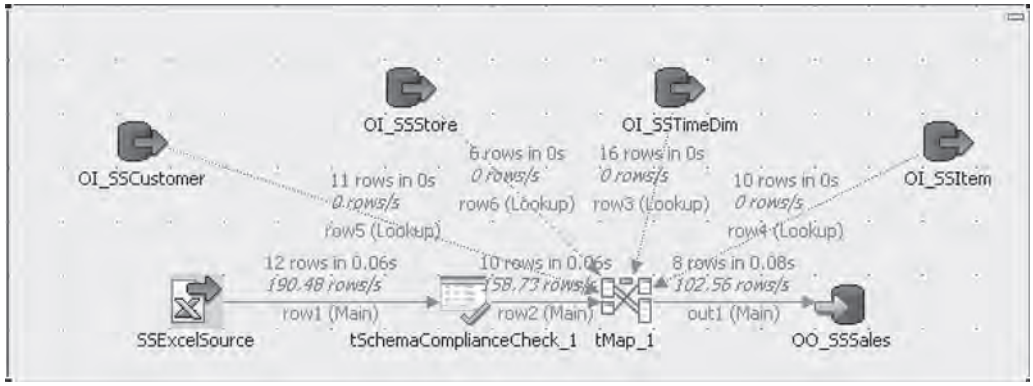


FIGURE 14.23
Example Job Execution using Talend Open Studio

Talend Open Studio provides graphical display of job executions as depicted in Figure 14.23. The screen snapshot was taken from the job design pane after executing the job. The job execution display shows the number of rows processed along with processing times. Figure 14.23 shows that the Excel input file contained 12 rows. The tSchemaComplianceCheck component rejected two rows for null value or data type violations, passing 10 rows to the tMap component. The tMap component rejected two rows for foreign key violations, passing 8 rows to the tOracleOutput component for loading into the *SSSales* fact table.

14.3.3 Pentaho Data Integration

Pentaho provides a unified platform for data integration, business analytics, and big data. Like Talend, Pentaho provides an open source community edition and proprietary extensions in commercial editions. Pentaho offers commercial products for data integration, business analytics, and big data analytics. The community edition of Pentaho Data Integration (known as Kettle) contains three components: Spoon for graphical design of transformations and jobs, Pan for execution of transformations, and Kitchen for job execution.

A Pentaho transformation supports data flow among steps and hops to connect steps. A transformation involves steps, hops, database connections, and distributed processing resources. Pentaho provides a library of steps as shown in Figure 14.24. A job is a higher level data flow among transformations and external entities. Hops provide directed connections among steps. Steps can have multiple input and output connections specified in hops. Pentaho also supports specification of database connections and distributed processing resources such as partitions and clusters.

The Spoon integrated development environment (Figure 14.25) supports viewing components in transformations, designing transformations, and executing transformations. The IDE is somewhat simpler than Talend. The View tab shows steps, hops, and other components in the transformation displayed in the Canvas. The Design tab contains folders of step types. An analyst drags a step from an open folder in the Design tab and places it in the Canvas. Execution controls appear in the tool bar above the Canvas.

Figure 14.26 depicts a simple transformation to filter a Microsoft Excel file. The graphical display of the transformation contains two steps, an input step for a Microsoft Excel file and a filter rows step. The hop indicates a data flow from Excel file step to the Filter rows step. Spoon supports a specification window to provide property values for a step. The specification window for the Excel file indicates the file location, worksheet, fields in the worksheet, and other details. The specification window for the filter rows step indicates the conditions in the bottom part and next steps executed for passing and failing the specified conditions.

This transformation in Figure 14.27 extends the transformation in Figure 14.26 with more steps and hops. Figure 14.27 shows a transformation that merges the Excel

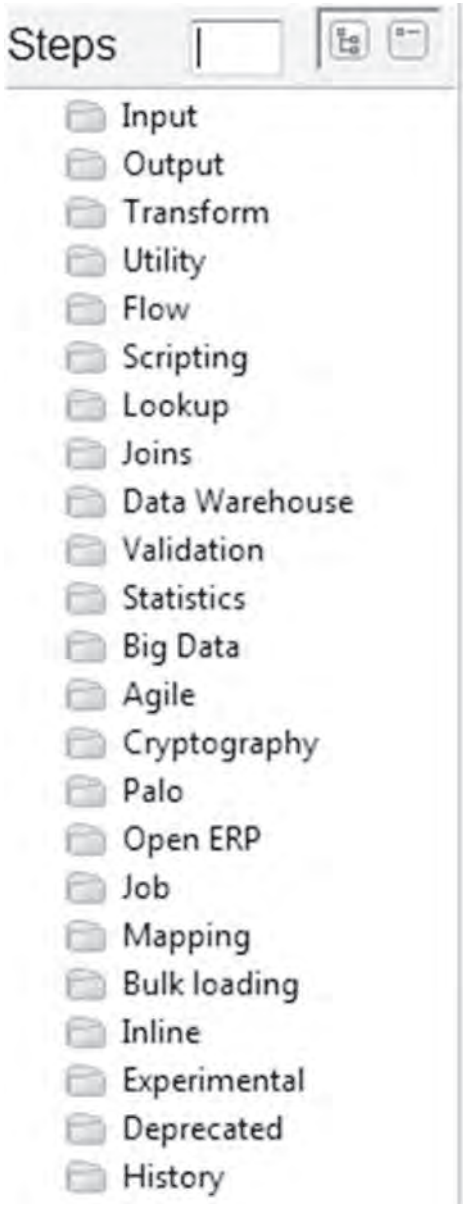


FIGURE 14.24
Pentaho Folders containing a Library of Steps

file input with the *SSTimeDim* table. The sorting row steps are necessary because the merge join step requires data sources sorted on the same criteria. The specification window for the Merge Join step indicates two input steps, join type, and key fields for merging. The merge step uses three date fields (*Day*, *Month*, and *Year*) from the Excel file and three date columns (*TimeDay*, *TimeMonth*, and *TimeYear*) from the *TimeDim* table.

This merge join in Figure 14.27 indicates the tedious nature of some transformations in the ETL architecture. Database compilers determine details about join algorithms and join order for SQL SELECT statements. In the Pentaho ETL architecture, transformations indicate join algorithms and join order, details handled by database compilers in the ELT approach. Figure 14.28 shows the complete Pentaho transformation for combining four tables using merge join steps. The transformation in Figure 14.28 performs the same task as the Talend job specification in Figure 14.21. The Pentaho transformation specifies join algorithms and join order, optimization decisions performed by optimizing database compilers. Unlike vendors of data integration

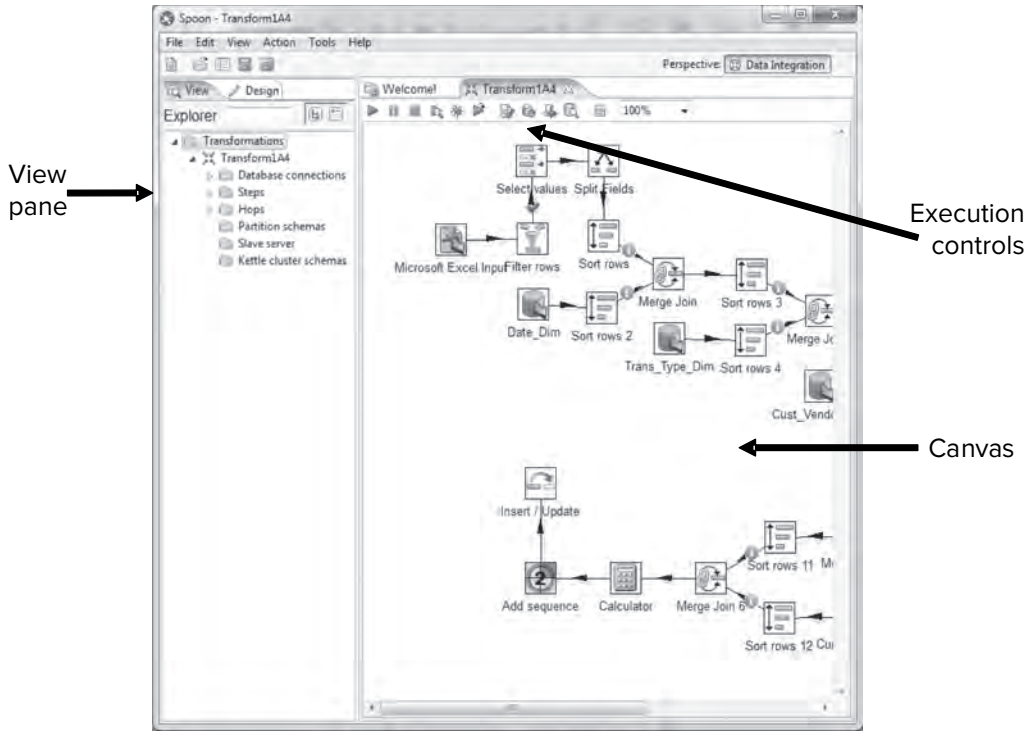


FIGURE 14.25
Integrated Development Environment for Pentaho Spoon

FIGURE 14.26
Example Transformation for Filtering an Excel File

products, DBMS vendors have decades of research and development in technology for optimizing database compilers.

14.3.4 Oracle Data Integrator

Oracle developed the Oracle Data Integrator (ODI) from earlier products and external acquisitions. The ODI has evolved from the acquisition of Sunopsis in 2006 and the Oracle Warehouse Builder, first released in 2000. Oracle has merged both products into the ODI with no new releases for the Oracle Warehouse Builder after 11gR2. The current version of the ODI supports development of platform independent data

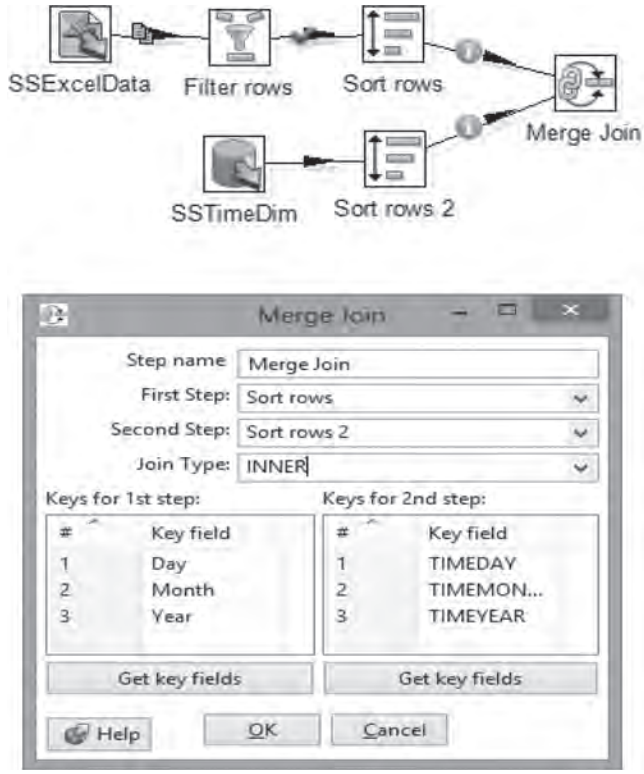


FIGURE 14.27
Example Transformation with a Merge Join Step

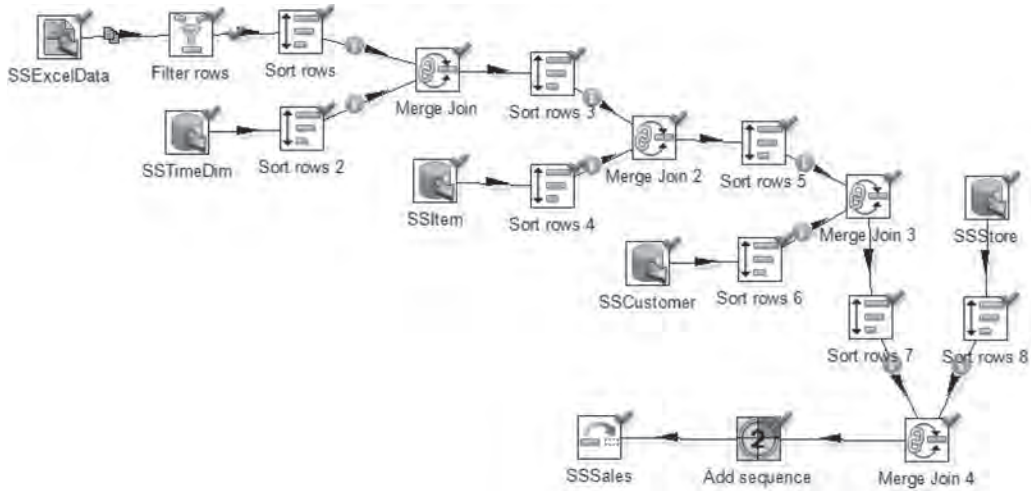


FIGURE 14.28
Pentaho Transformation for Joining 4 Tables

integration solutions although dependent on an Oracle database server in the Extraction, Loading, and Transformation (ELT) architecture.

The ODI contains four major components as depicted in Figure 14.29, adapted from the ODI 12 documentation[2]. On the desktop, the ODI Studio provides navigators for designing data integrity rules and transformations, executing production scenarios, defining the topology of a data integration architecture, and specifying security details such as users, roles, and profiles. On the server side, the ODI requires the WebLogic application server to provide components shown in Figure 14.29. An Oracle database server manages repositories defining information technology infrastructure, project scenarios, execution logs, and model details. The repositories support an open

[2] Oracle, *Oracle Fusion Middleware Getting Started with the Oracle Data Integrator 12c*, October 2015.

relational data model with user-defined elements. The ODI supports extraction from a wide range of data sources including legacy data, files, XML, DBMSs, third-party enterprise applications, and data warehouses.

A data warehouse administer or analyst uses the ODI studio to implement data integration projects using data sources and a target data warehouse. Figure 14.30, adapted from the ODI 12 documentation, depicts a small data integration project for a sales administration data warehouse. The project uses a database of orders and external files of sales people and age groups. A data warehouse administrator or analyst defines declarative rules for transformations and integrity control, both stored in the repository. The ODI Studio generates integration scenarios to implement transformations and integrity rules. An integration scenario implements transformations with mappings, packages, procedures, and variables. The ODI Operator Navigator supports sessions, execution of integration scenarios using data sources to refresh the Sales Administration data warehouse.

Mapping specifications support a declarative or rules driven approach to separate specification from implementation. Figure 14.31, from the ODI 12 documentation, depicts a mapping specification to transform and load a data source of customers into the customer dimension table. The mapping defines three data sources (age group table, customer source table, and sales person file), operations for lookup and join,

FIGURE 14.29
Components in the Oracle Data Integrator

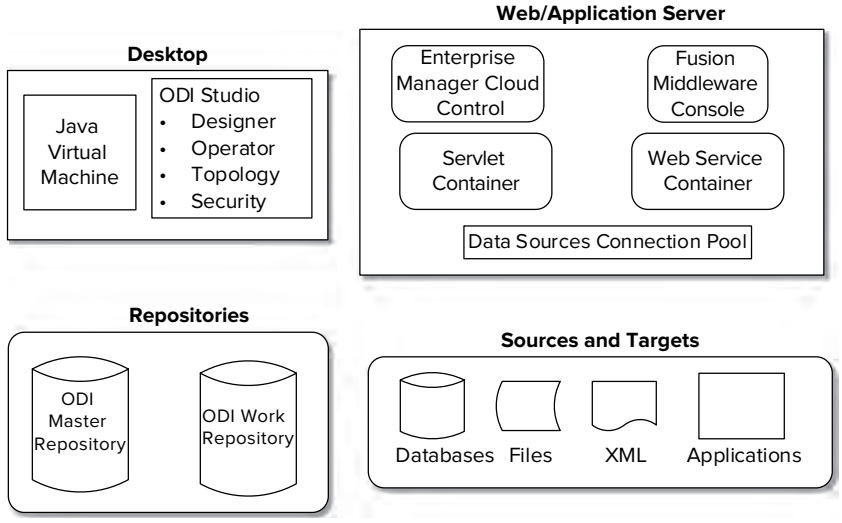
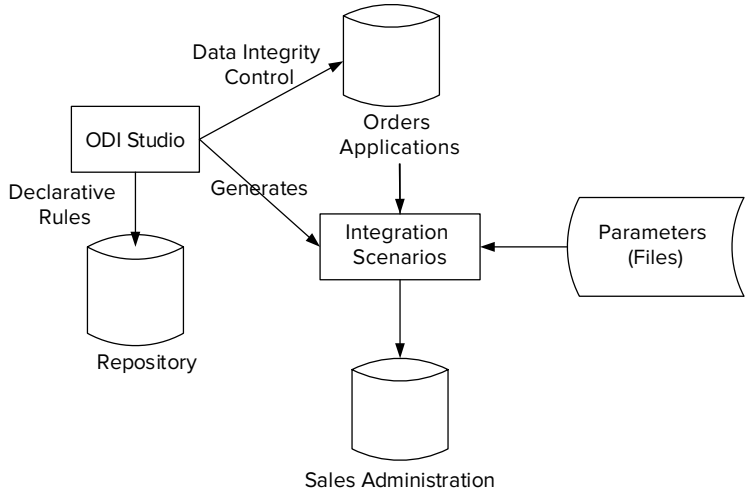


FIGURE 14.30
Simple ODI Project Example



and the target data source (customer dimension table). The lookup operation ensures that customer age is within minimum and maximum ages as defined in the age group table. The join operation combines the sales person file with the customer source table. Target expressions in a mapping transform columns in a data source into columns of the target table. In this example, target expressions map column names, concatenate first and last names, convert marital status to an abbreviation, and convert age in years to an age group.

Before acquiring and redeveloping tools for enterprise data integration, Oracle offered many independent tools appropriate for selected data integration tasks. Oracle has incorporated some of the features of the independent tools into the Oracle Data Integrator. The following list summarizes some independent integration tools available in Oracle.

- SQL*Loader is a long standing Oracle utility for loading data from text files into Oracle tables. SQL*Loader provides data type conversion, conditional loading of data, simple NULL value handling, transformation of data before loading using SQL functions, and direct-path loading for improved performance.
- Oracle Change Data Capture supports synchronous (via triggers) or asynchronous capture (via log files) of change data. The Change Data Capture feature uses a Publish and Subscribe model to control change data availability and notify subscribers about change data availability. Oracle Change Data Capture has been incorporated into the Oracle Data Integrator.
- The Oracle Data Pump enables very high-speed movement of data and metadata from one database to another using import and export utilities.

14.3.5 Oracle SQL Statements for Data Integration

Oracle provides two SQL statements for specific data integration tasks. The MERGE statement supports conditional processing of change data for dimension tables using a single SQL statement. Oracle's implementation of the MERGE statement follows

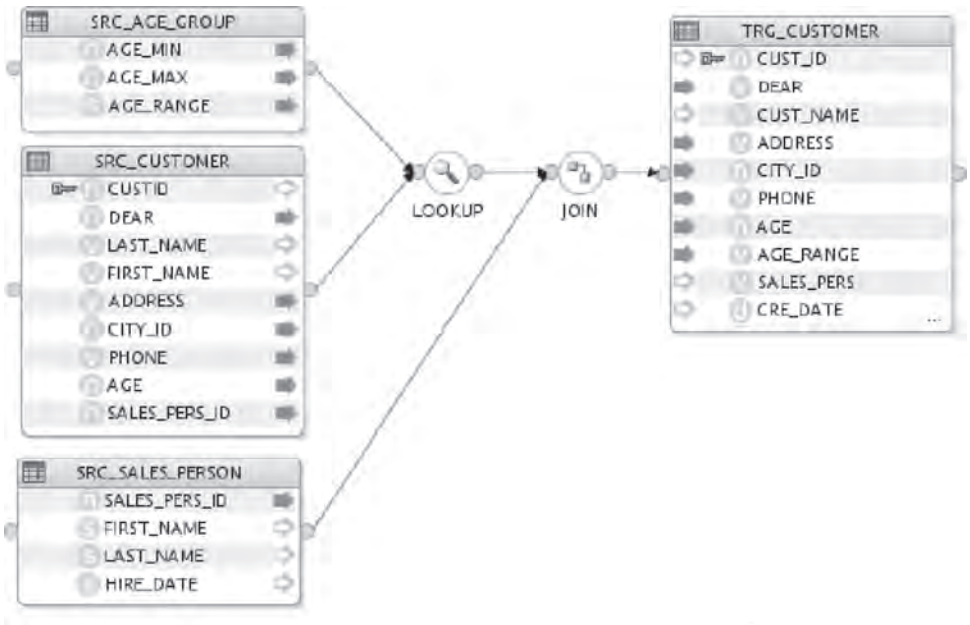


FIGURE 14.31
ODI Mapping Specification[3]

[3] Figure 14.31 Copyright © 2015 by Oracle. *Oracle Fusion Middleware Getting Started with the Oracle Data Integrator 12c*, October 2015.

the standard specification, originally proposed in the SQL:2003 standard. The proprietary Oracle INSERT statement supports addition of rows into multiple target tables with optional conditional specification. This section provides details about both statements.

MERGE Statement The MERGE statement provides conditional update or insert of change data. If a row in a change table matches a row in a target table, the matching target row is updated. Otherwise, the change row is inserted into the target table. The MERGE statement provides improved productivity and performance as compared to multiple statements (INSERT and UPDATE) to accomplish the same task.

The MERGE statement uses a source table, target table, and join condition to support conditional updating and inserting. The diagram in Figure 14.32 indicates that the MERGE statement matches rows in a source table to a target table to generate a new target table. The legend in Figure 14.32 indicates the common heading row in the source and target tables, new rows in the source table, matching rows between the source and target table, and non-matching rows in the target table. In the new target table, the MERGE statement inserts new rows in the source table, updates matching rows, and retains non-matching rows with changes.

To specify merge processing, the MERGE statement uses the MERGE INTO keywords followed by the target table name. The source table name follows the USING keyword. The join condition follows the ON keyword. The WHEN clauses provide action for a matching target row (typically an UPDATE statement) and a non-matching row (typically an INSERT statement).

The MERGE statement in Example 14.1 uses *SSCustomer* as the target table with the alias *Target* and *SSCustomerChanges* as the source table with the alias *Source*. Note that alias names are optional in the MERGE statement. The join condition compares *CustId* in the source and target tables. The MATCHED keyword precedes an UPDATE statement setting each column in the target table to the value specified in the corresponding source column. The NOT MATCHED keywords precede an INSERT statement using the columns of the target table (*SSCustomer*) and the values from the source table (*SSCustomerChanges*).

Sometimes matching change rows only contain values for modified columns. For non-modified columns, a change row contains null values. Partial content for matching change rows reduces the need to specify values for all columns in a row. For new rows, a change table still contains values for all required columns, however.

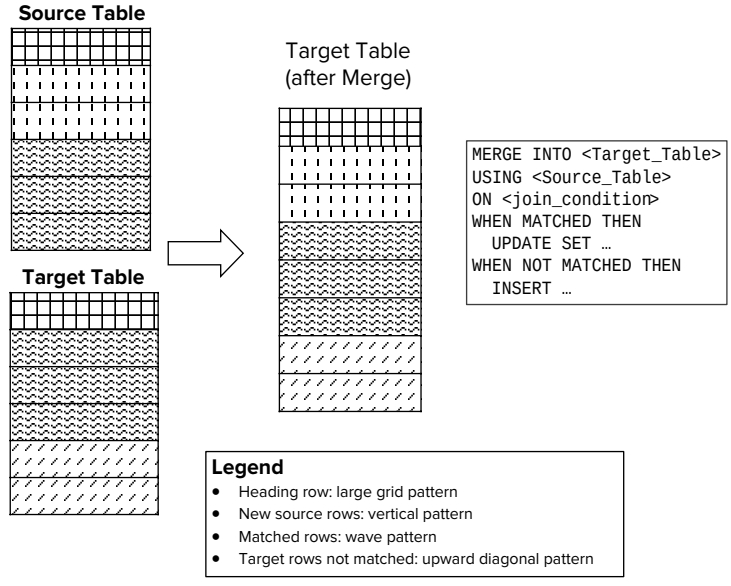


FIGURE 14.32
Overview of the SQL MERGE Statement

Example 14.1

MERGE statement updating matched rows and inserting non-matched rows. Appendix 14.A contains statements to create both tables

```
MERGE INTO SSCustomer Target
USING SSCustomerChanges Source
ON (Target.CustId = Source.CustId)
WHEN MATCHED THEN
 UPDATE SET
   Target.CustName = Source.CustName,
   Target.CustPhone = Source.CustPhone,
   Target.CustStreet = Source.CustStreet,
   Target.CustCity = Source.CustCity,
   Target.CustState = Source.CustState,
   Target.CustZip = Source.CustZip,
   Target.CustNation = Source.CustNation
WHEN NOT MATCHED THEN
 INSERT (Target.CustId, Target.CustName, Target.CustPhone,
         Target.CustStreet, Target.CustCity, Target.CustState,
         Target.CustZip, Target.CustNation)
 VALUES (Source.CustId, Source.CustName, Source.CustPhone,
         Source.CustStreet, Source.CustCity, Source.CustState,
         Source.CustZip, Source.CustNation);
```

Example 14.2 revises Example 14.1 with partial content for matching rows in the change table. For matching rows, the WHEN condition determines if a column value is null. If a change column value is null, the existing value in the target row is used. Otherwise, the new value in the change row is used. Oracle SQL provides the DECODE function for this type of comparison. DECODE (Source.Col1, NULL, Target.Col1, Source.Col1) generates the Target.Col1 value if Source.Col1 is null. Otherwise DECODE generates the Source.Col1 value.

Example 14.2

MERGE statement using the DECODE function to determine columns with null values. Appendix 14.A contains statements to create both tables

```
-- INSERT statements with null values for non-modified columns
-- CustStreet and CustZip modified and null values for other columns
INSERT INTO SSCustomerChanges2
(CustId, CustName, CustPhone, CustStreet, CustCity, CustState, CustZip)
VALUES
('C0954327',NULL,NULL,'444 Jump Ave.',NULL ,NULL,'80128-5443',NULL);

-- MERGE statement
MERGE INTO SSCustomer Target
USING SSCustomerChanges2 Source
ON (Target.CustId = Source.CustId)
```

```
WHEN MATCHED THEN
 UPDATE SET
   Target.CustName = DECODE(Source.CustName, NULL,
                            Target.CustName, Source.CustName),
   Target.CustPhone = DECODE(Source.CustPhone, NULL,
                             Target.CustPhone, Source.CustPhone),
   Target.CustStreet = DECODE(Source.CustStreet, NULL,
                              Target.CustStreet, Source.CustStreet),
   Target.CustCity = DECODE(Source.CustCity, NULL,
                            Target.CustCity, Source.CustCity),
   Target.CustState = DECODE(Source.CustState, NULL,
                             Target.CustState, Source.CustState),
   Target.CustZip = DECODE(Source.CustZip, NULL,
                           Target.CustZip, Source.CustZip),
   Target.CustNation = DECODE(Source.CustNation, NULL,
                              Target.CustNation, Source.CustNation)
WHEN NOT MATCHED THEN
 INSERT (Target.CustId, Target.CustName, Target.CustPhone,
         Target.CustStreet, Target.CustCity, Target.CustState,
         Target.CustZip, Target.CustNation)
 VALUES (Source.CustId, Source.CustName, Source.CustPhone,
         Source.CustStreet, Source.CustCity, Source.CustState,
         Source.CustZip, Source.CustNation);
```

Multiple Table INSERT Statement Oracle provides the proprietary, multiple table INSERT statement for data integration tasks to insert rows from a change table to a set of target fact tables. Typically, the target tables are partitioned by columns (vertical partitioning) or rows (horizontal partitioning). Insertion is usually unconditional for target tables partitioned by columns. Insertion is conditional for target tables partitioned by rows. The multiple table INSERT statement provides improved productivity and performance as compared to multiple INSERT statements to accomplish the same task. The proprietary multiple table INSERT statement has been part of Oracle SQL since the 1990s.

The multiple table INSERT statement uses a source table, multiple target tables, and optional conditions. The diagram in Figure 14.33 indicates a multiple table INSERT statement for two target tables with row partitioning. The legend in Figure 14.33 indicates the common heading row in the source and target tables, target 1 rows in the

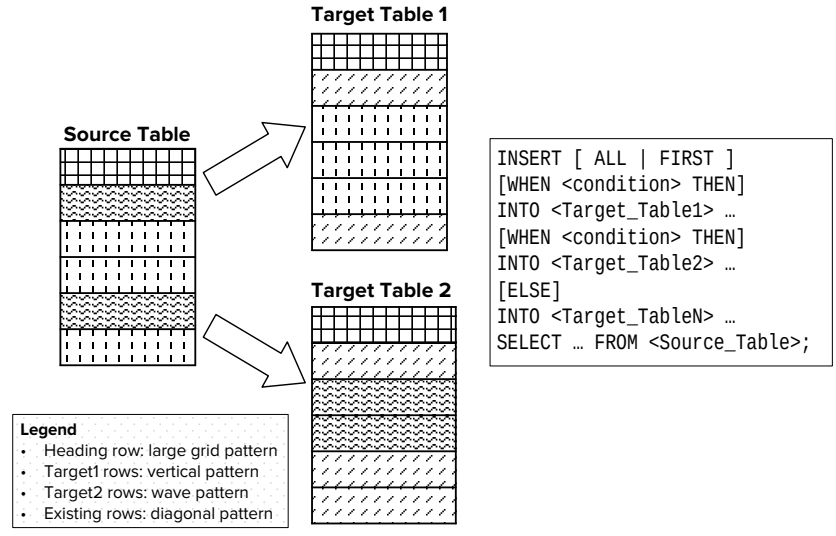


FIGURE 14.33
Overview of the SQL INSERT Statement for Multiple Tables

source and target 1 tables, target 2 rows in the source and target 2 table, and existing rows in the target tables. In the target tables, the INSERT statement adds new rows in each target table using row conditions applied to source rows. If target tables are partitioned by columns instead of rows, the columns in the target tables will overlap usually on the primary key, not on other columns.

To specify processing of insertions into multiple tables, Oracle provides two statement variations. The ALL keyword supports insertion into tables partitioned by columns. The ALL keyword indicates that rows of the source table are inserted into all target tables. Typically the ALL keyword is used unconditionally without WHEN conditions. The FIRST keyword supports target tables partitioned by rows. WHEN conditions are used with the FIRST keyword to indicate the target table in which to insert. A row is inserted into the target table with the first matching WHEN condition. The SELECT block at the end of the statement retrieves data from the source table.

Some examples clarify the unconditional and conditional INSERT statements for multiple tables. Figure 14.34 depicts insertions of rows from the *ProductSale* table into four target tables partitioned by columns. The source table shares three columns (*ProductId*, *ProductName*, and *ProductCategory*) with each target table. Four columns (*Qtr1, Qtr2, Qtr3,* and *Qtr4*) are specific to each target. The target tables contain different quarter sales in each table.

The INSERT ALL statement in Example 14.3 uses one INTO clause for each target table. For example, the INTO clause for the *Qtr1Sale* table uses the *Qtr1* column. The SELECT block at the end of the statement indicates retrieval of all rows from the *ProductSale* table.

A second example clarifies the conditional INSERT FIRST statement. Figure 14.35 depicts an example inserting rows from the *ProductSale* table into three target tables partitioned by rows. The first three columns of the source table (*ProductSale*) match the first three columns of the target tables (*ElectronicsSale, BooksSale,* and *MoviesSale*). The target tables are partitioned with different values for the *ProductCategory* column in each table. The last column (*TotalSales*) of the target tables is computed as the sum of the last four columns (*Qtr1, Qtr2, Qtr3,* and *Qtr4*) of the *ProductSale* table.

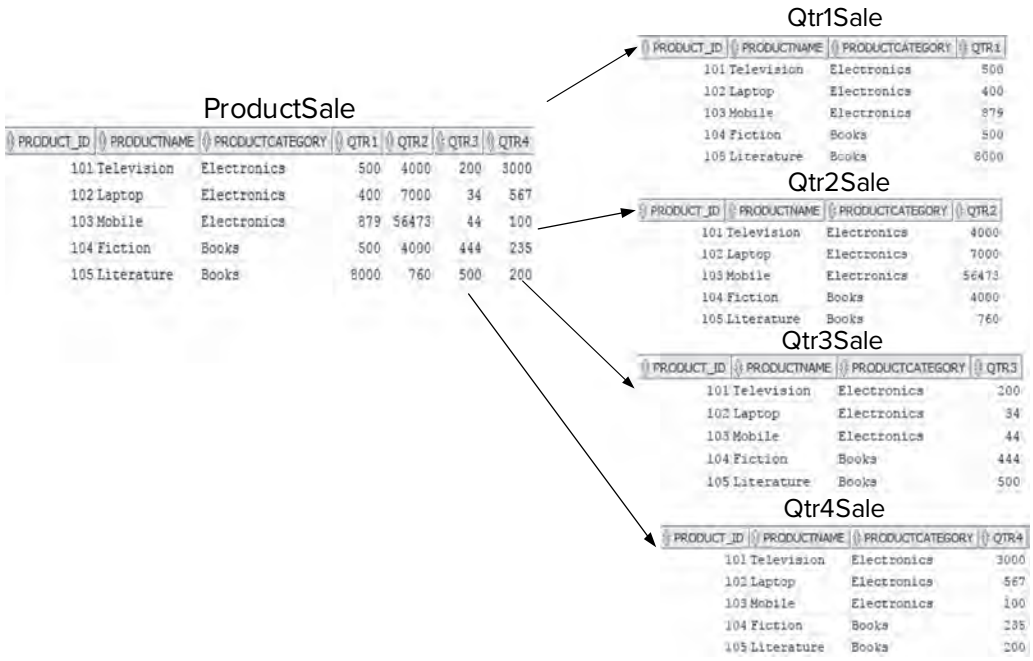


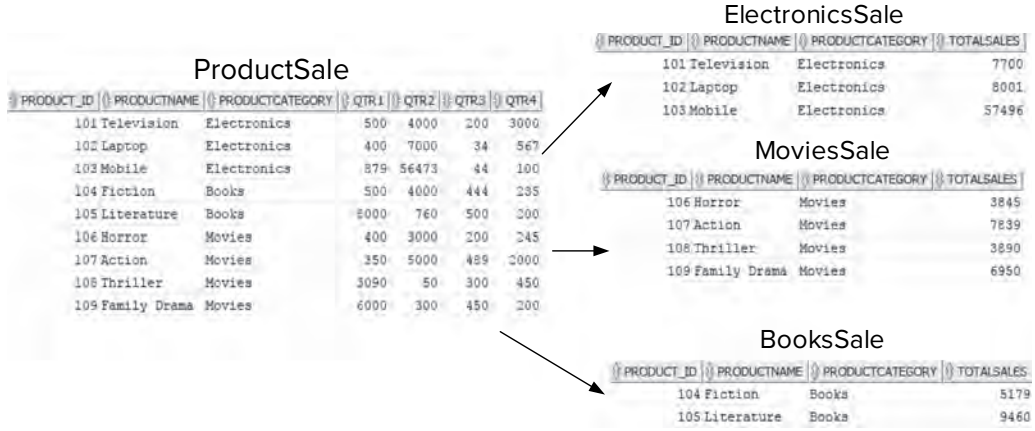
FIGURE 14.34
Example of Unconditional SQL INSERT ALL Statement

Example 14.3

Unconditional multiple table insert using INSERT ALL keywords. Appendix 14.A contains statements to create all tables

```
INSERT ALL
INTO Qtr1Sale VALUES (Product_ID,ProductName,ProductCategory,Qtr1 )
INTO Qtr2Sale VALUES (Product_ID,ProductName,ProductCategory,Qtr2 )
INTO Qtr3Sale VALUES (Product_ID,ProductName,ProductCategory,Qtr3 )
INTO Qtr4Sale VALUES (Product_ID,ProductName,ProductCategory,Qtr4 )
SELECT * FROM ProductSale;
```

FIGURE 14.35
Example of Conditional SQL INSERT FIRST Statement



The INSERT FIRST statement in Example 14.4 uses one WHEN clause and one INTO clause for each target table. For example, the INTO clause for the *ElectronicsSale* table uses "Electronics" as the value in the WHEN condition. The SELECT block at the end of the statement indicates retrieval of all rows from the *ProductSale* table.

Example 14.4

Conditional multiple table insert using INSERT FIRST keywords. Appendix 14.A contains statements to create all tables

```
INSERT FIRST
 WHEN (ProductCategory = 'Electronics') THEN
 INTO ElectronicsSale VALUES
  (Product_ID,ProductName,ProductCategory, (Qtr1+Qtr2+Qtr3+Qtr4) )
WHEN (ProductCategory = 'Movies') THEN
 INTO MoviesSale VALUES
  (Product_ID,ProductName,ProductCategory, (Qtr1+Qtr2+Qtr3+Qtr4) )
WHEN (ProductCategory = 'Books') THEN
 INTO BooksSale VALUES
  (Product_ID,ProductName,ProductCategory, (Qtr1+Qtr2+Qtr3+Qtr4) )
SELECT * FROM ProductSale;
```

You should only use INSERT ALL when a source table row can be inserted into multiple target tables although INSERT ALL and INSERT FIRST often produce identical results. For Example 14.4, INSERT ALL and INSERT FIRST produce identical results. Mutually exclusive conditions in WHEN clauses generate identical results with both ALL and FIRST options. Non-exclusive conditions such as > 400 and > 700 generate additional rows for INSERT ALL because a row matching > 700 also matches > 400. INSERT FIRST uses the first matching WHEN clause only, inserting each source row into at most one target table.

CLOSING THOUGHTS

This chapter extended the conceptual background and skills foundation provided in Chapters 12 and 13 with detailed coverage of data integration. Maintaining a data warehouse is a difficult process that must be carefully managed. The first part of Chapter 14 presented the kinds of data sources used in maintaining a data warehouse, a generic workflow describing the tasks involved in maintaining and populating a data warehouse, and conceptual background about managing the refresh process. This conceptual material extends management concepts covered in Chapter 12.2.

The data integration techniques in Chapter 14 extend the skills and concepts of the schema integration process covered in Chapter 13.4. The second part of Chapter 14 covered parsing with regular expressions, correcting and standardizing values, and entity matching. Writing regular expressions is a detailed skill important in extracting components embedded in text fields. Correcting and standardizing values involves resolution of missing, conflicting, and non-standardized data from data sources. Entity matching identifies and resolves duplicate records when no common, reliable identifier exists, an important integration task especially for customer-oriented data warehouses.

This chapter advocated usage of data integration tools with integrated development environments to improve productivity for development of data integration procedures as well as improve operational performance of refresh processing. The third part of this chapter provided broad coverage of features and architectures of data integration tools and details about prominent data integration tools. You are encouraged to gain experience with a commercial data integration tool, extending the details of product coverage (Talend Open Studio, Pentaho Data Integration, or Oracle Data Integrator) in this chapter. You should also practice using SQL statements for data integration, the SQL standard MERGE statement and the proprietary Oracle INSERT statement for multiple tables.

REVIEW CONCEPTS

- Primary goal of data integration: provide a single source of truth
- Data integration challenges: large volumes of data, legacy source systems, lack of standards for formats, units of measure, and integrity rules, varying update frequencies, missing data, and lack of common identifiers
- Classifying change data by processing level and source system requirements
- Kinds of change data used to populate a data warehouse: cooperative, logged, queryable, and snapshot
- Cooperative change data involving triggers in source systems to capture change data at transaction time
- Logged change data involving logs of activity and substantial processing to decompose text data in a log

- Queryable change data involving time stamps in a data source and periodically executed queries on a source system
- Snapshot change data involving periodic dumps of source data and difference processing to identify new records, changed records, and deleted records
- Phases in the workflow for maintaining a data warehouse: preparation, integration, and propagation
- Data quality problems encountered in data warehouse loading and maintenance: multiple identifiers, multiple names, different units, missing values, orphaned transactions, non-standard text data, conflicting data, and different update times
- Initial population of a data warehouse to discover and resolve data quality problems
- Control over valid time lag and load time lag in refreshing a data warehouse
- Factors influencing refresh frequency and schedules: timeliness importance, refresh costs, and constraints
- Determining the refresh frequency for a data warehouse: balancing frequency of refresh against refresh costs while satisfying refresh constraints
- Types of refresh constraints: source access, integration, completeness/consistency, availability
- Regular expressions for decomposing multipurpose text fields into constituent parts
- Regular expressions containing literals (characters to match exactly), metacharacters (characters with special meaning), and escape sequences (using a metacharacter as a literal)
- Metacharacters for specifying patterns in regular expressions involving iteration, range of characters, position, and alteration
- Pattern matching components: search expression, target string, and match result
- Sentiment analysis for context sensitive parsing
- Importance of understanding the reason for missing values in resolving problems
- Entity matching to identify duplicate records when no common, reliable identifier exists
- Entity matching outcomes when comparing records
- Usage of quasi identifiers in entity matching
- Distance measures for text comparisons as fundamental tools for entity matching
- Importance of data integration tools to improve productivity for developing data integration procedures and workflows
- Essential features of data integration tools: integrated development environments, graphical specification of workflows, non procedural specification of transformations, repository, and data source connectivity
- Secondary features of data integration tools: data profiling, change data capture, and job management
- ETL architecture emphasizing the usage of an ETL engine independent of a DBMS engine for performing complex data transformations before loading into data warehouse tables
- ELT architecture emphasizing the usage of a relational DBMS to perform complex data transformations after loading data into data warehouse tables
- Talend Open Studio: integrated development environment and non procedural specification of jobs and components

- Pentaho Data Integration: integrated development environment and non procedural specification of transformations, hops, and steps
- Oracle Data Integrator: prominent data integration tool using the Extraction, Loading, and Transformation architecture
- SQL standard MERGE statement combining insertion and updating of change data for dimensions
- Proprietary Oracle INSERT statement supporting inserts from a change table to multiple fact tables

QUESTIONS

1. What is cooperative change data?
2. What is logged change data?
3. What is queryable change data?
4. What is snapshot change data?
5. Briefly explain the classification of change data.
6. Briefly describe the phases of data warehouse maintenance.
7. Briefly define common data quality problems that should be resolved by the preparation and integration phases.
8. What is valid time lag?
9. What is load time lag?
10. What is the primary objective in managing the refresh process for a data warehouse?
11. How does importance of timeliness affect the objective of refreshing a data warehouse?
12. What are components of refresh costs?
13. What types of constraints affect the refresh process?
14. Briefly explain the initial loading process for a data warehouse.
15. What are the elements of a regular expression?
16. To perform pattern matching, what elements does a user provide?
17. How are regular expressions useful for resolving data quality problems?
18. Briefly explain metacharacters for iteration.
19. Briefly explain metacharacters for specification of a matching position.
20. When should you use an escape sequence in a regular expression?
21. Why are groups useful in regular expressions?
22. Briefly explain principles for handling missing values that are inapplicable or involved with summarizability problems.
23. Briefly explain principles for handling missing values that are unknown.
24. Briefly explain principles for standardizing values.
25. What is entity matching?
26. Briefly explain the outcomes of entity matching.
27. Which outcome is typically most important to avoid in entity matching?
28. What is a quasi identifier and how is it used in the entity matching process?
29. Why are distance measures for text comparisons important in entity matching?
30. What are common distance measures of text similarity used in entity matching?
31. What benefits are provided by data integration tools?

32. Briefly explain the ETL architecture for data integration.
33. Briefly explain the ELT architecture for data integration.
34. Which data integration architecture (ETL or ELT) will likely dominate in the future?
35. Briefly explain essential features of data integration tools.
36. How does the Gartner market summary classify firms?
37. What is job management as a feature of a data integration tool?
38. What is data profiling as a feature of a data integration tool?
39. What is change data capture as a feature of a data integration tool?
40. What features are provided by Talend Open Studio for Data Integration?
41. Briefly describe three components provided in Talend Open Studio.
42. What components are provided in the community edition of Pentaho Data Integration?
43. What is a transformation in a job for Pentaho Data Integration?
44. What is a hop in a transformation in Pentaho Data Integration?
45. How did the Oracle Data Integrator evolve?
46. What is a mapping specification in the Oracle Data Integrator?
47. What architectures are supported by Talend Open Studio, Pentaho Data Integration, and Oracle Data Integrator?
48. What is the purpose of the MERGE statement?
49. What elements are specified in a MERGE statement?
50. What is the purpose of the multiple table INSERT statement?
51. What elements are specified in an INSERT statement for multiple tables?
52. Briefly describe two statement variations for processing insertions into multiple tables.
53. What is sentiment analysis and how is it used for data warehouses?

PROBLEMS

The problems provide practice with regular expressions and Oracle SQL statements for data integration tasks. To gain experience with regular expressions, you should use a web page containing a regular expression tester (such as regex101.com, regexr.com, and www.regextester.com) for regular expression problems (1 to 3). Problems 4 to 11 involve the MERGE and multiple table INSERT statements available in Oracle. Appendix 14.B contains SQL statements for creating tables used in these problems. The textbook's website contains a complete of statements to create and populate all tables used in the problems.

For practice with data integration tools, the presentation in this chapter lacks sufficient details. You need a detailed tutorial and practice problems. Due to the length of a tutorial (more than 40 pages), the textbook's website contains tutorial material. You can find a detailed tutorial for Pentaho Data Integration on the textbook's website along with supporting materials for tables and files used in the tutorial.

1. Determine the results of pattern matching using the regular expressions and target strings in Table 14-P1. You should determine the results yourself using knowledge of the metacharacters and then use a regular expression testing tool to verify your predicted results.
2. Write a regular expression to match against phone numbers with the format "(ddd) ddd-dddd" where d is a digit and the parentheses are literal characters. The first digit in a phone number should not be a 0. There should not be

Regular Expression	Target Strings	Evaluation (please complete)
“labou?r”	“pro labor”, “labour union”	
“help*”	“hello”, “hellcat”, “herself”	
“help+”	“hepless”, “helpless”, “help me”	
“[aeiou][bcdfgm]”	“a dog”, “ihop”, “uomo”	
“[A-Z]{1}:[0-9]{2,4}”	“a:100”, “big A:2456”	
“^big”	“bill dig”, “one big bill”	
“big$”	“bill dig”, “dig bin”	
“[^a-z]+[^A-Z]”	“123ab”, “aBCd”, “aaBB”	
“^\^”	“^abc”, “abc^”, “abc”	
“\$$”	“abc$”, “$abc”, “abc$d”	

TABLE 14-P1
Regular Expressions and Search Strings to Evaluate

any other characters at the beginning or ending of the string. Use a regular expression testing tool to test your regular expression.

3. Write a regular expression to match against postal codes with the following format: “ddddd-dddd” where *d* is a digit. The hyphen is a literal character. Both the hyphen and last four digits are optional. If there is a hyphen, the last four digits must be present. There should not be any other characters at the beginning or ending of the string. Use a regular expression testing tool to test your regular expression.
4. Write an Oracle SQL MERGE statement to combine the *SSItem* dimension table and the *SSItemChanges1* change table. Appendix 14.AB contains CREATE TABLE statements for both tables. Each matching row of *SSItemChanges1* contains values for both modified and non-modified columns.
5. Write an Oracle SQL MERGE statement to combine the *SSItem* dimension table and the *SSItemChanges2* change table. Appendix 14.B contains CREATE TABLE statements for both tables. Each matching row of *SSItemChanges2* contains new values for modified columns and null values for non-modified columns.
6. Write an Oracle SQL INSERT FIRST statement to insert rows of the *ProductSale1* table into four tables (*ProductSales2014*, *ProductSales2015*, *ProductSales2016*, and *ProductSales2017*). Appendix 14.B contains CREATE TABLE statements for all tables. Note that the target tables lack the *SalesYear* column. If *SalesYear* equals 2014, insert a row into *ProductSales2014*. The comparisons for the other tables only differ on the *SalesYear* value and suffix in the name of the target table. You should compute the *SalesAmt* column in each target table as the sum of the quarter sales (*Qtr1*, *Qtr2*, *Qtr3*, and *Qtr4*) in the corresponding row of the *ProductSale* table.
7. On problem 6, is the number of the rows in the target tables the same when using INSERT FIRST versus INSERT ALL? Justify your answer.
8. Write an Oracle SQL INSERT FIRST statement to insert rows of the *ProductSale2* table into three target tables (*Year_Low_Sales*, *Year_Mid_Sales*, and *Year_High_Sales*). Appendix 14.B contains CREATE TABLE statements for all tables. Insert a row into *Year_Low_Sales* when annual sales (sum of *Qtr1*, *Qtr2*, *Qtr3*, and *Qtr4*) are less than 4,000. Insert a row into *Year_Mid_Sales* when annual sales are greater than or equal 4,000 and less than 7,000. Insert remaining rows into *Year_High_Sales*. You should compute the *SalesAmt* column in each target table as the sum of the quarter sales (*Qtr1*, *Qtr2*, *Qtr3*, and *Qtr4*) in the corresponding row of the *ProductSale* table.
9. On problem 8, is the number of the rows in the target tables the same when using INSERT FIRST versus INSERT ALL? Justify your answer.

10. Write an Oracle MERGE statement to combine the *Mobile_Bill* table with the *Mobile_Usage* table matching on *CustId*. Appendix 14.B contains CREATE TABLE statements for both tables. The *Mobile_Bill* table contains the most recent bill with the current amount (*CurrentAmt*) and past amount (*PastAmt*). When a match occurs, update the *Mobile_Bill.CurrentAmt* column as minutes used (*Mobile_Usage.MinutesUsed*) times 0.05 and the *Mobile_Bill.PastAmt* column as the previous current amount (*Mobile_Bill.CurrentAmt*) plus the previous past amount (*Mobile_Bill.PastAmt*). When a match does not occur, insert a row into the *Mobile_Bill* table with the customer identifier (*Mobile_Usage.CustId*), minutes used (*Mobile_Usage.MinutesUsed*) times 0.05, and 0 for the past amount (*Mobile_Bill.PastAmt*).
11. Write an Oracle INSERT FIRST statement to insert rows from a mobile customer table (*Mobile_Customer*) into three tables (*Mobile_Gold*, *Mobile_Silver*, and *Mobile_Bronze*) based on a customer's current revenue amount (*Mobile_Customer.CurrentAmt*). Appendix 14.B contains CREATE TABLE statements for all tables. If the current revenue amount is greater than or equal to 150, insert the mobile customer row into the *Mobile_Gold* table. If the current revenue amount is greater than or equal to 100, insert the mobile customer row into the *Mobile_Silver* table. Otherwise, insert the mobile customer row into the *Mobile_Bronze* table.
12. On problem 11, is the number of the rows in the target tables the same when using INSERT FIRST versus INSERT ALL? Justify your answer.

REFERENCES FOR FURTHER STUDY

Several references provide additional details about important parts of Chapter 14. You should consult Bouzeghoub et al. (1999) and Fisher and Berndt (2001) about the refresh process. For additional details about regular expressions, you should search under "regular expression tutorials" and "regular expression testers". For more details about Talend Open Studio, you should visit the Talend website (www.talend.com). For more details about Pentaho Data Integration, you should visit the Pentaho website (www.pentaho.com). You can install the free community edition products for both Talend and Pentaho. For details about Oracle data integration features, you should consult the online documentation in the Oracle Technology Network (www.oracle.com/technetwork).

chapter

15

Query Formulation for Data Warehouses

Learning Objectives

This chapter helps you develop skills for query formulation in business intelligence applications, extending skills for conceptual data warehouse design in Chapter 13 and data integration in Chapter 14. After this chapter, the student should have acquired the following knowledge and skills:

- Explain terminology and basic statement syntax of Microsoft Multidimensional Expressions
- Gain practice with a pivot table tool for OLAP queries
- Write and document SQL SELECT statements using the CUBE, ROLLUP, and GROUPING SETS operators
- Write and document SQL SELECT statements using analytic functions for qualitative performance, trend analysis, and quantitative comparisons
- Understand conceptual differences between materialized views for summary data storage and retrieval and traditional relational views for retrieval of derived data
- Gain insights about the complexity of the query rewriting process to match materialized views with user queries
- Explain storage and optimization technologies in relational DBMSs for data warehouses

OVERVIEW

After the foundation of skills and management concepts about data warehouses provided in Chapters 12 to 14, you are ready to complete your data warehouse study with detailed skills for query formulation. Although these query formulation skills build on the foundation in Chapters 4 and 9, query formulation for data warehouses involves substantial new concepts and practices. These skills are especially important for information technology professionals assisting business analysts to use data warehouses for business intelligence applications.

Chapter 15 emphasizes query formulation skills for both data cubes and relational databases. You will first learn concepts and query formulation skills for data cubes, providing a business analyst perspective about query formulation. Chapter 15 covers an important de facto standard for data cubes, Microsoft Multidimensional Expressions (MDX) as well as pivot table tools providing convenient interfaces for MDX retrievals. Next, you will learn about two major extensions of the SQL SELECT statement for a data warehouse stored as a relational database. Extensions to the GROUP BY clause support calculations of subtotals in query results. Extensions for analytic functions support qualitative performance, trend analysis, and quantitative comparisons, important in many business intelligence applications.

Analytic functions extend both the processing model and the syntax of the SQL SELECT statement.

The last part of Chapter 15 emphasizes extensions to relational DBMSs for efficient storage and retrieval of summary data. You will learn about materialized views, query rewriting principles to substitute materialized views in user queries, optimization techniques employed by relational DBMSs, and architectures to support data warehouse processing. Since relational DBMSs provide the underlying storage and retrieval capabilities for enterprise data warehouses, understanding relational DBMS features for data warehouses is essential background for a data warehouse professional.

15.1 ONLINE ANALYTIC PROCESSING (OLAP)

Online Analytic Processing (OLAP) refers to computing solutions for multidimensional analysis of data warehouses by business analysts. In the early days of data warehouses, OLAP solutions provided both primary storage of data warehouses as well as retrieval tools. Since large-scale investment by relational DBMS vendors, OLAP solutions now mostly provide storage of data marts to augment primary storage of data warehouses in relational databases as well as a foundation for business intelligence tools used by business analysts.

This section emphasizes basic aspects of OLAP client technology rather than detailed coverage appropriate in a business intelligence course. The first section provides an overview of Microsoft Multidimensional Expressions (MDX), an important de facto standard for OLAP technology. The second section presents an overview of pivot table tools that provide a convenient interface for data cube manipulation and MDX statements.

15.1.1 Microsoft Multidimensional Expressions (MDX)

Microsoft Multidimensional Expressions (MDX) has emerged as an important OLAP specification used in Microsoft products, other vendor products, and open source projects. MDX provides the foundation for the Microsoft SQL Server Analysis Service and Microsoft Excel pivot tables. A wide range of vendors has adopted MDX for both commercial products and open source projects. Prominent commercial vendors using MDX are Hyperion, IBM, and SAP in addition to Microsoft. Prominent open source projects using MDX are JPivot, Pivot4J, OLAP4J, and the Mondrian OLAP server.

MDX is a mature specification, becoming a de facto standard after careful development. As depicted in Figure 15.1, Microsoft initially developed MDX in the late 1990s with a specification in 1997 and release as part of the Microsoft OLAP Services 7.0 in 1998. After the initial specification in 1997, Microsoft made a major revision to MDX in 2005. With cooperation from Microsoft, a web standards group (XMLA Council) specified mdXML as part of the Extensible Markup Language (XML) for Analysis standard. XML is the underlying language or meta language for other web languages such as HTML.

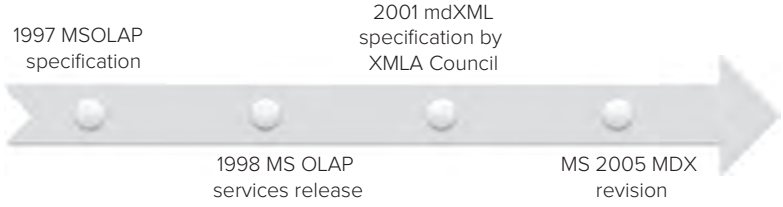


FIGURE 15.1
MDX History

Cube Representation in MDX This subsection presents concepts about MDX data cubes rather than tedious data definition statements. Typically, a data warehouse professional defines an MDX cube with a graphical tool or guided specification rather than using MDX data definition statements directly. For example, Microsoft provides the Cube Wizard to define an MDX cube.

MDX supports cube specification with some extensions to cube concepts presented in Chapter 13.1. An MDX cube consists of dimensions and measures. The cube in Figure 15.2[1] contains two measures (Quantity and Sales) along with five dimensions (Markets, Customers, Product, Time, and Order Status). An MDX dimension contains attributes, components of a dimension. For example, the Product dimension has attributes Line, Vendor, and Product. Attributes can independent or related in a hierarchy. In Figure 15.2, attributes are hierarchically related. For the Product dimension, the attributes have a hierarchy containing Line as the parent, Vendor within Line, and Product within Vendor.

In MDX, members are values of an attribute. Figure 15.3 depicts members of the attribute hierarchy for the Product dimension. For example, "Classic Cars" is a member of the Line attribute, "Autoart Studio Design" is a member of the Vendor attribute, and "1968 Ford Mustang" is a member of the Product attribute.

To comprehend MDX queries, you need insight from a populated cube in MDX. Figure 15.4 depicts a snapshot of the Steel Wheels data cube in the Pivot4J plugin of Pentaho Business Analytics, a client tool that supports MDX. In this snapshot, the rows contain members of the Product dimension, while the columns contain members of the Time dimension. The cells contain sales values for the combination of member values, one from each dimension. For example, 1,514,407 is the sales value for Classic Cars in 2003.

A somewhat subtle part of MDX cubes is the presence of measures on either the rows or columns. Figure 15.4 displays the Sales measure stacked on the columns inside the Time dimension.

An MDX cube can also contain aggregations. Figure 15.4 displays average sales for members of the Product and Time dimension. For example, 282,876 is the average sales in 2003 and 1,363,807 is the average sales of classic cars.

MDX has some cube terminology beyond terms defined in Chapter 13.1. An MDX tuple, a combination of members with one from each member, identifies a cell. For example in Figure 15.4, the combination of members, Classic Cars and 2003, identify a cell. Axis refers to a dimension from a source data cube used in a query. In MDX, cubes typically have two axis, rows and columns, although more axes can be used. Multiple dimensions may be stacked or embedded on the same dimension. A slicer refers to a tuple in the result of an MDX query result.

FIGURE 15.2
Example MDX Cube

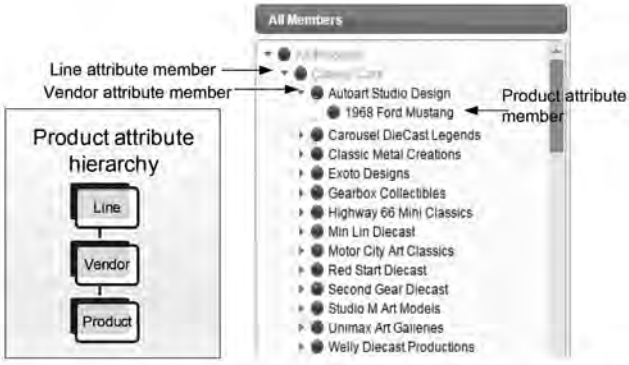


FIGURE 15.3
Attribute Hierarchy and Members in an MDX Cube

[1] Figures 15.2 through 15.4 contain snapshots using the Pivot4J plugin in Pentaho Business Analytics.

FIGURE 15.4
Steel Wheels Cube Display in Pivot4J

	Time				
	All Years	2003	2004	2005	Average
	Measures	Measures	Measures	Measures	Measures
Product	Sales	Sales	Sales	Sales	Sales
Classic Cars	4,091,420	1,514,407	1,838,275	738,738	1,363,807
Motorcycles	1,274,125	397,220	590,580	286,325	424,708
Planes	1,076,757	347,755	528,928	200,074	358,919
Ships	748,671	244,821	375,672	128,178	249,557
Autoart Studio Design	67,592	19,764	36,027	11,801	22,531
Carousel DieCast Legends	208,583	75,184	102,537	30,862	69,528
Min Lin Diecast	79,662	29,691	37,098	12,873	26,554
Red Start Diecast	77,872	25,207	40,948	11,717	25,957
Studio M Art Models	84,190	27,795	39,390	17,005	28,063
Unimax Art Galleries	147,078	42,313	73,966	30,799	49,026
Welly Diecast Productions	83,693	24,867	45,706	13,120	27,898
Trains	234,469	72,802	124,750	36,917	78,156
Trucks and Buses	1,154,281	420,430	531,976	201,875	384,760
Vintage Cars	2,066,226	679,949	997,560	388,718	688,742
Average	818,919	282,876	383,672	152,371	272,973

MDX SELECT Statement Although the MDX SELECT statement appears similar to the SQL SELECT statement, fundamental differences exist. The result of a SQL SELECT statement is a table, while the result of an MDX SELECT statement is a data cube. Tables are two-dimensional objects with rows and columns. Data cubes are n-dimensional possibly with summary calculations. Different mathematical approaches (relational algebra for tables and matrix algebra for data cubes) provide the foundation for query languages.

The difference in underlying objects (tables versus data cubes) makes retrieval languages rather different even though some important keywords remain the same. Table 15-1 compares SELECT statement clauses in SQL and MDX. In the SELECT clause, MDX supports cubes with a maximum of 128 dimensions although most result cubes contain a small number of dimensions. MDX provides aliases for the first five dimensions in a result cube: COLUMNS, ROWS, PAGES, SECTIONS, and CHAPTERS. The FROM clause in MDX uses only a single, source cube. Conditions in the WHERE clause slice the source cube in the FROM clause. Dimensions in the WHERE clause differ from dimensions in the SELECT clause.

Some examples clarify basic elements of the MDX SELECT statement. Figure 15.5 displays a data cube result and associated MDX SELECT statement. The SELECT clause contains members (2003 and 2004) of the Time Dimension on the rows. The columns axis contains two measures, Sales and Quantity. The FROM clause indicates that SteelWheelsSales is the source data cube. The WHERE clause restricts sales and quantity calculations to classic cars. The WHERE clause condition defines a slicer, a result tuple in the data cube result.

TABLE 15-1
Comparison of the SQL SELECT and MDX SELECT Statements

	Language	
Clause	**SQL**	**MDX**
SELECT	List of columns	List of axis dimensions (source cube cells)
FROM	List of tables	Cube name
WHERE	Conditions restricting rows	Restriction to a combination of dimension members (result cube cells)

FIGURE 15.5
MDX Query and Associated Data Cube

The cross join operator combines multiple dimensions or measures on a query axis. You should use the cross join operator only with a small number of elements as it generates all combinations of elements. In Figure 15.6, the cross join operator combines selected members (Shipped and Cancelled) of the Order Status dimension with the Sales and Quantity measures on the Columns axis. As in Figure 15.5, the WHERE clause in Figure 15.6 restricts sales and quantity calculations to classic cars. Since the Shipped and Cancelled members are not exhaustive, the result lacks Sales and Quantity values for other Order Status members.

The examples in Figures 15.7 and 15.8 show the impact of a slicer condition. A dimension in a slicer condition cannot be used in an axis. The MDX statement in Figure 15.8 contains a WHERE condition to limit the calculated results to the North American member in the territory attribute of the Markets dimension. The Sales values in the cells are smaller in Figure 15.8 than Figure 15.7 because of the slicer condition in Figure 15.8. For example, sales of classic cars in 2003 are 4,959 for North America, but 12,762 for all territories in 2003.

The SELECT statements in Figures 15.7 and 15.8 depict multiple dimensions on the columns axis and the usage of the default measure (Sales), not shown in either query axis. If no measure is specified on a query axis, the default measure is shown in the cells.

15.1.2 Pivot Table Tools for OLAP Queries

Unlike the SQL SELECT statement, the MDX SELECT statement is not typically used to query data cubes directly. Business analysts, typical users of data cubes, lack training of information technology professionals. The previous section indicated the tedious

FIGURE 15.6
MDX Query with a CrossJoin Operation and Associated Data Cube

FIGURE 15.7
MDX Query with a CrossJoin Operation and No Slicer

Product	Order Status: All Status Types; Time: 2003	2004	2005
Classic Cars	12,762	16,085	6,705
Motorcycles	4,031	5,906	2,771
Planes	3,833	5,820	2,207
Ships	2,844	4,309	1,346
Trains	1,000	1,409	409
Trucks and Buses	4,056	5,024	1,921
Vintage Cars	7,913	10,864	4,116

MDX Query

Run | Reset | Query Execution Time : 6 msec

```
1 SELECT CrossJoin({[Order Status].[All Status Types]}, {[Time].
  [2003], [Time].[2004], [Time].[2005]}) ON COLUMNS, {[Product].
  [Classic Cars], [Product].[Motorcycles], [Product].[Planes],
  [Product].[Ships], [Product].[Trains], [Product].[Trucks and
  Buses], [Product].[Vintage Cars]} ON ROWS FROM
  [SteelWheelsSales]
```

FIGURE 15.8
MDX Query with a Slicer Added to Figure 15.7

Product	Order Status: All Status Types; Time: 2003	2004	2005
Classic Cars	4,959	5,017	2,105
Motorcycles	1,744	2,809	568
Planes	977	2,224	592
Ships	702	1,642	537
Trains	409	326	177
Trucks and Buses	1,289	2,563	597
Vintage Cars	3,268	3,576	1,871

MDX Query

Run | Reset | Query Execution Time : 8 msec

```
1 SELECT CrossJoin({[Order Status].[All Status Types]}, {[Time].
  [2003], [Time].[2004], [Time].[2005]}) ON COLUMNS, {[Product].
  [Classic Cars], [Product].[Motorcycles], [Product].[Planes],
  [Product].[Ships], [Product].[Trains], [Product].[Trucks and
  Buses], [Product].[Vintage Cars]} ON ROWS FROM
  [SteelWheelsSales] WHERE Markets.Territory.NA
```

specification of MDX SELECT statement details even for relatively simple retrievals. The MDX SELECT statement involves substantial more complexity than indicated in the previous section. In practice, MDX provides a solid foundation for pivot table tools that hide the complexity of MDX while providing a convenient interface for business analysts.

Pivot tables have become the standard interface for data cubes. Microsoft Excel has provided a pivot table feature for many years. Other commercial vendors and open source projects also support pivot tables. This section provides an overview of two pivot table tools (Pivot4J plugin and WebPivotTable) to augment your background on data cube retrieval.

Pivot table tools provide a client interface to manipulate a data cube or other data source such as an Excel spreadsheet. To manipulate a data cube, a pivot table tool

depends on an OLAP server for cube storage and execution of MDX queries. Pivot4J provides a built-in OLAP server, while WebPivotTable depends on an external OLAP server. Pivot4J can also use an external OLAP server.

Pivot4J Pivot4J, an open source project, extends earlier open source projects, Olap4J and JPivot. Pivot4J provides an Application Program Interface (API) that developers can use to build a pivot table interface. The earlier project, JPivot provided both an API and pivot table interface. However, the open source community dropped JPivot in 2008 so developers must build an interface with Pivot4J. OLAP4J provides an open source API that generates MDX queries. Pentaho released OLAP4J in 2015 as a Java API that can use a variety of OLAP servers such as the open source Mondrian server and Microsoft SQL Server Analysis Services.

Pentaho has developed a Pivot4J plugin that can be installed in the community edition of Pentaho Business Analytics. The plugin uses the Pivot4J API and the Modrian OLAP server. Figure 15.9 depicts the interface for the Pivot4J plugin of Pentaho Business Analytics. The interface contains panes for feature buttons, OLAP Navigator, Cube Structure, Pivot Structure, Query Result, and MDX Query.

- The top pane contains buttons for various features.
- The OLAP (Online Analytic Process) Navigator (on the left) shows the selected cube (SteelWheelsSales).
- The Cube Structure pane shows the dimensions and measures of the selected cube.
- The Pivot Structure contains the dimensions and measures in the axes of the pivot table. Dimensions and measures can be dragged from the Cube Structure and dropped into the axes of the Pivot Structure.

Pivot4J provides duality between operations on pivot tables and MDX. Typically, a business analyst creates or modifies a query result by modifying the Pivot Structure pane or selected buttons in the feature pane. Alternatively, an analyst can type an MDX statement into the MDX Query pane. If the statement is meaningful, the result will be displayed in the Query Result pane. Figures 15.6 to 15.9 demonstrate duality between pivot table retrieval operations and MDX in Pivot4J.

WebPivotTable WebPivotTable, a Javascript based tool, executes in a browser by itself or integrates into websites. The original release of WebPivotTable mimicked features in Microsoft Excel pivot tables. Later releases of WebPivotTable extended its

FIGURE 15.9
Pivot4J Plugin Interface in Pentaho Business Analytics

FIGURE 15.10
WebPivotTable Interface

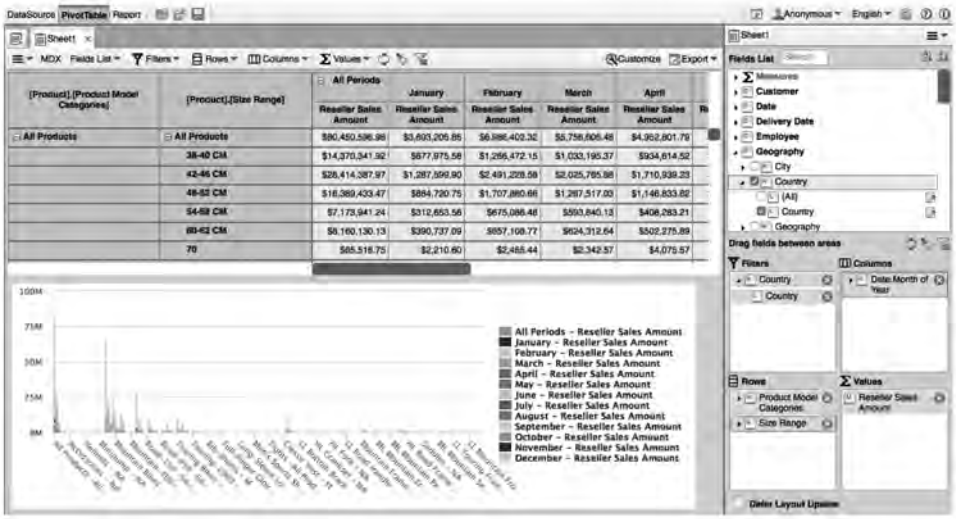
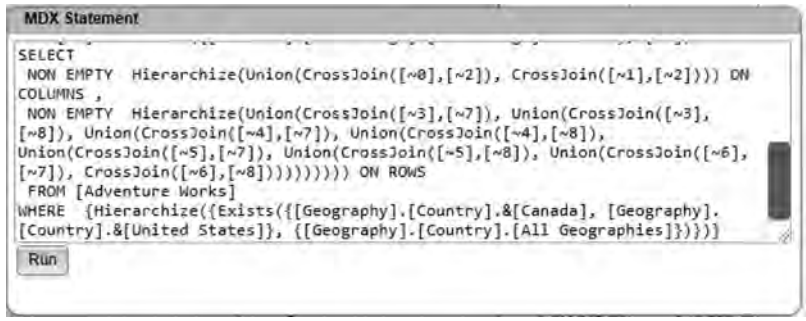


FIGURE 15.11
MDX SELECT Statement for Pivot Table in Figure 15.10



original goal to support external OLAP servers and MDX. WebPivotTable provides a free edition and commercial enterprise edition.

When using an OLAP data cube, WebPivotTable acts as a client connected to an OLAP server. With cube calculations performed on an external OLAP server, slow performance or server errors can occur for complex operations on a pivot table.

Similar to the Pivot4J plugin, WebPivotTable provides a convenient interface for manipulating data cubes. Figure 15.10 shows a pivot table query using the Microsoft AdventureWorks data cube. The AdventureWorks cube contains 16 dimensions with 47 measures as shown in the Fields List pane. The pivot table in Figure 15.10 contains Size Range and Product Model Categories on the Rows area, Date Month of Year on the Columns area, and Reseller Sales Amount measure in the Values area. The filter on Country acts as a slicer restricting calculations to selected countries.

Similar to the Pivot4J plugin, WebPivotTable provides duality between pivot table operations and MDX. The MDX statement displays after selecting the MDX button in Figure 15.10. The statement in Figure 15.11, generated by WebPivotTable, shows additional elements of MDX including the Hierarchize, Exists, and Union functions. The generated statement is tedious to read, demonstrating difficulty faced by a business analyst to use MDX directly.

15.2 SQL EXTENSIONS FOR SUBTOTAL CALCULATIONS

The GROUP BY clause was found inadequate for business intelligence applications. A major limitation of the original GROUP BY clause is the lack of subtotals in results. GROUP BY results only contain the lowest level totals for each combination of grouping columns. Pivot tables can have subtotals on any query dimension in an axis.

Subtotals are not limited to addition. Any statistical function can be used to summarize a dimension such as average and minimum.

A simple example depicts the lack of subtotals in GROUP BY results. Figure 15.12 compares a GROUP BY query result and associated data cube. The subtotals in the data cube show the sum of sales for rows (states) and columns (months) along with the grand total. In contrast, the GROUP BY result contains no subtotals. The GROUP BY result contains no missing values as only combinations with non-null values appear in the result. In contrast, a data cube shows missing values with a dash denoting a missing value. For example, the data cube contains a dash for State = "CA" and Month = "Jan" while the GROUP BY result does not contain this row.

As a response to the lack of subtotals, new summarization capabilities were added to the GROUP BY clause in the SQL:1999 standard. The extensions involve the ability to produce summary totals (CUBE and ROLLUP operators) as well as more precise specification of grouping columns (GROUPING SETS operator). This section describes these new parts of the GROUP BY clause using Oracle SQL as an enterprise DBMS that implements the standard SQL features. If you need a refresher about the GROUP BY clause, you should review GROUP BY coverage in Chapter 4 before embarking on this section.

The examples in the remainder of this chapter use the Store Sales Data Warehouse introduced in Chapter 13. Figure 15.13 displays the ERD for ease of reference.

CUBE Operator
an operator that augments the normal GROUP BY result with all combinations of subtotals. The CUBE operator is appropriate to summarize columns from independent dimensions rather than columns representing different levels of a single dimension.

15.2.1 CUBE Operator

The **CUBE operator** clause produces all possible subtotal combinations in addition to the normal totals shown in a GROUP BY clause. Because all possible subtotals are

GROUP BY Result

State	Month	SUM(Sales)
CA	Dec	100
CA	Feb	75
CO	Dec	150
CO	Jan	100
CO	Feb	200
CN	Dec	50
CN	Jan	75

Sales Pivot Table

State	Month			
	Dec	Jan	Feb	Total
CA	100	-	75	175
CO	150	100	200	450
CN	50	75	-	125
Total	300	175	275	750

```
SELECT State, Month, SUM(Sales)
...
GROUP BY State, Month
```

FIGURE 15.12
Comparison of GROUP BY Results and Pivot Table

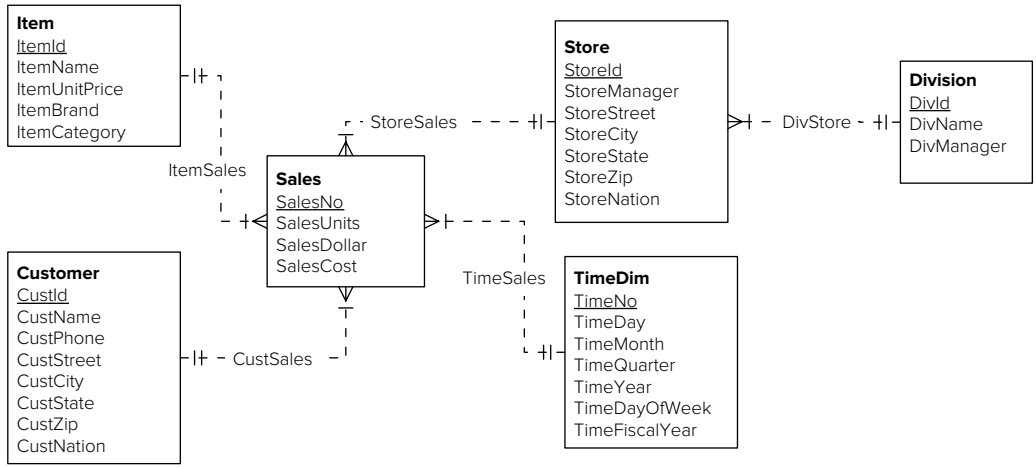


FIGURE 15.13
ERD Snowflake Schema for the Store Sales Data Warehouse

generated, the CUBE operator is appropriate to summarize columns from independent dimensions rather than columns representing different levels of the same dimension. For example, the CUBE operator would be appropriate to generate subtotals for all combinations of year, store state, and item brand. In contrast, a CUBE operation to show all possible subtotals of year, month, and day would have limited interest because of the hierarchy in the time dimension.

To depict the CUBE operator, Example 15.1 displays a SELECT statement with a GROUP BY clause containing just two columns. Only six rows are shown in the result so that the effect of the CUBE operator can be understood easily. With two values in the *StoreZip* column and three values in the *TimeMonth* column, the number of subtotal combinations is six (two *StoreZip* subtotals, three *TimeMonth* subtotals, and one grand total) as shown in Example 15.2. Blank values in the result represent a summary over all possible values of the column. For example, the row <80111, -, 33000> represents the total sales in the zip code 80111 over all months (- represents a do not care value for the month).

Example 15.1

GROUP BY clause and partial result without subtotals

```
SELECT StoreZip, TimeMonth, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId AND
       Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear = 2016
 GROUP BY StoreZip, TimeMonth;
```

StoreZip	TimeMonth	SumSales
80111	1	10000
80111	2	12000
80111	3	11000
80112	1	9000
80112	2	11000
80112	3	15000

Example 15.2 (Oracle)

GROUP BY clause and partial result with subtotals produced by the CUBE operator

```
SELECT StoreZip, TimeMonth, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear = 2016
 GROUP BY CUBE(StoreZip, TimeMonth);
```

StoreZip	TimeMonth	SumSales
80111	1	10000
80111	2	12000
80111	3	11000
80112	1	9000
80112	2	11000
80112	3	15000
80111		33000
80112		35000
	1	19000
	2	23000
	3	26000
		68000

With more than two grouping columns, the CUBE operator becomes more difficult to understand. Examples 15.3 and 15.4 extend Examples 15.1 and 15.2 with an additional grouping column (*TimeYear*). The number of rows in the result increases from 12 rows in the result of Example 15.3 without the CUBE operator to 36 rows in the result of Example 15.4 with the CUBE operator. For three grouping columns with *M, N*, and *P* unique values, the maximum number of subtotal rows produced by the CUBE operator is $M + N + P + M*N + M*P + N*P + 1$. Since the number of subtotal rows grows substantially with the number of grouped columns and the unique values per column, the CUBE operator should be used with caution when grouping on more than three columns.

Example 15.3

GROUP BY Clause with three grouping columns and the partial result without subtotals

```
SELECT StoreZip, TimeYear, TimeMonth, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear BETWEEN 2016 AND 2017
 GROUP BY StoreZip, TimeYear, TimeMonth;
```

StoreZip	TimeYear	TimeMonth	SumSales
80111	2016	1	10000
80111	2016	2	12000
80111	2016	3	11000
80112	2016	1	9000
80112	2016	2	11000
80112	2016	3	15000
80111	2017	1	11000

StoreZip	TimeYear	TimeMonth	SumSales
80111	2017	2	13000
80111	2017	3	12000
80112	2017	1	10000
80112	2017	2	12000
80112	2017	3	16000

Example 15.4 (Oracle)

GROUP BY clause with three grouping columns and the result with subtotals produced by the CUBE operator

```
SELECT StoreZip, TimeYear, TimeMonth, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear BETWEEN 2016 AND 2017
 GROUP BY CUBE(StoreZip, TimeYear, TimeMonth);
```

StoreZip	TimeYear	TimeMonth	SumSales
80111	2016	1	10000
80111	2016	2	12000
80111	2016	3	11000
80112	2016	1	9000
80112	2016	2	11000
80112	2016	3	15000
80111	2017	1	11000
80111	2017	2	13000
80111	2017	3	12000
80112	2017	1	10000
80112	2017	2	12000
80112	2017	3	16000
80111		1	21000
80111		2	25000
80111		3	23000
80112		1	19000
80112		2	22000
80112		3	31000
80111	2016		33000
80111	2017		36000
80112	2016		35000
80112	2017		38000

StoreZip	TimeYear	TimeMonth	SumSales
	2016	1	19000
	2016	2	23000
	2016	3	26000
	2017	1	21000
	2017	2	25000
	2017	3	28000
80111			69000
80112			73000
		1	40000
		2	48000
		3	54000
	2016		68000
	2017		74000
			142000

The CUBE operator is not a primitive operator. The result of a CUBE operation can be produced using a number of SELECT statements connected by UNION operations, as shown in Example 15.5. The additional SELECT statements generate subtotals for each combination of grouped columns. With two grouped columns, three additional SELECT statements are needed to generate the subtotals. With N grouped columns, $2^N - 1$ additional SELECT statements are needed. Obviously, the CUBE operator is much easier to write than a large number of additional SELECT statements.

Example 15.5

Rewrite Example 15.2 without using the CUBE operator. In each additional SELECT block, the NULL value replaces the column in which totals are not generated

```
SELECT StoreZip, TimeMonth, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear = 2016
 GROUP BY StoreZip, TimeMonth
 UNION
SELECT StoreZip, NULL, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear = 2016
 GROUP BY StoreZip
UNION
```

```
SELECT NULL, TimeMonth, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear = 2016
 GROUP BY TimeMonth
UNION
SELECT NULL, NULL, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear = 2016;
```

15.2.2 ROLLUP Operator

ROLLUP Operator
an operator that augments the normal GROUP BY result with a partial set of subtotals. The ROLLUP operator is appropriate to summarize levels from a dimension hierarchy.

The SQL **ROLLUP operator** provides a similar capability to the roll-up operator for data cubes. The roll-up operator for data cubes produces totals for coarser parts of a dimension hierarchy. The SQL ROLLUP operator produces subtotals for each ordered subset of grouped columns to simulate the effects of the roll-up operator for data cubes. For example, the SQL operation `ROLLUP(TimeYear, TimeQuarter, TimeMonth, TimeDay)` produces subtotals for the column subsets <*TimeYear, TimeQuarter, TimeMonth*>, <*TimeYear, TimeQuarter*>, <*TimeYear*> as well as the grand total. As this example implies, the order of columns in a ROLLUP operation is significant.

As the previous paragraph indicates, the ROLLUP operator produces only a partial set of subtotals for the columns in a GROUP BY clause. Examples 15.6 and 15.7 demonstrate the ROLLUP operator on hierarchically related dimensions. Example 15.6 uses two grouping columns (*TimeYear* and *TimeQuarter*), producing subtotals on *TimeYear* and the grand total. Example 15.7 uses three grouping columns (*TimeYear, TimeQuarter*, and *TimeMonth*), producing subtotals for the column subsets <*TimeYear, TimeQuarter*>, *TimeYear*, and the grand total. The partial result in Example 15.7 only contains data for one month in each quarter so the month sales equals the subtotal of quarter sales containing the month.

Example 15.6 (Oracle)

GROUP BY clause with two grouping columns and result with subtotals produced by the ROLLUP operator

```
SELECT TimeYear, TimeQuarter, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear BETWEEN 2016 AND 2017
 GROUP BY ROLLUP(TimeYear, TimeQuarter);
```

TimeYear	TimeQuarter	SumSales
2016	1	52490
2016	2	61720

TimeYear	TimeQuarter	SumSales
2016	3	56866
2016	4	59836
2016		230932
2017	1	74910
2017	2	89110
2017	3	75086
2017	4	81288
2017		320394
		551326

Example 15.7 (Oracle)

GROUP BY clause with three grouping columns and partial result with subtotals produced by the ROLLUP operator

```
SELECT TimeYear, TimeQuarter, TimeMonth, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear BETWEEN 2016 AND 2017
 GROUP BY ROLLUP(TimeYear, TimeQuarter, TimeMonth);
```

TimeYear	TimeQuarter	TimeMonth	SumSales
2016	1	2	52490
2016	1		52490
2016	2	5	61720
2016	2		61720
2016	3	7	56866
2016	3		56866
2016	4	10	59386
2016	4		59386
2016			230932
2017	1	2	74910
2017	1		74910
2017	2	5	89110
2017	2		89110
2017	3	7	75086
2017	3		75086
2017	4	10	81288
2017	4		81288
2017			320394
			551326

Examples 15.8 and 15.9 contrast the ROLLUP operator with the CUBE operator. Note that Example 15.8 contains three subtotal rows compared to six subtotal rows in Example 15.2 with the CUBE operator. In Example 15.9, subtotals are produced for the values in the column combinations <*StoreZip, TimeYear*>, <*StoreZip*>, and the grand total. In Example 15.4 with the CUBE operator, subtotals are also produced for the values in the column combinations <*StoreZip, TimeYear*>, <*TimeMonth, TimeYear*>, <*TimeMonth*>, and <*TimeYear*>. Thus, the ROLLUP operator produces far fewer subtotal rows compared to the CUBE operator as the number of grouped columns and unique values per column increases.

Example 15.8 (Oracle)

ROLLUP example compared with Example 15.2 to understand the difference between the CUBE and ROLLUP operators

```
SELECT StoreZip, TimeMonth, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear = 2016
 GROUP BY ROLLUP(StoreZip, TimeMonth);
```

StoreZip	TimeMonth	SumSales
80111	1	10000
80111	2	12000
80111	3	11000
80112	1	9000
80112	2	11000
80112	3	15000
80111		33000
80112		35000
		68000

Example 15.9 (Oracle)

ROLLUP example compared with Example 15.4 to understand the difference between the CUBE and ROLLUP operators

```
SELECT StoreZip, TimeYear, TimeMonth, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear BETWEEN 2016 AND 2017
 GROUP BY ROLLUP(StoreZip, TimeYear, TimeMonth);
```

StoreZip	TimeYear	TimeMonth	SumSales
80111	2016	1	10000
80111	2016	2	12000
80111	2016	3	11000
80112	2016	1	9000
80112	2016	2	11000
80112	2016	3	15000
80111	2017	1	11000
80111	2017	2	13000
80111	2017	3	12000
80112	2017	1	10000
80112	2017	2	12000
80112	2017	3	16000
80111	2016		33000
80111	2017		36000
80112	2016		35000
80112	2017		38000
80111			69000
80112			73000
			142000

Like the CUBE operator, the ROLLUP operator is not a primitive operator. The result of a ROLLUP operation can be produced using a number of SELECT statements connected by the UNION operator as shown in Example 15.10. The additional SELECT statements generate subtotals for each ordered subset of grouped columns. With three grouped columns, three additional SELECT statements are needed to generate the subtotals. With *N* grouped columns, *N* additional SELECT statements are needed. In each additional SELECT statement, the NULL value replaces the column in which totals are not generated. Obviously, the ROLLUP operator is much easier to write than a large number of SELECT blocks.

Example 15.10

Rewrite of Example 15.7 without using the ROLLUP operator

```
SELECT TimeYear, TimeQuarter, TimeMonth, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear BETWEEN 2016 AND 2017
 GROUP BY TimeYear, TimeQuarter, TimeMonth
 UNION
SELECT StoreZip, TimeYear, NULL, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
```

```
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear BETWEEN 2016 AND 2017
 GROUP BY TimeYear, TimeQuarter
UNION
SELECT TimeYear, NULL, NULL, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear BETWEEN 2016 AND 2017
 GROUP BY TimeYear
UNION
SELECT NULL, NULL, NULL, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.No
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear BETWEEN 2016 AND 2017;
```

15.2.3 GROUPING SETS Operator

GROUPING SETS Operator an operator in the GROUP BY clause that requires explicit specification of column subsets. The GROUPING SETS operator is appropriate when precise control over subtotals is required.

The **GROUPING SETS operator** provides more flexibility than the CUBE and ROLLUP operators. The GROUPING SETS operator requires explicit specification of column subsets, even normal GROUP BY columns. In contrast, the CUBE and ROLLUP operators provide implicit specification of subtotals. The GROUPING SETS operator is appropriate when precise control over subtotals is needed. If explicit control is not required, the CUBE and ROLLUP operators provide more succinct specification.

To depict the GROUPING SETS operator, the previous examples are recast using the GROUPING SETS operator. In Example 15.11, the GROUPING SETS operator involves subtotals for the *StoreZip* and *TimeMonth* columns along with the grand total denoted by the empty parentheses. The subset (*StoreZip, TimeMonth*) also must be specified because all column combinations must be explicitly specified, even the normal grouping without the GROUPING SETS operator. Example 15.12 contains eight column combinations to provide the same result as Example 15.4 with the CUBE of three columns. Example 15.13 contains three column combinations to provide the same result as Example 15.7 with the ROLLUP of three columns.

Example 15.11 (Oracle)

GROUP BY clause using the GROUPING SETS operator producing the same result as Example 15.2

```
SELECT StoreZip, TimeMonth, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND StoreNation IN ('USA', 'Canada') AND TimeYear = 2016
GROUP BY GROUPING SETS((StoreZip, TimeMonth), StoreZip,
       TimeMonth, ());
```

Example 15.12 (Oracle)

GROUP BY Clause using the GROUPING SETS operator producing the same result as Example 15.4

```
SELECT StoreZip, TimeYear, TimeMonth, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear BETWEEN 2016 AND 2017
 GROUP BY GROUPING SETS((StoreZip, TimeYear, TimeMonth),
    (StoreZip, TimeMonth), (StoreZip, TimeYear),
    (TimeMonth, TimeYear), StoreZip, TimeMonth, TimeYear, () );
```

Example 15.13 (Oracle)

GROUP BY clause using the GROUPING SETS operator producing the same result as Example 15.7

```
SELECT TimeYear, TimeQuarter, TimeMonth, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear BETWEEN 2016 AND 2017
 GROUP BY GROUPING SETS((TimeYear, TimeQuarter, TimeMonth),
     (TimeYear, TimeQuarter), TimeYear,() );
```

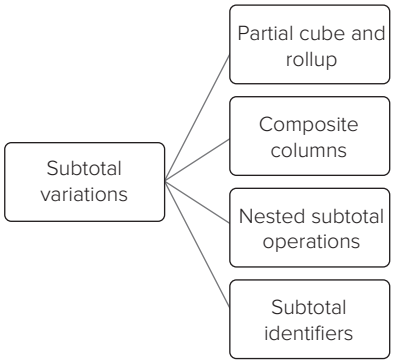
Example 15.14 depicts a situation in which the GROUPING SETS operator is preferred to the CUBE operator. Because the *TimeYear* and *TimeMonth* columns are from the same dimension hierarchy, a full cube usually is not warranted. Instead,

Example 15.14 (Oracle)

GROUP BY clause using the GROUPING SETS operator to indicate the column combinations from which subtotals are needed

```
SELECT StoreZip, TimeYear, TimeMonth, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear BETWEEN 2016 AND 2017
 GROUP BY GROUPING SETS((StoreZip, TimeYear, TimeMonth),
    (StoreZip, TimeYear), (TimeYear, TimeMonth),
    StoreZip, TimeYear, () );
```

FIGURE 15.14
Important SQL Subtotal Variations



the GROUPING SETS operator can be used to specify the column combinations from which subtotals are needed. Subtotals involving *TimeMonth* without *TimeYear* are excluded in Example 15.14 but are included in a full CUBE operation.

15.2.4 Variations of Subtotal Operators

The CUBE, ROLLUP, and GROUPING SETS operators presented in previous subsections provide a solid foundation for subtotals in the GROUP BY clause. The SQL standard provides more flexibility to combine subtotals operators and ability to identify generated subtotals. This flexibility and identification ability can be useful in specialized situations. This subsection provides examples of subtotal variations depicted in Figure 15.14 and summarized in the following list.

- Partial cube and rollup operators on a subset of grouping columns
- Composite columns to treat a combination of columns as a single column
- Nesting cube and rollup operators inside the GROUPING SETS operator
- Usage of subtotal identifiers to indicate the grouping level of result rows

A partial cube produces subtotals for a subset of independent columns. In Example 15.15, the clause, `GROUP BY TimeMonth, CUBE(DivId, StoreZip)`, produces totals on the subtotal groups <*TimeMonth, DivId, StoreZip*>, <*TimeMonth, DivId*>, <*TimeMonth, StoreZip*>, and <*TimeMonth*>. *TimeMonth* concatenates with each subtotal group generated by the CUBE operator. For example, *TimeMonth* concatenates with the grand total group generated by the CUBE operator so that the last subtotal group contains *TimeMonth*.

Example 15.15 (Oracle)

Partial CUBE example

```
SELECT TimeMonth, DivId, StoreZip, SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear = 2016
 GROUP BY TimeMonth, CUBE(DivId, StoreZip)
 ORDER BY TimeMonth, DivId, StoreZip;
```

A partial rollup produces subtotals for a subset of hierarchically related columns. In Example 15.16, the clause, `GROUP BY StoreState, ROLLUP(TimeMonth, TimeDay)`, produces totals on the subtotal groups <*StoreState, TimeMonth, TimeDay*>, <*StoreState, TimeMonth*>, and <*StoreState*>. *StoreState* concatenates with each subtotal group generated by the ROLLUP operator. For example, *StoreState* concatenates with the grand total group (empty parentheses) generated by the ROLLUP operator so that the last subtotal group contains *StoreState*.

Composite columns can be used with the CUBE or ROLLUP operators to skip some subtotal groups. In Example 15.17, the clause, `GROUP BY ROLLUP(StoreNation, (StoreState, StoreCity)`, produces totals on the subtotal groups <*StoreNation, StoreState, StoreCity*>, <*StoreNation*>, and <>. The ROLLUP operation skips the subtotal group (*StoreNation, StoreState*) because the ROLLUP operator treats the composite column (*StoreState, StoreCity*) as a single column.

Example 15.16 (Oracle)

Partial ROLLUP example

```
SELECT StoreState, TimeMonth, TimeDay,
       SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear = 2016
 GROUP BY StoreState, ROLLUP(TimeMonth, TimeDay)
 ORDER BY StoreState, TimeMonth, TimeDay;
```

Example 15.17 (Oracle)

Composite columns example

```
SELECT StoreNation, StoreState, StoreCity,
       SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND TimeYear = 2016
 GROUP BY ROLLUP(StoreNation, (StoreState, StoreCity))
 ORDER BY StoreNation, StoreState, StoreCity;
```

Nesting a cube or rollup operation inside a GROUPING SETS operation generates different subtotals than a partial CUBE or ROLLUP operation. As previously shown, a partial CUBE or ROLLUP operation concatenates with the other GROUP BY columns. In contrast, concatenation does not occur when nesting a CUBE or ROLLUP operation inside a GROUPING SETS operation.

Example 15.18 uses a nested rollup operation with a composite column. The GROUP BY clause generates subtotals for (*StoreNation, StoreState, StoreCity*), *StoreNation*, and the grand total for the nested ROLLUP operation and *TimeMonth* subtotals for the GROUPING SETS operation. *TimeMonth* does not concatenate with the ROLLUP subtotals.

Example 15.18 (Oracle)

Nested ROLLUP example

```
SELECT TimeMonth, StoreNation, StoreState, StoreCity,
       SUM(SalesDollar) AS SumSales
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND (StoreNation = 'USA' OR StoreNation = 'Canada')
   AND TimeYear = 2016
 GROUP BY GROUPING SETS(TimeMonth,
     ROLLUP(StoreNation, (StoreState, StoreCity) ) )
 ORDER BY TimeMonth, StoreNation, StoreState, StoreCity;
```

Sometimes it is useful to distinguish subtotal groups. The GROUPING_ID function generates a hierarchical group number for each result row. For example a CUBE of three columns generates 8 subtotal groups. The GROUPING_ID function, using the CUBE columns, labels rows with a number from 0 to 7. Other functions are GROUP_ID to identify duplicate subtotal rows and GROUPING to distinguish normal grouping rows (0) from subtotal rows (1).

Example 15.19 shows a GROUPING_ID function using three columns. It is typical to include all grouping columns in a GROUPING_ID function although not necessary. The GROUPING_ID function labels each row with a grouping number from 0 to 7 with 0 for the finest level of totals (*StoreZip, TimeMonth*, and *DivId*) and 7 for the grand total. The ORDER BY sorts by GROUPING_ID to cluster rows in the same subtotal group.

Example 15.19 (Oracle)

GROUPING_ID example

```
SELECT StoreZip, TimeMonth, DivId, SUM(SalesDollar) AS SumSales,
       GROUPING_ID(StoreZip, TimeMonth, DivId) AS Group_Level
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND StoreNation IN ('USA', 'Canada') AND TimeYear = 2016
 GROUP BY CUBE (StoreZip, TimeMonth, DivId)
 ORDER BY Group_Level;
```

The subtotal variations are specialized so actual uses are not easy to identify. The GROUPING SETS operator provides complete control of subtotal rows so these variations do not provide additional control. However, the variations allow more compact specification than GROUPING SETS when complete CUBE and ROLLUP are not needed. Beyond these specialized situations, you can use these variations to amaze and amuse coworkers. If you understand these variations, you have a solid understanding of the SQL subtotal operators.

15.3 SQL EXTENSIONS FOR ANALYTIC FUNCTIONS

Business intelligence applications typically involve analysis that combines data retrieval and computations. The GROUP BY clause, extended with subtotal operators, supports capabilities of data cubes but lacks capabilities for common business intelligence applications. The SQL SELECT statement requires a substantial extension, not just extensions to an individual clause.

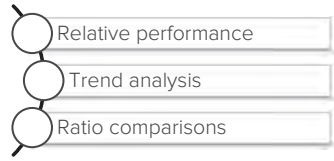
This section covers analytic function extensions to the SQL SELECT statement to support business intelligence applications. This section begins with motivation and processing overview of analytic function extensions. The next three sections cover query formulation for prominent business intelligence applications involving relative performance, trend analysis, and quantitative contributions.

15.3.1 Motivation and Processing Overview

Business analysts use a data warehouse for business intelligence applications combining retrieval with complex computations. These applications are beyond the scope of queries typically supported by the SQL SELECT statement. Figure 15.15 depicts common business intelligence analysis that a data warehouse can support. Relative

performance identifies top and worst performers by ranking business units such as stores by sales and agents by commissions. Trend analysis depicts changes between time periods often involving summary calculations on sliding windows such as moving averages of security prices over a number of periods. Ratio comparisons show top or bottom thresholds such as the top ten percent of stores by sales. In addition, ratios show contributions to a whole such as a region's sales as part of total sales.

FIGURE 15.15
Common Business Intelligence Analysis



Organizations found the SQL SELECT statement inadequate to support these common types of business intelligence analysis. Figure 15.16 depicts factors influencing extensions to the SQL SELECT statement for common business intelligence analysis. Before extensions to the SQL SELECT statement, implementing business intelligence solutions required a complex skill set involving data retrieval, procedural coding, and external tool usage. Organizations experienced difficulties to find individuals possessing the necessary skill set. Productivity was poor for developing business intelligence applications with complex SELECT statements and procedural coding often required. Performance was slow with SQL compilers often developing poorly performing plans for complex SELECT statements. In addition, complex calculations were typically done outside of SQL statements so SQL compilers could not optimize both data retrieval and computations.

To overcome these deficiencies, the SQL SELECT statement needed a major extension for both processing and statement specification. The processing and statement requirements make the analytic function extension more complex than the subtotal operator extension covered in section 15.2. Analytic functions were initially added to SQL:1999 with major enterprise DBMS vendors now providing substantial support. The other subsections cover major analytic functions supported by Oracle. Most enterprise DBMS vendors and the SQL standard support these functions along with many other analytic functions.

Analytic functions differ from aggregate functions although both operate on groups of rows. An **analytic function** computes multiple values for a groups of rows, while an aggregate function computes a single value for a group of rows. Thus, an analytic function preserves the number of rows in a group of rows, while an aggregate function reduces a group of rows to a single row. Processing of analytic functions occurs after aggregate functions so a single SELECT statement can contain both aggregate functions and analytic functions.

Analytic Function
an extension in the SQL SELECT statement to support business intelligence applications. An analytic function computes multiple values for a group of rows, preserving the number of rows in the group. Processing of an analytic function occurs after computing aggregate functions used in a GROUP BY clause.

Processing of analytic functions involves a new step after GROUP BY processing. As depicted in Figure 15.17, analytic processing occurs after row and group processing so that calculations can be performed on GROUP BY results. Many queries with analytic functions use grouping so that analytic calculations can use summary calculations for groups of rows. Analytic function processing involves organizing results into partitions, evaluating functions over partitions, and then ordering the partitions. Ordering provides a criteria for an analytic function rather than arrangement of the final result for the ORDER BY clause.

The revised processing model has three subtle points. First, processing of analytic functions often involves partitions and grouping in the same statement. Partitioning

FIGURE 15.16
Factors influencing Analytic Function Extensions

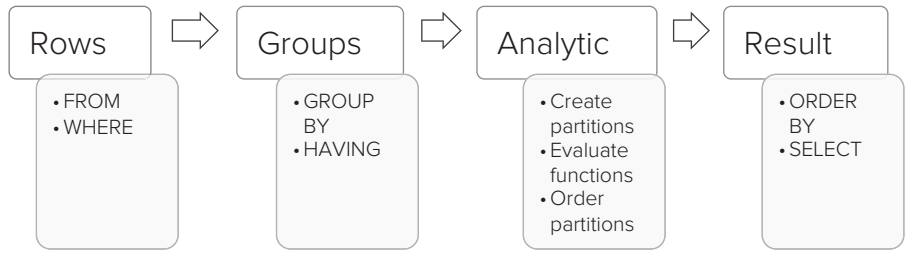


FIGURE 15.17
Extension of SELECT Statement Processing for Analytic Functions

in analytic functions occurs after grouping and calculation of aggregate functions. Second, ordering in analytic functions differs from ordering of a query result. Ordering applies to some analytic functions as a criteria. Third, the appearance of analytic functions in a SELECT statement seems to conflict with the evaluation order. Analytic functions appear in the SELECT clause but evaluation occurs after row and group processing before result processing in the final step. Writing and executing SELECT statements with analytic functions clarifies these subtle points.

15.3.2 Query Formulation for Relative Performance

Identifying top and worst performers involves functions that provide a relative ordering or ranking of entities. The SQL standard provides a variety of analytic functions for ranking. This section first introduces basic syntax before covering details of prominent functions for relative performance.

Before examining complete SELECT statements, you should focus just on the basic syntax for analytic functions. The SELECT clause contains result columns and analytic functions. Figure 15.18 shows the basic syntax of an analytic function along with partial statement examples. The syntax of an analytic function involves an analytic function name followed by parentheses surrounding an optional list of columns. The syntax continues with the OVER keyword followed by parentheses surrounding optional ORDER BY keywords and an ordering specification. The ordering indicates a criteria for function evaluation, not a final ordering of results.

Two partial examples in Figure 15.18 with the RANK function depict the basic syntax. The RANK function does not use a ColumnList so empty parentheses are used. The first example ranks on the *ItemPrice* column. The AS keyword renames the computed rank column. The second example ranks on an aggregate function, the sum of *SalesDollar*. The aggregate function, SUM, indicates that a complete SELECT statement requires a GROUP BY clause.

Example 15.20 ranks items by unit price. In the statement, you should note the RANK function with the empty parentheses, the OVER keyword, and the ORDER BY specification inside the OVER clause. The ORDER BY clause determines ranking by ascending order (default) of *ItemUnitPrice*. Sample rows indicate the values of the RANK function with the smallest unit price (12.00) receiving the top rank (1).
Example 15.21 ranks customers by descending sum of dollar sales. Because the RANK function uses a SUM function, the SELECT statement requires a GROUP BY clause. The ORDER BY specification in the OVER clause indicates that customers are ranked using the SUM function on the *SalesDollar* column in descending order. Thus, the first result row with the largest sum of sales obtains the top ranking (1).

FIGURE 15.18
Basic Syntax and Partial Examples of Analytic Functions

```
Syntax: <AnalyticFunction> ( [<ColumnList>] )
          OVER ( [ ORDER BY <OrderSpec> ] )
Partial Example 1: RANK() OVER ( ORDER BY ItemPrice ) AS RankUnitPrice
Partial Example 2: RANK() OVER ( ORDER BY SUM(SalesDollar) ) AS RankSales
```

Example 15.20 (Oracle)

RANK example over entire result table and partial results

```
SELECT ItemId, ItemBrand, ItemUnitPrice,
  RANK() OVER ( ORDER BY ItemUnitPrice ) AS RankUnitPrice
  FROM Item;
```

ItemId	ItemBrand	ItemUnitPrice	RankUnitPrice
I1412138	Ethlite	12.00	1
I1445671	Intersafe	14.99	2
I6677900	Connex	25.69	3
I3455443	Connex	38.00	4

Example 15.21 (Oracle)

RANK function using a SUM function over entire result table along with partial query results

```
SELECT CustName, SUM(SalesDollar) AS SumSales,
  RANK() OVER (ORDER BY SUM(SalesDollar) DESC) AS SumSalesRank
  FROM Sales, Customer
  WHERE Sales.CustId = Customer.CustId
  GROUP BY CustName;
```

CustName	SumSales	SumSalesRank
Sheri Gordon	556322	1
Wally Jones	94004	2
Jim Glussman	91100	3
Candy Kendall	90664	4

Examples 15.20 and 15.21 use the RANK function over an entire result table. To provide rankings on multiple groups of rows, analytic functions can be applied to partitions, not just an entire result table. As indicated in the processing model (Figure 15.17), partitioning occurs twice in a SELECT statement with both a GROUP BY clause and partitioning in an analytic function. This situation is common because analytic functions are often applied to row summaries created by the GROUP BY clause. You should remember that analytic function processing occurs after GROUP BY processing with analytic functions often evaluated on row summaries generated in GROUP BY results.

The extended syntax shown in Figure 15.19 indicates optional partitioning as shown by square brackets surrounding the PARTITION BY keywords followed by a partition specification. The PARTITION BY keywords followed by a list of columns indicates that the associated analytic function is computed multiple times, once for each set of rows in a partition. The optional ordering details follow the partition details with no changes from the basic syntax.

FIGURE 15.19
Extended Syntax and Partial Example of an Analytic Functions

Syntax:

```
<AnalyticFunction> ( [<ColumnList>] )
  OVER ( [ PARTITION BY <PartitionSpec> ] [ ORDER BY <OrderSpec> ] )
```

Partial Example:

```
RANK() OVER (
 PARTITION BY CustState
 ORDER BY SUM(SalesRank) ) AS SalesRank
```

The partial example in Figure 15.19 extends example 1 in Figure 15.18 with the PARTITION BY clause. The RANK function evaluates on sets of rows with the same *CustState* value. The RANK function values start over for each *CustState* value. For example, if 10 CustState values exist in the results before analytic function evaluation, the RANK function generates 10 different rankings. A separate ranking will be determined for all rows with the same *CustState* value.

Example 15.22 extends Example 15.21 with partitioning. Example 15.22 ranks customers by the sum of dollar sales with partitioning on customer state. The SELECT statement uses the PARTITION BY keywords with the *CustState* column. The ORDER BY keyword ensures that the results are sorted on *CustState*. Within each state, the RANK function arranges rows by ranking value.

Example 15.22 (Oracle)

RANK function with partitioning on GROUP BY result

```
SELECT CustState, CustName, SUM(SalesDollar) AS SumSales,
   RANK() OVER (PARTITION BY CustState
                ORDER BY SUM(SalesDollar) DESC) AS SalesRank
  FROM Sales, Customer
  WHERE Sales.CustId = Customer.CustId
  GROUP BY CustState, CustName
  ORDER BY CustState;
```

CustState	CustName	SumSales	SalesRank
BC	Larry Styles	50620	1
CO	Sheri Gordon	556322	1
CO	Jim Glussman	91100	2
CO	Jerry Wyatt	87420	3
CO	Mike Boren	47412	4

Oracle provides additional functions for relative performance besides RANK. DENSE_RANK differs from RANK on duplicate values. RANK creates a gap on the next value after duplicates. DENSE_RANK does not create gaps. For example, golf leaderboards are typically reported using gaps. A leaderboard with -10, -9, -9, -8 displays with gaps (RANK function) as 1, 2, 2, 4. Without gaps (DENSE_RANK function), the leaderboard displays as 1, 2, 2, 3.

The NTILE and ROW_NUMBER function are less widely used than RANK and DENSE_RANK. The NTILE function divides rows into equal divisions. For example, NTILE(4) divides rows into 4 divisions or quartiles. ROW_NUMBER generates a total order of rows with row numbers 1 to the number of rows. For rows with duplicate values, ROW_NUMBER may generate inconsistent results depending on other parts of a statement.

Example 15.23 demonstrates each ranking function to depict differences. Each analytic function uses the same criteria, descending sum of unit sales. The analytic functions evaluate over the entire GROUP BY result on customer zip code. No partitioning is specified to focus just on the analytic functions.

Example 15.23 (Oracle)

Common qualitative ranking functions on entire GROUP BY result

```
SELECT CustZip, SUM(SalesUnits) AS SumSalesUnits,
  RANK() OVER (ORDER BY SUM(SalesUnits) DESC) SURank,
  DENSE_RANK() OVER (ORDER BY SUM(SalesUnits) DESC) SUDenseRank,
  NTILE(4) OVER (ORDER BY SUM(SalesUnits) DESC) SUNTile,
  ROW_NUMBER() OVER (ORDER BY SUM(SalesUnits) DESC) SURowNum
 FROM Sales, Customer
 WHERE Sales.CustId = Customer.CustId
 GROUP BY CustZip;
```

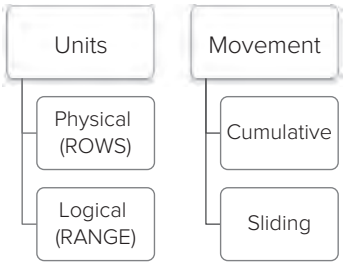
CustZip	SumSalesUnits	SURank	SUDenseRank	SUNTile	SURowNum
80129-5543	11440	1	1	1	1
98105-1093	2000	2	2	1	2
80111-0033	1960	3	3	1	3
98105-3345	1760	4	4	2	4
80222-0022	1720	5	5	2	5
98104-2211	1272	6	6	3	6
98178-3311	1272	6	6	3	7
80113-5431	1152	8	7	4	8
98103-1121	720	9	8	4	9

15.3.3 Query Formulation for Trend Analysis

After coverage of the basic and extended syntax of analytic functions as well as functions for relative performance, you are ready for extensions for another prominent type of business intelligence analysis. Trend analysis in finance and forecasting typically involves changes in numeric variables such as sales or stock prices in sets of rows known as windows. Windows are typically defined by time intervals such as years or months. Typical examples of window comparisons are a 90 day moving average of stock prices, percentage annual sales growth, performance of an advertising campaign over recent months, and cumulative sales performance for various organizational units. Analytic function extensions for window comparisons provide easier application development with a reduced skill set, increased software development productivity, and improved execution performance.

Before studying syntax and examples, you should understand basic window concepts as summarized in Figure 15.20. A window contains a collection of rows in which a numeric variable is calculated. A window can be specified in units of physical rows using the ROWS keyword or logical rows using the RANGE keyword. Logical windows are specified by values such as number of days. Numeric variables are calculated in windows so that variables can be compared across windows. A cumulative window is fixed on one end, typically the beginning and changes on the other end. A sliding window changes on both ends.

FIGURE 15.20
Window Concepts for SQL Analytic Functions



A window specification, after ordering details, is optional for analytic functions as noted in the syntax specification shown in Figure 15.21. Common summary functions such as AVG, SUM, COUNT, MIN, MAX, and VARIANCE can be used with a window specification. The Oracle documentation provides a complete list of aggregate functions that can be used with a window specification.

A window specification beginning with ROWS indicates a physical window. For example, `ROWS UNBOUNDED PRECEDING` indicates a window of the current row and all preceding rows. `ROWS 2 PRECEDING` indicates the current row and the two previous rows. `ROWS 3 FOLLOWING` indicates the current row and the next three rows. Logical window specifications are presented later in this subsection.

Figure 15.22 depicts a cumulative, physical window specified as `ROWS UNBOUNDED PRECEDING`. Initially, the window is just a single row. When processing the second row, the window is the current row (second row) and previous rows (first row). Figure 15.22 shows the window for the fifth row comprising the current row (row 5) and the previous rows (rows 1 to 4). The arrow indicates the direction of window movement from the first row towards the last physical row. For each window, an aggregate function such as SUM is calculated.

Example 15.24 demonstrates a SELECT statement for a cumulative physical window using the SUM function. The SELECT statement calculates the cumulative sum of dollar sales by zip code and year over the entire result without partitioning. The window specification, `ROWS UNBOUNDED PRECEDING`, indicates a cumulative physical

FIGURE 15.21
Extended Syntax and Partial Examples for Window Specification

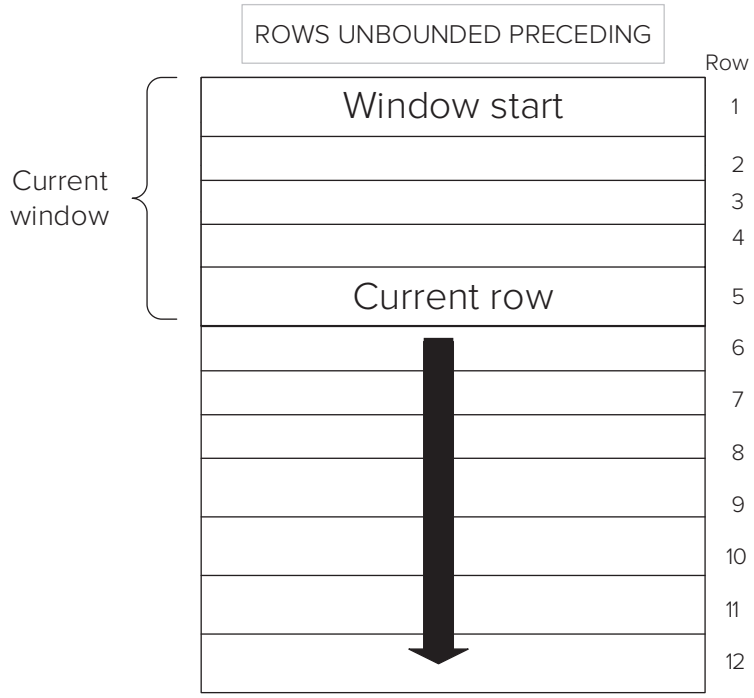
Syntax

```
<AnalyticFunction> ( [<ColumnList>] )
  OVER ( [ PARTITION BY <PartitionSpec> ] [ ORDER BY <OrderSpec> ]
    [<WindowSpec>] )
```

Partial examples of window specification

```
ROWS UNBOUNDED PRECEDING
ROWS 2 PRECEDING
ROWS 3 FOLLOWING
```

FIGURE 15.22
Cumulative Window Depiction



window. Note that window calculations involve the sum of the sum of dollar sales. Processing of the GROUP BY clause calculates the sum of dollar sales so analytic function processing just calculates the cumulative sum of sales. For clarity, the result contains both sum of sales and cumulative sales.

Example 15.24 (Oracle)

Cumulative sales by zip code and year over the entire grouping result along with partial result

```
SELECT StoreZip, TimeYear, SUM(SalesDollar) AS SumSales,
  SUM(SUM(SalesDollar)) OVER
   (ORDER BY StoreZip, TimeYear
    ROWS UNBOUNDED PRECEDING ) AS CumSumSales
 FROM Store, TimeDim, Sales
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
 GROUP BY StoreZip, TimeYear;
```

StoreZip	TimeYear	SumSales	CumSumSales
80111-0033	2014	89572	89572
80111-0033	2015	91972	181544
80111-0033	2016	43056	224600
80111-0033	2017	133238	357838
80129-5543	2014	92360	450198

Example 15.25 extends example 15.24 with partitioning. The SELECT statement calculates the cumulative sum of dollar sales by zip code and year over partitions of store zip code values. The partitioning clause with the PARTITIONED BY keywords

Example 15.25 (Oracle)

Cumulative sales by zip code and year with partitioning along with partial result

```
SELECT StoreZip, TimeYear, SUM(SalesDollar) AS SumSales,
  SUM(SUM(SalesDollar)) OVER (PARTITION BY StoreZip
    ORDER BY StoreZip, TimeYear
    ROWS UNBOUNDED PRECEDING ) AS CumSumSales
 FROM Store, TimeDim, Sales
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
 GROUP BY StoreZip, TimeYear;
```

StoreZip	TimeYear	SumSales	CumSumSales
80111-0033	2014	89572	89572
80111-0033	2015	91972	181544
80111-0033	2016	43056	224600
80111-0033	2017	133238	357838
80129-5543	2014	92360	92360
80129-5543	2015	92260	185020

indicates that window calculation restarts for each zip code value. The window specification, `ROWS UNBOUNDED PRECEDING` indicates a cumulative physical window.

After initial coverage of cumulative physical windows specified with the ROWS keyword (Figure 15.20), this subsection now covers sliding windows, both physical and logical. Sliding windows, also known as moving windows, change on both ends. Logical windows, indicated by the RANGE keyword, are specified for an ordering variable such as shipment date or number of years. The RANGE values indicate starting and ending points of a window.

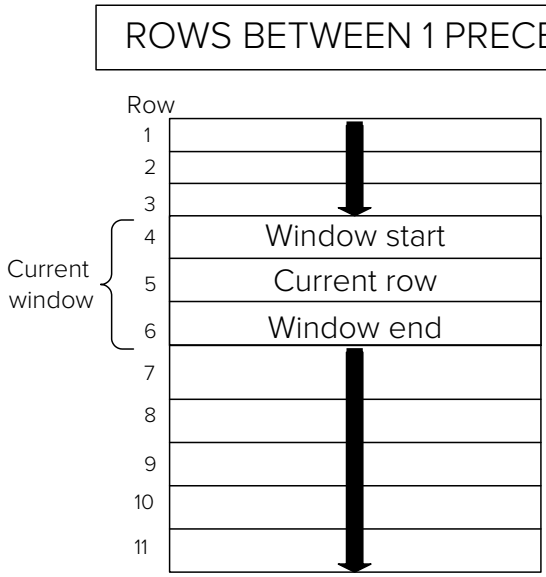
To clarify logical window concepts, Figure 15.23 shows some partial examples. These partial examples show the ordering column and the logical window specification because a logical window specification cannot be understood without knowing the ordering column. Logical windows require additive ordering columns, typically columns with a DATE or INTEGER data type.

- Partial example 1 in Figure 15.23 indicates a cumulative window containing the current row and rows with all previous values of *TimeYear*. `RANGE UNBOUNDED PRECEDING` differs from `ROWS UNBOUNDED PRECEDING` when the current row has the same ordering column value as the next row. For `RANGE UNBOUNDED PRECEDING` the next row is included in the window while the next row is not included for `ROWS UNBOUNDED PRECEDING`.
- Partial example 2 in Figure 15.23 defines a sliding logical window. The window specification, `RANGE 90 PRECEDING`, includes the current row and rows with *HireDate* in the previous 90 days. For columns with a DATE data type, the default interval is days.
- Partial example 3 in Figure 15.23 defines a centered, sliding, logical window. The window specification, `RANGE BETWEEN 365 PRECEDING AND 365 FOLLOWING`, includes the current row, rows within the previous 365 days of *ShipDate* in the current row, and rows within the next 365 days of the current row.
- Partial example 4 in Figure 15.23 uses the INTERVAL keyword to specify a centered, sliding, logical window. The window specification in examples 3 and example 4 differ only for leap years. Interval values must be entered as a text value with an integer inside single quotation marks. The ordering column must have a DATE data type when using the INTERVAL keyword. Intervals can also be specified using the MONTH and DAY keywords.

Figure 15.24 depicts a sliding, physically centered window specified as `ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING`. In Figure 15.24, you should note the window for the fifth row comprising the current row (row 5), the previous row (row 4), and the next row (row 6). The arrows indicate the forward direction of movement for both the start and end of the window. For each window, an aggregate function such as AVG is calculated.

FIGURE 15.23
Partial Examples for Logical Windows

Partial example 1
```
ORDER BY TimeYear RANGE UNBOUNDED PRECEDING
```
Partial example 2
```
ORDER BY HireDate RANGE 90 PRECEDING
```
Partial example 3
```
ORDER BY ShipDate
RANGE BETWEEN 365 PRECEDING AND 365 FOLLOWING
```
Partial example 4
```
ORDER BY ShipDate
RANGE BETWEEN INTERVAL '1' YEAR PRECEDING AND
              INTERVAL '1' YEAR FOLLOWING
```



Window Boundaries

Current Row	Window Start	Window End
1	1	2
2	1	3
3	2	4
4	3	5
5	4	6
6	5	7
7	6	8
8	7	9
9	8	10
10	9	11
11	10	11

FIGURE 15.24
Depiction of a Sliding, Physically Centered Window

As a more complete reference, the Window Boundaries table in Figure 15.24 shows the starting and ending rows of the window for each current row. Note that the windows for first and last rows contain only two rows.

Example 15.26 demonstrates a SELECT statement for a sliding physical window using the AVG function. The SELECT statement calculates the average of the sum of dollar sales by zip code and year over the entire result without partitioning. The window specification, `ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING`, indicates a sliding, physically-centered window.

Example 15.26 (Oracle)

Moving average of sum of sales by zip and year, and month, centered with previous and next row along with partial results

```
SELECT StoreZip, TimeYear, SUM(SalesDollar) AS SumSales,
  AVG(SUM(SalesDollar)) OVER
   (ORDER BY StoreZip, TimeYear
   ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS CenterMovAvgSumSales
 FROM Store, TimeDim, Sales
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
 GROUP BY StoreZip, TimeYear;
```

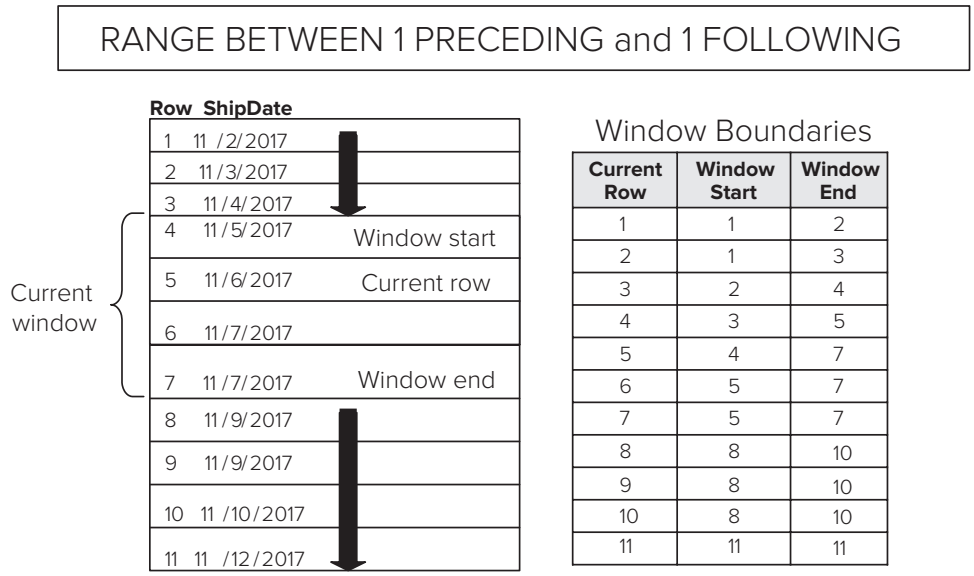
StoreZip	TimeYear	SumSales	CenterMovAvgSumSales
80111-0033	2014	89572	90772.00
80111-0033	2015	91972	74866.67
80111-0033	2016	43056	89422.00
80111-0033	2017	133238	89551.33
80129-5543	2014	92360	106086.00
80129-5543	2015	92260	92593.33

Figure 15.25 depicts a sliding, logically centered window for ship date, specified as `RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING`. Figure 15.25 shows the window for the fifth row with ship date 11/6/2017: the current row (row 5), row 4 with the previous date (11/5/2017), and rows 6 and 7 with the next date (11/7/2017).

As a more complete reference, the Window Boundaries table in Figure 15.25 shows the starting and ending rows in the window for each current row. With duplicate or missing values, a centered, logical window may not be physically centered. For example, the window shown in Figure 15.25 is not physically centered on the current row with starting row 4 and ending row 7. You should study the window boundaries to see other examples of logically centered windows that are not physically centered.

Example 15.27 demonstrates a SELECT statement for a sliding, logically-centered window using the AVG function. The SELECT statement calculates the average of

FIGURE 15.25
Depiction of a Sliding, Logically Centered Window



Window Boundaries

Current Row	Window Start	Window End
1	1	2
2	1	3
3	2	4
4	3	5
5	4	7
6	5	7
7	5	7
8	8	10
9	8	10
10	8	10
11	11	11

Example 15.27 (Oracle)

Moving average of sum of sales by zip, year, and month, centered with previous and next logical rows along with results

```
SELECT TimeYear, SUM(SalesDollar) AS SumSales,
  AVG(SUM(SalesDollar)) OVER
   (ORDER BY TimeYear
   RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS CenterMovAvgSumSales
 FROM Store, TimeDim, Sales
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
 GROUP BY TimeYear ;
```

TimeYear	SumSales	CenterMovAvgSumSales
2014	275648	277198.00
2015	278748	261776.00
2016	230932	276691.33
2017	320394	275663.00

sum of dollar sales by year without partitioning. The window specification, `RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING`, indicates a sliding, logically-centered window.

15.3.4 Query Formulation for Ratio Comparisons

Ratio comparisons, common in business intelligence, provide more precision than relative performance comparisons. Ratios support quantitative evaluation of business entities determining contribution and distribution of measures among entities. Contribution ratios indicate parts of a whole such as the share of total sales for each division. Distribution ratios specify the size of cumulative subsets as compared to the size of a total population such as the threshold for the top 5 percent of unit sales.

Oracle provides three analytic functions for ratio comparisons as summarized in Figure 15.26. The RATIO_TO_REPORT function computes contributions for additive columns such as sales units. The ratios in the result sum to 1 showing the contribution of each part to the whole. The CUME_DIST and PERCENT_RANK functions compute distribution ratios for ordered columns such as unit prices and credit ratings. The largest value of a cumulative distribution function is 1. Oracle provides two functions for distribution ratios differing on the minimum value in the range. The CUME_DIST function does not generate 0 as a result while PERCENT_RANK generates 0. Precise definitions of both functions are given later in this subsection.

Examples begin with the RATIO_TO_REPORT function since it is simpler than the distribution functions. Example 15.28 computes contribution ratios of the sum of dollar sales by year and customer city with contribution ratios restarting on year. For clarity, the result is ordered by year and descending sum or dollar sales. In the SELECT statement, you should note the parameter for the RATIO_TO_REPORT function is the

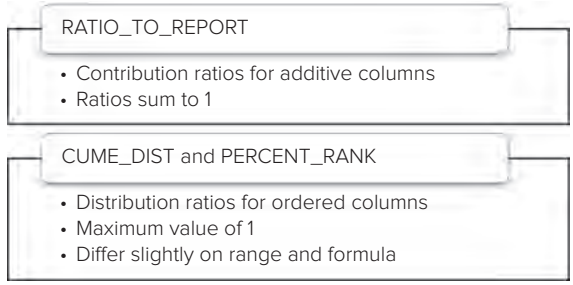


FIGURE 15.26
Summary of Functions for Ratio Comparisons

Example 15.28 (Oracle)

Contribution ratio on sum of dollar sales by year and customer city, partitioned on year, and result ordered by year and descending sum of sales along with partial results

```
SELECT TimeYear, CustCity, SUM(SalesDollar) AS SumSales,
       RATIO_TO_REPORT(SUM(SalesDollar))
         OVER (PARTITION BY TimeYear) AS SumSalesRatio
 FROM Customer, Sales, TimeDim
 WHERE Sales.CustId = Customer.CustId
   AND Sales.TimeNo = TimeDim.TimeNo
 GROUP BY TimeYear, CustCity
 ORDER BY TimeYear, SUM(SalesDollar) DESC;
```

TimeYear	CustCity	SumSales	SumSalesRatio
2014	Littleton	138838	0.5037
2014	Seattle	68032	0.2468
2014	Denver	44380	0.1610
2014	Vancouver	12490	0.0453
2014	Englewood	11908	0.0453
2015	Littleton	141638	0.5081
2015	Seattle	68532	0.2459
2015	Denver	44380	0.1592
2015	Vancouver	12290	0.0441
2015	Englewood	11908	0.0427

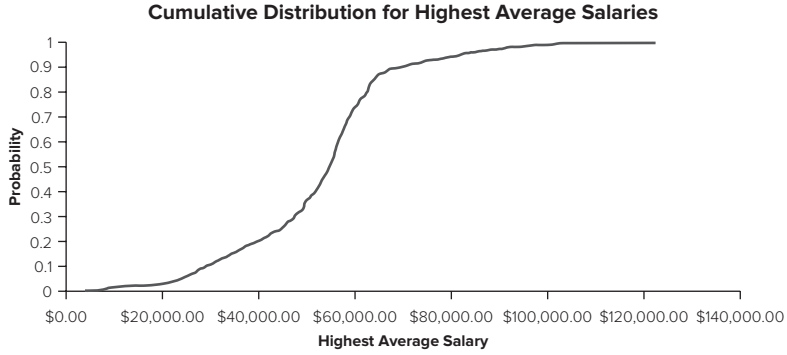
SUM function of *SalesDollar*. Unlike other analytic functions presented in this section, an ordering inside the OVER clause cannot be specified. However, a partitioning can be specified in the PARTITION BY clause.

To understand examples with analytic functions for cumulative distribution, you need some basic probability background. A cumulative distribution determines the cumulative probability for each *x* value, the probability that a random or chance variable is less than or equal to a specified *x* value. Figure 15.27 displays the cumulative distribution for highest average salaries. Starting pension amounts for public employees are based on their highest average salaries. This data comes from several studies about public employee retirees in Colorado from 2001 to 2006. Looking at the graph carefully, you can see that about half of the observations have a highest average salary of less than or equal to $54,711.

The Oracle functions for cumulative distribution differ slightly on formulas for cumulative probabilities and result range as depicted in Figure 15.28. In the numerator, CUME_DIST uses the number of preceding rows inclusive of the specified row, while PERCENT_RANK uses rank minus 1. In the denominator, CUME_DIST uses the total number of rows (*N*), while PERCENT_RANK uses the number of rows minus 1 (*N*-1). For the result range, CUME_DIST generates probabilities greater than 0 to 1, while PERCENT_RANK generates probabilities inclusive between 0 and 1. In the cumulative distribution of highest average salaries (Figure 15.27), CUME_DIST(54,950) is 0.50987 (801/1571), while PERCENT_RANK(54,950) is 0.50955 (800/1570).

Example 15.29 demonstrates the cumulative distribution functions (PERCENT_RANK and CUME_DIST) for item unit prices. To help verify calculations, the result also shows RANK and ROW_NUMBER function values. You can easily compute values for the cumulative distribution functions as the result contains 10 rows.

FIGURE 15.27
Cumulative Distribution Graph



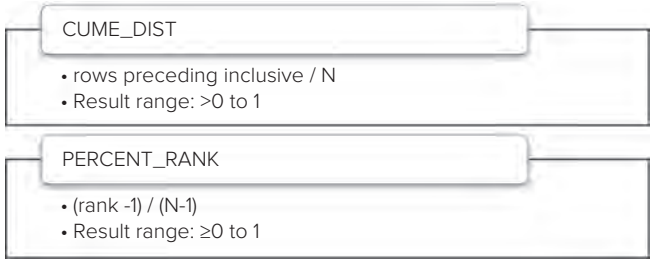


FIGURE 15.28
Summary of Cumulative Distribution Functions

Example 15.29 (Oracle)

Cumulative distribution functions on item unit price along with the result table

```
SELECT ItemName, ItemUnitPrice,
  RANK() OVER (ORDER BY ItemUnitPrice) As RankUP,
  PERCENT_RANK() OVER (ORDER BY ItemUnitPrice) As PerRankUP,
  ROW_NUMBER() OVER (ORDER BY ItemUnitPrice) As RowNumUP,
  CUME_DIST() OVER (ORDER BY ItemUnitPrice) As CumDistUP
 FROM Item;
```

ItemName	ItemUnitPrice	RankUP	PerRankUP	RowNumUP	CumDistUP
Cable	12.00	1	0.000	1	0.100
Surge Protector	14.99	2	0.111	2	0.200
Black Cartridge	25.69	3	0.222	3	0.300
Color Cartridge	38.00	4	0.333	4	0.400
Battery Backup	89.00	5	0.444	5	0.500
CVP Printer	99.00	6	0.555	6	0.600
17 inch Monitor	169.00	7	0.666	7	0.700
Color Scanner	199.99	8	0.777	8	0.800
19 inch Monitor	319.00	9	0.888	9	0.900
Color Laser	699.00	10	1.000	10	1.000

Example 15.30 demonstrates handling of equal values by the cumulative distribution functions. To help verify calculations, the RANK and ROW_NUMBER function values are also shown. The SELECT statement uses the cumulative distribution functions for the sum of unit sales by customer name. You can easily compute values for the cumulative distribution functions as the result contains nine rows.

Example 15.31 demonstrates a more useful statement to retrieve top performers. The SELECT statement uses the CUME_DIST function for item unit price. The result should contain the items in the top 30 percent of the largest unit prices. The SELECT statement uses a nested query in the FROM clause because analytic functions cannot be tested in the WHERE or HAVING clauses. The nested query generates the computed cumulative distribution column so that a WHERE condition can reference it in the outer query. The nested query in the FROM clause is surrounded by parentheses.

Example 15.30 (Oracle)

Demonstrate handling of duplicate values in cumulative distribution functions on sum of sales units grouped by customer name

```
SELECT CustName, SUM(SalesUnits) AS SumSalesUnits,
  RANK() OVER (ORDER BY SUM(SalesUnits) ) AS RankSU,
  PERCENT_RANK() OVER (ORDER BY SUM(SalesUnits) ) AS PerRankSU,
  ROW_NUMBER() OVER (ORDER BY SUM(SalesUnits)) As RowNumSU,
  CUME_DIST() OVER (ORDER BY SUM(SalesUnits) ) AS CumDistSU
 FROM Sales, Customer
 WHERE Sales.CustId = Customer.CustId
 GROUP BY CustName;
```

CustName	SumSalesUnits	RankSU	PerRankSU	RowNumSU	CumDistSU
Beth Taylor	720	1	0.000	1	0.111
Mike Boren	1152	2	0.115	2	0.222
Betty Wise	1272	3	0.250	3	0.444
Larry Styles	1272	3	0.250	4	0.444
Jerry Wyatt	1720	5	0.500	5	0.555
Candy Kendall	1760	6	0.625	6	0.666
Jim Glussman	1960	7	0.750	7	0.777
Wally Jones	2000	8	0.875	8	0.888
Sheri Gordon	11440	9	1.000	9	1.000

Example 15.31 (Oracle)

Demonstrate retrieval of top performers using the CUME_DIST function in a nested query

```
SELECT ItemName, ItemBrand, ItemUnitPrice, CumDistUP
 FROM ( SELECT ItemId, ItemName, ItemBrand, ItemUnitPrice,
          CUME_DIST()
  OVER (ORDER BY ItemUnitPrice DESC) As CumDistUP
  FROM Item )
 WHERE CumDistUnitPrice <= 0.3;
```

ItemName	ItemBrand	ItemUnitPrice	CumDistUP
Color Laser	Connex	699.00	0.100
19 inch Monitor	ColorMeg, Inc.	319.00	0.200
Color Scanner	UV Components	199.99	0.300

You should note that analytic functions for ratio comparisons use measures with numeric columns. The RATIO_TO_REPORT function computes contributions of additive measures to an overall total. The cumulative distribution functions show cumulative probabilities of ordered columns.

15.4 SUMMARY DATA MANAGEMENT AND OPTIMIZATION

To support queries involving large fact tables, relational DBMSs provide materialized views. A **materialized view** is a stored view that must be periodically synchronized with its source data. Materialized views are attractive in data warehouses because fact tables are stable except for periodic refreshments typically performed during nonpeak times. In contrast, traditional (non materialized) views dominate operational database processing because transaction tables can be extremely volatile with high refresh cost. Along with materialized views, most relational DBMSs support automatic substitution of materialized views for source tables in a process known as query rewriting.

Materialized View
a stored view that must be periodically synchronized with its source data. Materialized views support storage of summarized data for fast query response.

The first two parts of this section depict materialized views using the Oracle syntax and provides examples of the query rewriting process. The last part of this section covers storage and optimization technologies, extending coverage in Chapter 8.

15.4.1 Materialized Views in Oracle

As a context for materialized definition and usage, this subsection compares traditional views (covered in Chapter 10) and materialized views. Figure 15.29 provides a convenient comparison. Like a traditional view, a materialized view involves a SELECT statement so it can contain any content of a database, not just row and column subsets. Unlike a traditional view, materialized views provide performance improvement in query intensive environments, not simplification. Materialized views are invisible to users in contrast to direct usage of traditional views. Performance improvements from materialized views involve substantial processing and difficult design choices in contrast to relatively simple processing and design choices for traditional views.

Specification of a materialized view in Oracle involves storage properties, mapping details, and materialization properties unique for materialized views. Because materialized views are stored, most of the storage properties for base tables can also be specified for materialized views. Since storage properties are not the focus here, they will not be depicted. The mapping specification is the same for traditional views as for materialized views. A SELECT statement provides the mapping necessary to populate a materialized view. The following list summarizes materialization properties.

- *Method of refresh (incremental or complete):* Oracle has a number of restrictions on the types of materialized views that can be incrementally refreshed so incremental refreshment is not presented.
- *Refresh timing (on demand or on commit):* For the on demand option, Oracle provides the DBMS_MView package with several refresh procedures (Refresh, Refresh_All_MViews, Refresh_Dependent) to specify refresh timing details.
- *Build timing (immediate or deferred):* For the deferred option, the refresh procedures in the DBMS_MView package can be used to specify the details of populating a materialized view.

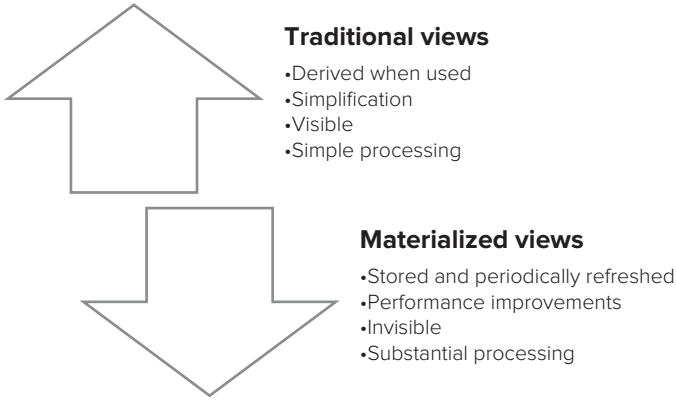


FIGURE 15.29
Summary of Traditional Views and Materialized Views

Examples 15.32 to 15.34 depict the syntax of the CREATE MATERIALIZED VIEW statement. These statements appear similar to CREATE VIEW statements except for the materialization clauses. The build timing is immediate in Examples 15.32 and 15.34, while the build timing is deferred in Example 15.33. The refresh method is complete and the refresh timing is on demand in all three materialized views. The SELECT statement following the AS keyword provides the mapping to populate a materialized view.

Example 15.32 (Oracle)

Materialized view containing sales for all countries for years after 2014 grouped by state and year

```
CREATE MATERIALIZED VIEW MV1
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT StoreState, TimeYear, SUM(SalesDollar) AS SUMDollar1
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND TimeYear > 2014
 GROUP BY StoreState, TimeYear;
```

Example 15.33 (Oracle)

Materialized view containing USA sales in all years grouped by state, year, and month

```
CREATE MATERIALIZED VIEW MV2
BUILD DEFERRED
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT StoreState, TimeYear, TimeMonth,
       SUM(SalesDollar) AS SUMDollar2
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND StoreNation = 'USA'
 GROUP BY StoreState, TimeYear, TimeMonth;
```

Example 15.34 (Oracle)

Materialized view containing Canadian sales before 2015 grouped by city, year, and month

```
CREATE MATERIALIZED VIEW MV3
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE AS
```

```
SELECT StoreCity, TimeYear, TimeMonth,
       SUM(SalesDollar) AS SUMDollar3
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND StoreNation = 'Canada'
   AND TimeYear <= 2014
 GROUP BY StoreCity, TimeYear, TimeMonth;
```

User awareness is an important difference between traditional views and materialized views. For queries using operational databases, traditional views simplify query formulation by replacing base tables in queries. A user perceives a database as a view shielding a user from the complexities of base tables. In contrast, data warehouse queries use fact and dimension tables although traditional views may also hide the complexity of a data warehouse design. In addition, fact and dimension tables may be hidden through a query tool to simplify query formulation. However, data warehouse users are unaware of materialized views as a DBMS uses materialized views internally to improve performance.

Processing of materialized views involves query rewrite, design, and refresh as summarized in Figure 15.30. The query rewriting process substitutes materialized views for base tables to improve performance of queries. The next section covers details of query rewriting. Design involves selection of the best set of materialized views for a given workload. Evaluation of sets of candidate materialized views uses query rewriting and cost estimates from an optimizing SQL compiler. Some enterprise DBMSs provide design tools to help select the best set of materialized views. Refreshing materialized views usually occurs after transformation and loading. The frequency and schedule of refreshing materialized views is part of refresh process management. Some enterprise DBMSs provide tools to manage details about refreshing materialized views.

Query Rewriting
a substitution process in which a materialized view replaces references to fact and dimension tables in a query. An optimizing compiler evaluates whether a substitution will improve performance over the original query without the materialized view substitution.

15.4.2 Query Rewriting Principles

The **query rewriting** process for materialized views reverses the query modification process for traditional views presented in Chapter 10. Recall that the query modification process (Figure 15.31) substitutes base tables for views so that materialization of the views is not needed. In contrast, the query rewriting process (Figure 15.32)

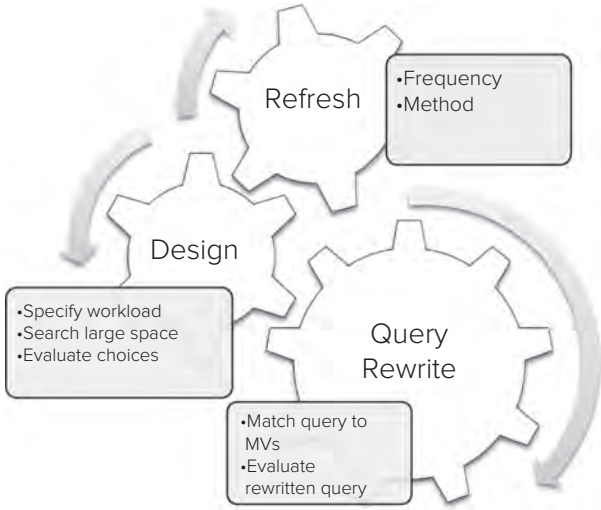


FIGURE 15.30
Summary of Processing for Materialized Views

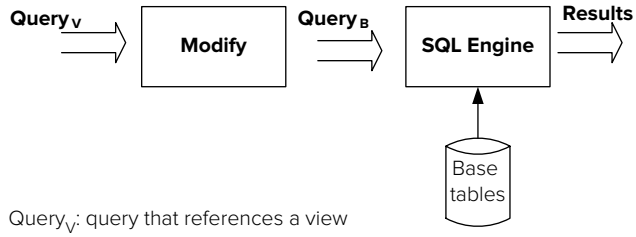


FIGURE 15.31
Process Flow of View Modification

$Query_V$: query that references a view

$Query_B$: modification of $Query_V$ such that references to the view are replaced by references to base tables.

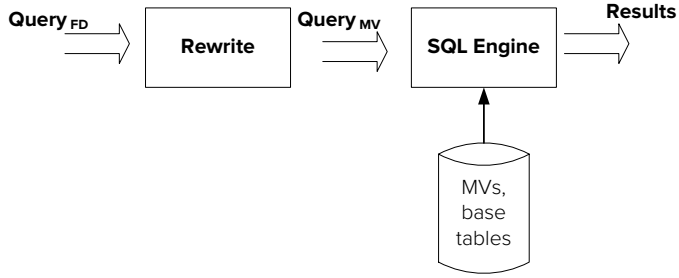


FIGURE 15.32
Process Flow of Query Rewriting

$Query_{FD}$: query that references fact and dimension tables

$Query_{MV}$: rewrite of $Query_{FD}$ such that materialized views are substituted for fact and dimension tables whenever justified by expected performance improvements.

substitutes materialized views for fact and dimension tables to avoid accessing large fact and dimension tables. The substitution process is only performed if performance improvements are expected.

Overall, query rewriting is more complex than query modification because query rewriting involves a more complex substitution process and requires an optimizing compiler to evaluate costs. In both processes, a DBMS, not a user, performs the substitution process. In the query rewriting process, an optimizing compiler evaluates whether the substitution will improve performance over the original query. In the query modification process, an optimizing compiler does not compare the cost of the modified query to the original query because modification usually provides substantial performance improvement.

Query Rewriting Details Query rewriting involves a matching process between a query using fact and dimension tables and a collection of materialized views containing summary data. In brief, a materialized view can provide data for a query if the materialized view matches components of a SELECT statement as explained in the following list and summarized in Table 15-2.

- *Row condition match:* materialized view rows specified in its WHERE clause must contain query rows defined by its WHERE clause. Rewrite is not possible if a materialized view contains more restrictive conditions than a query. Rewrite is possible if the conditions in a query are at least as restrictive as a materialized view. For example, if a materialized view has the conditions StoreNation = 'USA' AND TimeYear = 2016, but a query only has the condition StoreNation = 'USA', the materialized view cannot provide data for the query because the conditions in the materialized view are more restrictive.
- *Grouping match for level of detail:* Grouping columns in a materialized view must contain grouping columns in a query. Rewrite is not possible if a grouping column in a query is not contained in a materialized view. For example, a query

with a grouping on *TimeYear* and *TimeMonth* cannot use a materialized view with a grouping on *TimeYear*. However, a materialized view with grouping on *TimeYear* and *TimeMonth* can be rolled up to provide data for a query with grouping on *TimeYear*.

- *Grouping dependencies:* if a column in a query does not appear in a materialized view, rewrite is still possible if a functional dependency derives the column. Primary keys, candidate keys, and dimension dependencies (via the DETERMINES clause in the Oracle CREATE DIMENSION statement) provide functional dependencies. For example, a query with a grouping on *StoreCity* can be derived from a materialized view with a grouping on *StoreId* because *StoreId* → *StoreCity*. Joins can be used to retrieve columns in a query but not in a materialized view as long as there is a functional relationship (usually a 1-M relationship) connecting the tables.
- *Aggregate match:* aggregates in the query must match available aggregates in the materialized view or be derivable from aggregates in the materialized view. For example, a query containing average is derivable from a materialized view containing sum and count.

Example 15.35 presents an example data warehouse query and the rewritten query to depict the matching process. Table 15-3 depicts matching between *MV1* and the query in Example 15.35. *MV1* and the query match directly on the grouping columns and aggregate computations. The condition on *TimeYear* (> 2014) in *MV1* contains the query condition (2016). In addition, the query contains an extra condition on

Example 15.35

Data warehouse query and rewritten query using the *MV1* materialized view

```
-- Data warehouse query
SELECT StoreState, TimeYear, SUM(SalesDollar)
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND StoreNation IN ('USA','Canada')
   AND TimeYear = 2016
 GROUP BY StoreState, TimeYear;

-- Query Rewrite: replace Sales and TimeDim tables with MV1
SELECT DISTINCT MV1.StoreState, TimeYear, SumDollar1
 FROM MV1, Store
 WHERE MV1.StoreState = Store.StoreState
   AND TimeYear = 2016
   AND StoreNation IN ('USA','Canada');
```

Matching Type	Requirements
Row conditions	Materialized view rows specified by its WHERE clause must contain the query rows defined by its WHERE clause.
Grouping detail	Grouping columns in a materialized view must contain grouping columns in a query or derivable by a functional dependency.
Grouping dependencies	Query columns must match or be derivable by functional dependencies involving materialized view columns.
Aggregate functions	Query aggregate functions must match or be derivable from materialized view aggregate functions.

TABLE 15-2
Summary of Matching Requirements for Query Rewriting

TABLE 15-3
Matching between Materialized View and Example 15.35

	Materialized View	Query
Grouping	StoreState, TimeYear	StoreState, TimeYear
Conditions	TimeYear > 2014	TimeYear = 2016 StoreNation = ('USA', 'Canada')
Aggregates	SUM(SalesDollar)	SUM(SalesDollar)

StoreNation. The materialized view result does not need to contain *StoreNation* because *StoreState* → *StoreNation*. Grouping is not necessary in the rewritten query because identical grouping is already performed in *MV1*. The DISTINCT keyword removes duplicate state values in the result because the *Store* table has multiple rows with the same state value.

Example 15.36 presents a more complex example of query rewriting involving three SELECT blocks combined using the UNION operator. Table 15-4 depicts matching between the materialized views (*MV1, MV2, MV3*) and the query in Example 15.36. The first query block retrieves the total sales in the United States or Canada from 2014 to 2017. The second query block retrieves the USA store sales in 2014. The third query block retrieves Canadian store sales in 2014. The GROUP BY clauses are necessary in the second and third query blocks to roll-up the finer level of detail in the materialized views. In the third query block, the condition on *StoreNation* is needed because some cities have identical names in both countries. The materialized view results do not need to contain StoreNation because *StoreState* → *StoreNation*. The DISTINCT keyword in the first and third query blocks removes duplicate state values in the result because the Store table has multiple rows with the same state value.

Example 15.36

Data Warehouse Query and Rewritten Query using the *MV1, MV2,* and *MV3* Materialized Views

```
-- Data warehouse query
SELECT StoreState, TimeYear, SUM(SalesDollar)
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND StoreNation IN ('USA','Canada')
   AND TimeYear BETWEEN 2014 and 2017
 GROUP BY StoreState, TimeYear;

-- Query Rewrite
SELECT DISTINCT MV1.StoreState, TimeYear, SumDollar1 AS StoreSales
 FROM MV1, Store
 WHERE MV1.StoreState = Store.StoreState
   AND TimeYear <= 2017 AND StoreNation IN ('USA','Canada')
UNION
SELECT StoreState, TimeYear, SUM(SumDollar2) as StoreSales
 FROM MV2
 WHERE TimeYear = 2014
 GROUP BY StoreState, TimeYear
UNION
SELECT DISTINCT StoreState, TimeYear, SUM(SumDollar3) as StoreSales
 FROM MV3, Store
 WHERE MV3.StoreCity = Store.StoreCity
   AND TimeYear = 2014 AND StoreNation = 'Canada'
 GROUP BY StoreState, TimeYear;
```

	MV1	MV2	MV3	Query
Grouping	StoreState, TimeYear	StoreState, TimeMonth, TimeYear	StoreCity, TimeMonth, TimeYear	StoreState, TimeYear
Conditions	TimeYear > 2014	StoreNation = 'USA'	TimeYear <= 2014 StoreNation = 'Canada'	TimeYear BETWEEN 2014 AND 2017 StoreNation IN ('USA', 'Canada')
Aggregates	SUM(SalesDollar)	SUM(SalesDollar)	SUM(SalesDollar)	SUM(SalesDollar)

TABLE 15-4
Matching between Materialized Views and Example 15.36

Example 15.37 extends Example 15.36 with a CUBE operator. In the rewritten query, the CUBE is performed one time at the end rather than in each SELECT block. To perform the CUBE one time, the FROM clause should contain a nested query. An alternative to the nested query in the FROM clause is to place the nested query in a separate CREATE VIEW statement.

Example 15.37

Data warehouse query and rewritten query using *MV1, MV2,* and *MV3*

```
-- Data warehouse query
SELECT StoreState, TimeYear, SUM(SalesDollar)
 FROM Sales, Store, TimeDim
 WHERE Sales.StoreId = Store.StoreId
   AND Sales.TimeNo = TimeDim.TimeNo
   AND StoreNation IN ('USA','Canada')
   AND TimeYear BETWEEN 2014 and 2017
 GROUP BY CUBE(StoreState, TimeYear);

-- Query Rewrite
SELECT StoreState, TimeYear, SUM(StoreSales) as SumStoreSales
 FROM (
  SELECT DISTINCT MV1.StoreState, TimeYear, SumDollar1 AS StoreSales
   FROM MV1, Store
   WHERE MV1.StoreState = Store.StoreState
     AND TimeYear <= 2017
     AND StoreNation IN ('USA','Canada')
 UNION
  SELECT StoreState, TimeYear, SUM(SumDollar2) as StoreSales
   FROM MV2
   WHERE TimeYear = 2014
   GROUP BY StoreState, TimeYear
 UNION
  SELECT DISTINCT StoreState, TimeYear, SUM(SumDollar3) as StoreSales
   FROM MV3, Store
   WHERE MV3.StoreCity = Store.StoreCity
     AND TimeYear = 2014 AND StoreNation = 'Canada'
   GROUP BY StoreState, TimeYear )
GROUP BY CUBE(StoreState, TimeYear);
```

These examples indicate the range of query rewriting possibilities rather than capabilities of actual DBMSs. Most enterprise DBMSs support query rewriting, but the range of query rewriting support varies. Because of the complexity and the proprietary nature of query rewriting, the details of query rewriting algorithms are beyond the scope of this textbook.

15.4.3 Storage and Optimization Technologies

Several storage technologies have been developed to support multidimensional data capabilities in OLAP servers and enterprise DBMSs. In addition to storage technologies, DBMS vendors have developed data warehouse appliances, complete hardware and software solutions for deploying enterprise level data warehouses. This section describes features of storage technologies along with details about vendor offerings of data warehouse appliances.

MOLAP (Multidimensional OLAP) Originally, vendors of business intelligence software developed a storage architecture that directly manipulates data cubes. This storage architecture, known as **MOLAP** for Multidimensional OLAP, was the only choice as a storage technology for data warehouses until the mid-1990s. At the current time, MOLAP has been eclipsed as the primary storage architecture for data warehouses, but it still is an important technology for summary data cubes, small data warehouses, and data marts.

MOLAP
a storage engine that directly stores and manipulates data cubes. MOLAP engines generally offer the best query performance but place limits on the size of data cubes.

MOLAP storage engines directly manipulate stored data cubes as depicted in Figure 15.33. The storage engines of MOLAP systems are optimized for the unique characteristics of multidimensional data such as sparsity and complex aggregation across thousands of cells. Because data cubes are precomputed, MOLAP query performance is generally better than competing approaches that use relational database storage. Even with techniques to deal with sparsity, MOLAP engines can be overwhelmed by the size of data cubes. A fully calculated data cube may expand many times as compared to the raw input data. This data explosion problem limits the size of data cubes that MOLAP engines can manipulate.

ROLAP (Relational OLAP) Because of potential market size and growth of data warehouse processing, vendors of relational DBMSs have extended their products with additional features to support operations and storage structures for multidimensional data. These product extensions are collectively known as **ROLAP** for Relational OLAP. Given the growing size of data warehouses and the intensive research and development by relational DBMS vendors, it was only a matter of time before ROLAP became the dominant storage engine for data warehouses.

ROLAP
relational DBMS extensions to support multidimensional data. ROLAP engines support a variety of storage and optimization techniques for summary data retrieval.

In the ROLAP approach, relational databases store multidimensional data using the star schema or its variations as depicted in Figure 15.33. Data cubes are dynamically constructed from fact and dimension tables as well as from materialized views. Typically, only a subset of a data cube must be constructed as specified in a user's query. Extensions to SQL as described in section 15.2 allow users to manipulate the dimensions and measures in virtual data cubes.

ROLAP engines incorporate a variety of storage and optimization techniques for summary data retrieval. This list explains the most prominent techniques:

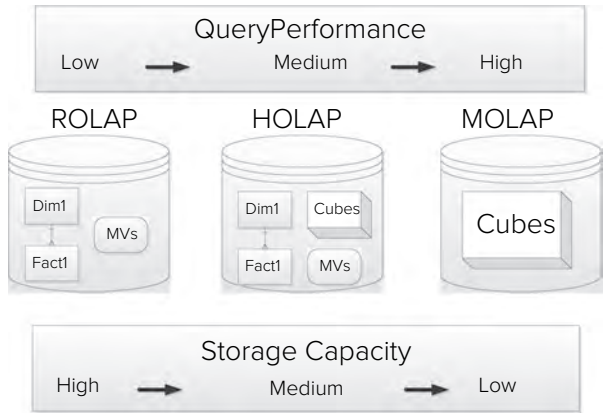


FIGURE 15.33
Summary of Storage Architectures for Data Warehouses

- *Bitmap join indexes* (see Chapter 8 for details) are particularly useful for columns in dimension tables with few values such as *CustState*. A bitmap join index provides a precomputed join from the values of a dimension table to the rows of a fact table. To support snowflake schemas, some DBMS vendors support bitmap join indexes for dimension tables related to other dimension tables. For example, a bitmap index for the *Division.DivManager* column has an index record that contains a bitmap for related rows of the *Store* table and a second bitmap for rows of the *Sales* table related to matching rows of the *Store* table.
- *Columnstore indexes* (see Chapter 8 for details) provide substantial performance improvements for grouping queries performing calculations on relatively few columns in large fact tables with millions of rows.
- *Star join query optimization* uses bitmap join indexes on dimension tables to reduce the number of rows in a fact table to retrieve. A star join involves a fact table joined with one or more dimension tables. Star join optimization involves three phases. In the first phase, the bitmap join indexes on each dimension table are combined using the union operator for conditions connected by the OR operator and the intersection operator for conditions connected by the AND operator. In the second phase, the bitmaps resulting from the first phase are combined using the intersection operator. In the third phase, the rows of the fact table are retrieved using the bitmap resulting from the second phase. Star join optimization can result in substantially reduced execution time as compared to traditional join algorithms that combine two tables at a time.
- *Query rewriting using materialized views* can eliminate the need to access large fact and dimension tables. If materialized views are large, they can be indexed to improve retrieval performance. Query rewriting uses an optimizing compiler to evaluate the benefit of using materialized views as compared to the fact and dimension tables.
- *Summary storage advisors* determine the best set of materialized views that should be created and maintained for a given query workload. For consistency with other components, the summary advisor is integrated with the query rewriting component and optimizing query compiler.
- *Partitioning, striping, and parallel query execution* provide opportunities to reduce the execution time of data warehouse queries. The choices must be carefully studied so that the use of partitioning and striping support the desired level of parallel query execution.

Despite intensive research and development on ROLAP storage and optimization techniques, MOLAP engines still provide faster query response time. However, MOLAP storage suffers from limitations in data cube size so that ROLAP storage is necessary for fine-grained data warehouses. In addition, the difference in response time has narrowed so that ROLAP storage may involve only a slight performance penalty if ROLAP storage and optimization techniques are properly utilized.

HOLAP (Hybrid OLAP) Because of the tradeoffs between MOLAP and ROLAP, a third technology known as **HOLAP** for Hybrid OLAP has been developed to combine ROLAP and MOLAP. HOLAP supports division of a data warehouse between relational storage of fact and dimension tables and multidimensional storage of summary data cubes as depicted in Figure 15.33. When an OLAP query is submitted, the HOLAP system can combine data from the ROLAP managed data and the MOLAP managed data.

HOLAP
a storage engine for data warehouses that combines ROLAP and MOLAP storage engines. HOLAP involves both relational and multidimensional data storage as well as combining data from both relational and multidimensional sources for data cube operations.

Despite the appeal of HOLAP, it has potential disadvantages that may limit its use. First, HOLAP can be more complex than either ROLAP or MOLAP, especially if a DBMS vendor does not provide full HOLAP support. To fully support HOLAP, a DBMS vendor must provide both MOLAP and ROLAP engines as well as tools to combine both storage engines in the design and operation of a data warehouse. Second, there is considerable overlap in functionality between the storage and optimization techniques in ROLAP and MOLAP engines. It is not clear whether the ROLAP storage

and optimization techniques should be discarded or used in addition to the MOLAP techniques. Third, because the difference in response time has narrowed between ROLAP and MOLAP, the combination of MOLAP and ROLAP may not provide significant performance improvement to justify the added complexity.

Data Warehouse Appliances and Cloud Solutions Data warehouse appliances provide prepackaged solutions for operating a data warehouse using various storage technologies and optimization methods. A **data warehouse appliance** is a combination of hardware and software components for rapid deployment and transparent operation of data warehouses. The components typically include an operating system, a DBMS, server hardware, and storage devices. Early appliances emphasized proprietary components but the trend now is for more open and industry standard components especially commodity hardware and open source operating systems.

Data Warehouse Appliance a prepackaged solution for operating a data warehouse using various storage technologies and optimization methods. A data warehouse appliance is a combination of hardware and software components for rapid deployment and transparent operation of data warehouses.

Data warehouse appliances offer the promise of increased performance, reduced maintenance costs, and improved scalability. Most appliance products provide parallel database processing (see Chapter 18) and other performance enhancements to improve both query and refresh processing. Performance improvement is partially due to dedicated components and tuning for data warehouse processing without compromise for other types of processing. Vendors have designed data warehouse appliances for rapid deployment and transparent operation. Organizations have reported lower staffing costs for data warehouse administrators due to less performance tuning and monitoring. The performance and maintenance benefits are especially important as data warehouse processing loads increase. Vendors have designed data warehouse appliances for scalability with relatively easy addition of components as loads increase.

Data warehouse appliances lack flexibility. An organization purchasing a data warehouse appliance commits to a proprietary solution with limited flexibility for design changes. Using a data warehouse appliance often means redeploying a data warehouse to a new environment with potential migration costs.

The market for data warehouse appliances has stagnated due to emphasis on flexibility and customized solutions. Major DBMS vendors continue to offer data warehouse appliances as shown in Table 15-5, but innovation in these products has slowed. These offerings are becoming niche products for organizations willing to sacrifice flexibility for turnkey solutions.

Cloud base solutions for data warehouse deployment provide flexibility lacking in data warehouse appliances. Major DBMS vendors, major web service providers (Google and Amazon), and specialized vendors provide cloud solutions for data warehouse deployment. Cloud solutions differ widely from external hosting of a vendor's data warehouse products to complete new solutions only available in a cloud service. For example, IBM, Microsoft, Oracle, and Teradata manage cloud-enabled versions of their base products. Amazon, Google, and Snowflake offer new data warehouse solutions designed only for cloud deployment. The market seems poised for growth of flexible, cloud solutions compared to rigid data warehouse appliances. However, traditional local hosting of data warehouse solutions should continue to dominate for many years over cloud solutions and appliances.

TABLE 15-5
Major Offerings of Data Warehouse Appliances

Company	Products	Notes
Microsoft	Parallel Data Warehouse, Fast Track Data Warehouse	Alliances with HP and Dell
IBM	PureData System	Variety of product offerings for a range of data warehouse and business analytic needs
TeraData	Teradata Data Warehouse Appliance 2800	Alliance with Intel
Oracle	Oracle Exadata Intelligent Warehouse Solution	Variety of configurations for both data warehouse and transaction processing

CLOSING THOUGHTS

This chapter extended the conceptual background and skills foundation provided in Chapters 12 to 14 with detailed coverage of query formulation and summary data management. Online Analytic Processing (OLAP) for multidimensional databases occupies an important niche in many organizations, providing the foundation of query formulation by business analysts. The first part of this chapter covered Microsoft Multidimensional Expressions (MDX), the de facto standard for OLAP databases along with pivot table tools providing a convenient interface for MDX data cubes. The MDX coverage in this chapter extended conceptual material about multidimensional data representation in Chapter 13.1.

Because relational DBMSs provide efficient management and powerful query formulation for large databases, relational DBMSs provide the foundation for most enterprise data warehouses. For querying data warehouses implemented as relational databases, extensions to the SQL SELECT statement were specified in the SQL standard and implemented by major DBMS vendors. The second part of this chapter presented extensions to the GROUP BY clause for subtotal calculations, important when summarizing multidimensional data. Details about the CUBE, ROLLUP, and GROUPING SETS operators were shown to extend query formulation skills for multidimensional data in star schemas. The third part of this chapter presented SQL extensions for common types of business intelligence applications. These analytic function extensions involve an additional processing step for SELECT statements and syntax revisions in the SQL SELECT statement. The third part of this chapter covered analytic functions for relative performance of business units, trend analysis, and ratio comparisons.

DBMS vendors have made substantial product extensions for efficient management of summary data and new optimization technologies for data warehouse queries. The last part of this chapter presented materialized views, a fundamental tool for efficient management of summary data. Details about defining materialized views and the query rewriting process to use materialized views to improve retrieval performance of summary data were provided. In addition, the last part of this chapter presented storage technologies for data warehouses using both relational and data cube storage engines. Together, background on materialized views and storage technologies equip a student to understand basic issues with efficient implementation of a data warehouse.

REVIEW CONCEPTS

- Microsoft Multidimensional Expressions (MDX) providing a de facto standard for Online Analytic Processing (OLAP)
- MDX SELECT statement for queries on data cubes
- Pivot table tools providing a convenient interface for manipulating MDX data cubes
- Pivot4J plugin using a built-in OLAP server and MDX statement generation
- WebPivotTable using an external OLAP server and MDX statement generation
- Extensions of the GROUP BY clause for subtotal calculation: CUBE, ROLLUP, and GROUPING SETS operators
- CUBE operator for generating all possible combinations of subtotals
- ROLLUP operator for generating subtotals for columns related in a dimension hierarchy
- GROUPING SETS operator for precise control of subtotals
- Equivalence of GROUP BY operators to a collection of SELECT blocks connected by UNION operations

- Variations of subtotal operators for partial cube and rollup operations, composite columns, nested subtotal operations, and subtotal identifiers
- Analytic function extensions to support relative performance of business units, trend analysis, and ratio comparisons
- Factors influencing analytic function extensions: difficult to write, poor productivity, and poor performance
- Additional step in SELECT statement processing for analytic functions
- Basic syntax for analytic functions with OVER keyword and ordering specification
- Extended syntax for partitioning in analytic functions with PARTITION BY keywords
- Analytic functions for relative performance calculations: RANK, DENSE_RANK, ROW_NUMBER, and NTILE
- RANK function with gaps for duplicate values and DENSE_RANK function without gaps for duplicate values
- Window concepts for queries involving trend analysis: units (physical and logical) and movement (sliding and cumulative)
- Extended syntax for window specifications with the ROWS and RANGE keywords
- Analytic functions for ratio comparisons: RATIO_TO_REPORT, CUME_DIST, and PERCENT_RANK
- Cumulative distribution, an important concept for quantitative comparisons of business units
- Differences between CUME_DIST and PERCENT_RANK on range of result and formulas for computing cumulative probabilities
- Materialized views for storage of precomputed summary data
- CREATE MATERIALIZED VIEW statement involving method of refresh, refresh timing, build timing, storage properties, and a SELECT statement
- Process flow for query rewriting (materialized views) versus process flow for view modification (traditional views)
- Query rewriting involving substitution of materialized views for fact and dimension tables to improve performance of data warehouse queries
- Matching requirements for query rewriting involving row conditions, grouping detail, grouping dependencies, and aggregate functions
- Data warehouse storage architectures: ROLAP, MOLAP, and HOLAP
- Star join optimization using bitmap indexes on dimension tables to reduce the number of rows in a fact table to retrieve
- Columnstore indexes for improved performance of grouping queries on large fact tables
- Data warehouse appliances providing combinations of hardware and software components for rapid deployment and transparent operation of data warehouses

QUESTIONS

1. Briefly provide a brief history of Microsoft Multidimensional Expressions (MDX).
2. What commercial products and open source projects use MDX?
3. What is an attribute in MDX cube representation?

4. What is a member in MDX cube representation?
5. How are measures shown in an MDX cube?
6. What is a tuple in MDX cube representation? How is a tuple related to a slicer?
7. What is an axis in an MDX query?
8. Compare the MDX SELECT statement to the SQL SELECT statement.
9. What relational algebra operator (see Chapter 3) is equivalent to the MDX cross join operator?
10. What is a slicer condition? What is the relationship between dimensions on an axis and dimensions in a slicer condition?
11. Why is MDX not used directly by business analysts?
12. What is the difference between the Pivot4J plugin in Pentaho Business Analytics and WebPivotTable for usage of an OLAP server?
13. What is the relationship between MDX and pivot table tools such as the Pivot4J plugin and WebPivotTable?
14. What limitation in the GROUP BY clause is addressed by the CUBE, ROLLUP, and GROUPING SETS operators?
15. How are missing values shown in a data cube versus in a result of an SQL SELECT statement with a GROUP BY clause?
16. What is the purpose of the CUBE operator?
17. Briefly explain derivation of the CUBE operator using the UNION operator.
18. What is the purpose of the ROLLUP operator?
19. Briefly explain derivation of the ROLLUP operator using the UNION operator.
20. What is the purpose of the GROUPING SETS operator?
21. What is a partial CUBE?
22. What is a partial ROLLUP?
23. How are composite columns used in a CUBE or ROLLUP operation?
24. How does a partial CUBE or ROLLUP differ from a nested CUBE or ROLLUP?
25. What is the GROUPING_ID function?
26. Briefly explain common business intelligence applications supported by analytic function extensions in the SQL SELECT statement.
27. What factors influenced extensions to the SQL SELECT statement for analytic functions?
28. Briefly describe extensions in SELECT statement processing for analytic function extensions.
29. Briefly explain subtle points in the revised processing model for analytic functions.
30. Briefly explain the basic syntax for analytic functions without partitioning.
31. Briefly explain the extended syntax for analytic functions with partitioning.
32. What is the difference between the RANK and DENSE_RANK functions?
33. What is the role of windows for trend analysis queries?
34. Briefly explain window concepts in the SQL SELECT statement.
35. Briefly explain three examples of physical windows in an SQL window specification.
36. What is the difference between a cumulative window and a sliding window?
37. What types of ratio comparisons are supported in extensions to the SQL SELECT statement?
38. What is the RATIO_TO_REPORT function?

39. What is a cumulative distribution?
40. What are the CUME_DIST and PERCENT_RANK functions?
41. What are differences between the CUME_DIST and PERCENT_RANK functions?
42. In Example 15.30, show the computation in the CUME_DIST function for sum of sales unit with value of 1272.
43. In Example 15.30, show the computation in the PERCENT_RANK function for sum of sales unit with value of 1272.
44. Why are materialized views important for data warehouses but typically not important for operational databases?
45. What materialization properties does Oracle provide for materialized views?
46. Compare and contrast query rewriting for materialized views to query modification for traditional (nonmaterialized) views.
47. Briefly explain the matching process in query rewriting.
48. Explain the importance of indexing fact and dimension tables in a data warehouse.
49. What are the pros and cons of a MOLAP storage engine?
50. What are the pros and cons of a ROLAP storage engine?
51. What are the pros and cons of a HOLAP storage engine?
52. List some storage and optimization techniques in ROLAP engines.
53. What is star join optimization?
54. What are the advantages of data warehouse appliances?
55. Why has development and usage of data warehouse appliances stagnated?

PROBLEMS

The problems provide practice with relational database manipulation of multidimensional data using the store sales snowflake schema shown in Section 15.2. For your reference, Figure 15.P1 displays the ERD for the store sales snowflake schema. To support the usage of Oracle with these problems, the textbook's website contains Oracle CREATE TABLE statements and sample data for the tables of the store sales schema. In the Oracle statements, the tables have been prefixed with "SS" to avoid conflicts with other table names in the same schema.

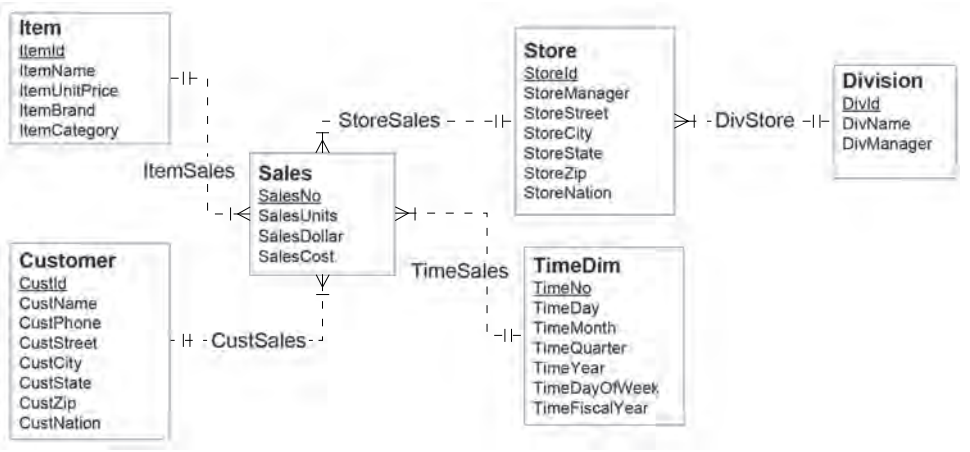


FIGURE 15.P1
ERD Snowflake Schema for the Store Sales Example

1. Write a SELECT statement to summarize sales by store state, year, and item brand. The result should compute the SUM of the dollar sales for the years 2016 and 2017. The result should include full totals for every combination of grouped fields.
2. Write a SELECT statement to summarize sales by year, quarter, and month. The result should compute the SUM of the dollar sales for the years 2016 and 2017. The result should include partial totals in order of the grouped fields (year, quarter, and month).
3. Write a SELECT statement to summarize sales by store state, month, and year. The result should compute the SUM of the dollar sales for the years 2016 and 2017. The result should include partial totals in order of the hierarchical dimension (year and month). Do not use the GROUPING SETS operator in your SQL statement.
4. Write a SELECT statement to summarize sales by customer state, customer zip, year, and quarter. The result should compute the SUM of the dollar sales for the years 2016 and 2017. The result should include partial totals for hierarchical dimensions (year/quarter and state/zip). Do not use the GROUPING SETS operator in your SQL statement.
5. Rewrite the SQL statement solution for problem 1 without using the CUBE, ROLLUP, or GROUPING SETS operators. To reduce your time, you can write the first few query blocks and then indicate the pattern to rewrite the remaining query blocks. In rewriting the queries, you may use two single quotes (with nothing inside) as the default text value and 0 as the default numeric value.
6. Rewrite the SQL statement solution for problem 2 without using the CUBE, ROLLUP, or GROUPING SETS operators. In rewriting the queries, you may use two single quotes (with nothing inside) as the default text value and 0 as the default numeric value.
7. Rewrite the SQL statement solution for problem 3 without using the CUBE, ROLLUP, or GROUPING SETS operators. In rewriting the queries, you can use two single quotes (with nothing inside) as the default text value and 0 as the default numeric value.
8. Rewrite the SQL statement solution for problem 3 using the GROUPING SETS operator instead of the ROLLUP operator.
9. Rewrite the SQL statement solution for problem 4 using the GROUPING SETS operator instead of the ROLLUP operator.
10. Perform the indicated calculation and show the underlying formula for the following problems. The number of unique values in each dimension is shown in parentheses.
    - Calculate the maximum number of rows for a query with a rollup of year (2), quarter (4), and month (12). Separate the calculation to show the number of rows appearing in the normal GROUP BY result and the number of subtotal rows generated by the ROLLUP operator.
    - Calculate the maximum number of rows in a query with a rollup of year (2), quarter (4), month (12), and weeks per month (4). Separate the calculation to show the maximum number of rows appearing in the normal GROUP BY result and the maximum number of subtotal rows generated by the rollup operator.
    - Calculate the maximum number of rows in a query with a cube of state (5), brands (10), and year (2). Separate the calculation to show the maximum number of rows appearing in the normal GROUP BY result and the maximum number of subtotal rows generated by the cube operator.
    - Calculate the number of subtotal groups in a query with a cube of state (5), division (4), brand (10), and year (2). A subtotal group is equivalent to a

SELECT statement when formulating the query without any GROUP BY operators.

11. Write an Oracle CREATE MATERIALIZED VIEW statement to support the store sales schema. The materialized view should include the sum of the dollar sales and the sum of the cost of sales. The materialized view should summarize the measures by the store identifier, the item identifier, and the time number. The materialized view should include sales in the year 2016.
12. Write an Oracle CREATE MATERIALIZED VIEW statement to support the store sales schema. The materialized view should include the sum of the dollar sales and the sum of the cost of sales. The materialized view should summarize the measures by the store identifier, the item identifier, and the time number. The materialized view should include sales in the year 2017.
13. Rewrite the SQL statement solution for problem 1 using the materialized views in problems 11 and 12. You should ignore the CUBE operator in the solution for problem 1. Your SELECT statement should reference the materialized views as well as base tables if needed.
14. Rewrite the SQL statement solution for problem 1 using the materialized views in problems 11 and 12. Your SELECT statement should reference the materialized views as well as base tables if needed. You should think carefully about how to handle the CUBE operator in your rewritten query.
15. Rewrite the SQL statement solution for problem 3 using the materialized views in problems 11 and 12. You should ignore the ROLLUP operator in the solution for problem 3. Your SELECT statement should reference the materialized views as well as base tables if needed.
16. Rewrite the SQL statement solution for problem 3 using the materialized views in problems 11 and 12. Your SELECT statement should reference the materialized views as well as base tables if needed. You should think carefully about how to handle the ROLLUP operator in your rewritten query.
17. Write an Oracle CREATE MATERIALIZED VIEW statement using the store sales schema. The materialized view should include the sum of sales units and the sum of the cost of sales. The materialized view should summarize the measures by the customer zip code and year of sales. The materialized view should include sales in 2016 and before.
18. Write an Oracle CREATE MATERIALIZED VIEW statement using the store sales schema. The materialized view should include the sum of sales units and the sum of the cost of sales. The materialized view should summarize the measures by the customer zip code, sales year, and sales quarter. The materialized view should include USA sales only.
19. Write an Oracle CREATE MATERIALIZED VIEW statement using the store sales schema. The materialized view should include the sum of the sales units and sum of the cost of sales. The materialized view should summarize the measures by the customer zip code, sales year, and sales quarter. The materialized view should include Canadian sales for 2016 and 2017.
20. Write a SELECT statement using the base data warehouse tables to retrieve the sum of the sales cost divided by the sum of the unit sales in the USA and Canada in 2016. The result should include customer zip code, year, and sum of the sales cost per unit. Rewrite the SELECT statement using one or more materialized views defined in problems 17 to 19.
21. Write a SELECT statement using the base data warehouse tables to retrieve the sum of the sales cost divided by the sum of the unit sales in the USA and Canada from 2015 to 2017. The result should include customer zip code, year, and sum of the sales cost per unit. Rewrite the SELECT statement using one or more materialized views defined in problems 17 to 19.

22. Identify the subtotals generated with the following GROUP BY clause: `GROUP BY TimeYear, CUBE(ItemCategory, StoreZip)`.
23. Identify the subtotals generated with the following GROUP BY clause: `GROUP BY TimeYear, TimeQuarter, CUBE(ItemCategory, StoreZip)`.
24. Identify the subtotals generated with the following GROUP BY clause: `GROUP BY ItemCategory, ROLLUP(TimeYear, TimeQuarter, TimeMonth)`.
25. Identify the subtotals generated with the following GROUP BY clause: `GROUP BY ItemCategory, StoreZip, ROLLUP(TimeYear, TimeQuarter, TimeMonth)`.
26. Identify the subtotals generated with the following GROUP BY clause: `GROUP BY CUBE(ItemCategory, StoreZip), ROLLUP(TimeYear, TimeQuarter, TimeMonth)`.
27. Identify the subtotals generated with the following GROUP BY clause: `GROUP BY ROLLUP(ItemCategory, ItemBrand), ROLLUP(TimeYear, TimeQuarter, TimeMonth)`.
28. Identify the subtotals generated with the following GROUP BY clause: `GROUP BY ROLLUP(StoreNation, StoreState, (StoreZip, StoreCity))`.
29. Identify the subtotals generated with the following GROUP BY clause: `GROUP BY GROUPING SETS(ItemCategory, ROLLUP(TimeYear, TimeQuarter), StoreZip)`.
30. Identify the subtotals generated with the following GROUP BY clause: `GROUP BY GROUPING SETS(TimeYear, StoreZip, CUBE(ItemBrand, ItemCategory), CustZip)`.
31. Write a SELECT statement to rank item brands by descending average dollar sales in 2014 to 2015. The result should display item brand, average dollar sales, and rank. The result should contain a single ranking for all item brands.
32. Write a SELECT statement to rank item brands by descending average dollar sales in 2014 to 2015. The result should only contain brands with more than 10 sales in 2014 to 2015. The result should display item brand, count of sales, average dollar sales, and rank. The result should contain a single ranking for all item brands.
33. Write a SELECT statement to dense rank item brands by descending count of sales rows. The result should contain a separate ranking for each year. The result should only contain brands with more than 5 sales in a year. The result should display item brand, year, count of rows, and dense rank.
34. Write a SELECT statement to rank and dense rank item brands by descending sum of dollar sales for sales in 2014. The result should contain a separate ranking for each month. The result should display item brand, month, sum of sales, rank, and dense rank.
35. Write a SELECT statement to calculate the cumulative sum of 2014 sales by item brand and month. The result should contain a separate cumulative sum of sales for each item brand. The result should contain item brand, month, sum of sales, and cumulative sum of sales.
36. Write a SELECT statement to calculate the cumulative sum of sales by year and item brand. The result should contain a separate cumulative sum of sales for each year. The result should only contain brands with more than 5 sales in a year. The result should display year, item brand, count of rows, sum of sales, and cumulative sum of sales.
37. Write a SELECT statement to calculate the moving average of sum of sales by year and item brand. The result should contain a separate moving average for each year. The window should be centered on the 2 preceding and 2 following rows. The result should only contain brands with more than 5 sales in a year.

The result should display year, item brand, count of rows, sum of sales, and average sum of sales.

38. Write a SELECT statement to calculate the moving average of sum of 2014 sales by month. The moving average should be centered on the 3 preceding and 3 following months. The result should display month, sum of sales, and average sum of sales.
39. Write a SELECT statement to calculate a moving average of sum of dollar sales by store zip and sales date. The result should contain a separate moving average for each store zip. The moving average should be centered on the 3 previous and 3 following months. The result should display store zip, sales date, sum of sales, and average sum of sales. In the statement, you need to combine *TimeDim* columns into a complete sales date. You can use this expression: `to_date(to_char(TimeDay, 'FM00') || to_char(TimeMonth, 'FM00') || to_char(TimeYear), 'DDMMYYYY') )`.
40. Write a SELECT statement to compute the cumulative distribution (CUME_DIST) of dollar sales in Colorado (CO). The result should not contain duplicates. The result should display dollar sales and cumulative distribution of dollar sales.
41. Write a SELECT statement to display the top performing customer zip codes (30%) by year on sum of dollar sales. You can use either cumulative distribution function. The result should contain separate top performing zip codes for each year. The result should display year, customer zip code, sum of dollar sales, and cumulative distribution. Order the result by year and cumulative distribution.
42. Write a SELECT statement to compute the contribution ratio on the sum of 2015 sales by month and item brand. The result should contain a separate contribution ratio for each month. The result should display month, item brand, sum of unit sales, and contribution ratio. The result should be ordered by month and descending sum of unit sales.
43. Write a SELECT statement to compute sum of dollar sales by item brand, store state, and month. The result should sum 2016 sales in stores in USA or Canada. The result should contain subtotals for a partial CUBE on item brand and store state along with grouping on month. Sort the result in a convenient ordering. In documentation before the SELECT statement, you should list the subtotal groups in the result.
44. Write a SELECT statement to compute sum of dollar sales by item brand, quarter, month, and day. The result should sum 2016 sales in stores in USA or Canada. The result should contain subtotals for a partial ROLLUP on quarter, month, and day along with grouping on item brand. Sort the result in a convenient ordering. In documentation before the SELECT statement, you should list the subtotal groups in the result.
45. Write a SELECT statement to compute sum of dollar sales by year, quarter, month, and day. The result should sum sales occurring in USA or Canada stores. The result should contain subtotals for a ROLLUP on a composite column (year and quarter), month, and day. Sort the result in a convenient ordering. In documentation before the SELECT statement, you should list the subtotal groups in the result.
46. Write a SELECT statement to compute sum of dollar sales by item brand, store state, month, and day. The result should sum 2016 sales in stores in USA or Canada. The result should contain subtotals for a nested ROLLUP on month and day along with normal subtotals on item brand and store state. Sort the result in a convenient ordering. In documentation before the SELECT statement, you should list the subtotal groups in the result.
47. Write a SELECT statement to compute sum of dollar sales by item brand, store state, and month. The result should sum 2016 sales in stores in USA or Canada.

The result should contain subtotals for a partial CUBE on item brand and store state along with grouping on month. The result columns should contain the grouping columns, sum of dollar sales, and grouping identifier for all grouping columns. Sort the result in a convenient ordering. In documentation before the SELECT statement, you should list the subtotal groups in the result along with the grouping identifier of each subtotal group.

48. Write a SELECT statement to compute the contribution ratio on sum of dollar sales by store city for sales in 2017. The result should contain one set of contribution ratios for all store cities in the result. Order the result by descending sum of dollar sales.
49. Install and use the Pivot4J plugin in Pentaho Business Analytics. The textbook's website contains an installation document and a tutorial with instructions about usage of a predefined cube in the Pivot4J plugin.
50. Use the OLAP cube demonstration on the WebPivotTable website. No installation is required. The textbook's website contains a guided tutorial about the OLAP cube demonstration.

REFERENCES FOR FURTHER STUDY

The Microsoft website contains tutorials about MDX. This Microsoft page (see bullet) contains an overview about data access with MDX along with links for details about query formulation details.

- https://docs.microsoft.com/en-us/sql/analysis-services/multidimensional-models/mdx/multidimensional-model-data-access-analysis-services-multidimensional-data

The Pivot4J page (www.pivot4j.org) contains an overview of Pivot4J and download links for the Pivot4J plugin of Pentaho Business Analytics. WebPivotTable.com contains documentation and a demonstration using a data cube hosted on an external OLAP server. This chapter presented important analytic functions although enterprise DBMSS provide more functions. For details about other analytic functions, consult enterprise DBMS documentation. For example the search terms, "Oracle analytic functions" provides links to documentation about analytic functions provided in Oracle.

part seven

Managing Database Environments

The chapters in Part 7 emphasize the role of database specialists and the details of managing databases in various operating environments. Chapter 16 provides a context for the other chapters through coverage of the responsibilities, tools, and processes used by database administrators and data administrators. The other chapters in Part 7 provide a foundation for managing databases in important environments: Chapter 17 on transaction processing, Chapter 18 on distributed processing, parallel databases, and distributed data, and Chapter 19 on object and NoSQL databases. These chapters emphasize concepts, architectures, and design choices important to database specialists. In addition, Chapters 16 and 19 provide details about SQL statements used in transaction processing and object database development as well as Couchbase N1QL statements to manipulate document databases.

chapter

16

Data and Database Administration

Learning Objectives

This chapter provides detailed coverage about the responsibilities and tools of database specialists known as data administrators and database administrators. After this chapter, the student should have acquired the following knowledge and skills:

- Compare and contrast the responsibilities of database administrators and data administrators especially for meeting challenges of big data
- Write SQL statements to control databases for security and integrity
- Manage stored procedures and triggers
- Understand the roles of data dictionary tables and the information resource dictionary
- Describe the data planning process
- Explain the motivation for data governance and the components of data governance programs in organizations
- Understand the process to select and evaluate DBMSs
- Gain insights about the processing environments in which database technology is used

OVERVIEW

Utilizing the knowledge and skills in Parts 1 through 6, you should be able to develop databases and implement applications that use the databases. You learned about query formulation, conceptual data modeling, relational database design, physical database design, application development with views, stored procedures, triggers, and data warehouse development and processing. Part 7 complements these knowledge and skill areas by exploring concepts and skills involved in managing databases in different processing environments. This chapter describes the responsibilities and tools of data specialists (data administrators and database administrators) and provides an introduction to the different processing environments for databases.

Before learning details of the processing environments, you need to understand the organizational context in which databases exist and learn tools and processes for managing databases. This chapter first discusses an organizational context for databases. You will learn about database support for management decision making, the goals of programs to support information management in organizations, the responsibilities of data and database administrators, and the challenges in managing exploding data growth. After explaining the organizational context, this chapter presents new tools

and processes to manage databases. You will learn SQL statements for security and integrity, management of triggers and stored procedures, and data dictionary manipulation as well as processes for data planning, data governance, and DBMS selection. This chapter concludes with an introduction to the different processing environments that will be presented in more detail in the other chapters of Part 7.

16.1 ORGANIZATIONAL CONTEXT FOR MANAGING DATABASES

This section reviews management decision-making levels and discusses database support for decision making at all levels. After this background, this section describes organizational programs (information resource management, knowledge management, and data governance), responsibilities of data specialists to manage information resources, and challenges of managing exploding data growth.

16.1.1 Database Support for Management Decision Making

Operational Database
a database to support the daily functions of an organization.

Databases support business operations and management decision making at various levels. Most large organizations have developed many **operational databases** to help conduct business efficiently. Operational databases directly support major functions such as order processing, manufacturing, accounts payable, and product distribution. The reasons for investing in an operational database are typically faster processing, larger volumes of business, and reduced personnel costs.

As organizations achieve improved operations, they begin to realize the decision-making potential of their databases. Operational databases provide the raw materials for management decision making as depicted in Figure 16.1. Lower-level management can obtain exception and problem reports directly from operational databases. However, much value must be added to leverage the operational databases for middle and upper management. The operational databases must be cleaned, integrated, and summarized to provide value for tactical and strategic decision making. Integration is necessary because operational databases often are developed in isolation without regard for the information needs of tactical and strategic decision making.

Table 16-1 provides examples of management decisions and data requirements. Lower-level management deals with short-term problems related to individual transactions. Periodic summaries of operational databases and exception reports assist operational management. Middle management relies on summarized data that are integrated across operational databases. Middle management may want to integrate data across different departments, manufacturing plants, and retail stores. Top management relies on the results of middle management analysis and external data sources. Top management needs to integrate data so that customers, products, suppliers, and other important entities can be tracked across an entire organization. In addition, external data must be captured and then integrated with internal data.

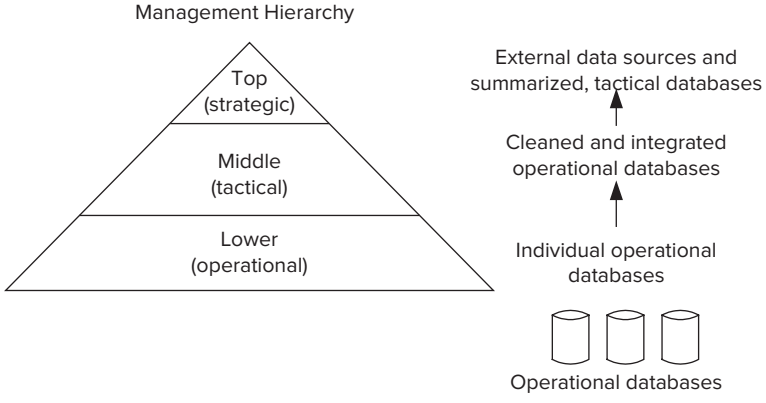


FIGURE 16.1
Database Support for Management Levels

Level	Example Decisions	Data Requirements
Top	Identify new markets and products; plan growth; reallocate resources across divisions	Economic and technology forecasts; news summaries; industry reports; medium-term performance reports
Middle	Choose suppliers; forecast sales, inventory, and cash; revise staffing levels; prepare budgets	Historical trends; supplier performance; critical path analysis; short-term and medium-term plans
Lower	Schedule employees; correct order delays; find production bottlenecks; monitor resource usage	Problem reports; exception reports; employee schedules; daily production results; inventory levels

TABLE 16-1
Examples of Management Decision Making

16.1.2 Approaches for Managing Data Resources

As a response to the challenges of leveraging operational databases and information technology for management decision making, several management approaches have been developed over the last two decades. The original approach known as information resource management was developed in the 1990s. Information resource management involves processing, distributing, and integrating information throughout an organization. A key element of information resource management is control of **information life cycles** (Figure 16.2). Each level of management decision making and business operations has its own information life cycle. For effective decision making, the life cycles must be integrated to provide timely and consistent information. For example, information life cycles for operations provide input to life cycles for management decision making.

Information Life Cycle
the stages of information transformation in an organization. Each entity has its own information life cycle that should be managed and integrated with the life cycles of other entities.

Data quality is a particular concern for information resource management because of the impact of data quality on management decision making. As discussed in Chapter 2, data quality involves a number of dimensions such as correctness, timeliness, consistency, completeness, and reliability. Often the level of data quality that suffices for business operations may be insufficient for decision making at upper levels of management. This conflict is especially true for the consistency dimension. For example, inconsistency of customer identification across operational databases can impair decision making at the upper management level. Information resource management emphasizes a long-term, organization-wide perspective on data quality to ensure support for management decision making.

Starting about the mid-1990s, a movement developed to extend information resource management into **knowledge management**. Traditionally, information resource management has emphasized technology to support predefined recipes for decision making rather than the ability to react to a constantly changing business environment. To succeed in today's business environment, organizations must emphasize fast response and adaptation to extend planning efforts. To meet this challenge, organizations should develop systems that facilitate knowledge creation rather than information management. For knowledge creation, a greater emphasis is on human information processing and organization dynamics to balance the technology emphasis, as shown in Figure 16.3.

Knowledge Management
applying information technology with human information processing capabilities and organization processes to support rapid adaptation to change.

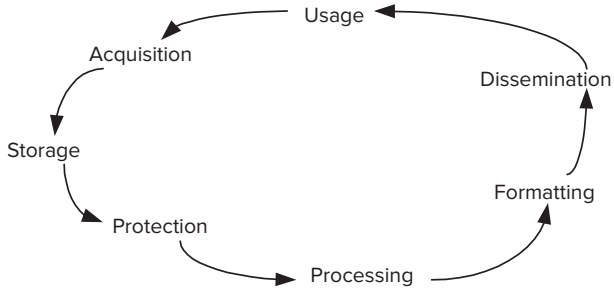


FIGURE 16.2
Typical Stages of an Information Life Cycle

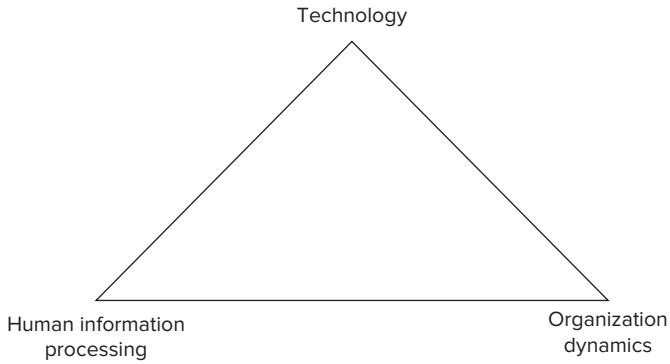


FIGURE 16.3
Three Pillars of Knowledge Management

This vision for knowledge management provides a context for usage of information technology to solve business problems. The best information technology will fail if not aligned with human and organization elements. Information technology should amplify individual intellectual capacity, compensate for limitations in human processing, and support positive organization dynamics.

The emphasis on information and knowledge management has shifted to data governance over the last decade. The rapid growth of electronic commerce and financial scandals in the 2000s propelled major changes in regulatory oversight and corporate responsibilities. In the U.S., the Sarbanes-Oxley law, the Health Insurance Portability and Accountability Act, and the more recent Dodd-Frank law have added new corporate responsibilities for data management. The European Union has enacted broad data privacy directives impacting organizations with any European operations.

Data Governance
according to the Data Governance Institute, data governance involves the application of decision-making and authority for data-related issues.

This changed environment has spurred the movement for **data governance**. According to the Data Governance Institute (http://www.datagovernance.com), "data governance is the exercise of decision-making and authority for data-related matters." Data governance attempts to mitigate risks associated with the complex regulatory environment, information security, and information privacy especially for personal identifiable data and related business transactions.

Data governance provides a system of checks and balances to develop data rules and policies, support application of data rules and policies, and evaluate compliance of data rules and policies as depicted in Figure 16.4. The system of data governance operates in a manner similar to separate government branches in which the legislative branch makes laws, the executive branch enforces laws, and the judicial branch resolves disputes about the meaning and application of laws.

Data governance has been applied to a number of corporate initiatives. Perceived shortcomings in data quality especially in attributes impacted by regulations are primary drivers of data governance initiatives. For data quality improvements,

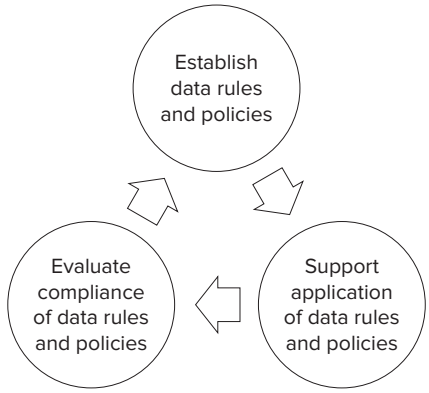


FIGURE 16.4
Data Governance Checks and Balances

data governance initiatives typically focus on development of data quality measures, reporting status of data quality, and establishing decision rights and accountabilities. Mergers and acquisitions often trigger data governance initiatives to ensure consistent data definitions and integrate corporate policies involving data privacy and security. Similarly, business intelligence developments often lead to data governance initiatives to establish policies for data integration particularly for changes to source data in different parts of an organization.

16.1.3 Responsibilities of Data Specialists

As part of controlling information resources, new management responsibilities have arisen. The data administrator (DA) is a middle- or upper-management position with broad responsibilities for information resource management. The database administrator (DBA) is a support role with responsibilities related to individual databases and DBMSs. Table 16-2 compares the responsibilities of data administrators and database administrators. The data administrator views the information resource in a broader context than the database administrator. The data administrator considers all kinds of data whether stored in relational databases, files, web pages, or external sources. The data administrator also supports data governance through membership in the data governance office and consulting on activities managed by the data governance office. The database administrator typically considers only data stored in databases and implementing controls to support data governance policies.

Development of an **enterprise data model** is one of the most important responsibilities of the data administrator. An enterprise data model provides an integrated model of all databases of an organization. Because of its scope, an enterprise data model is less detailed than the individual databases that it encompasses. The enterprise data model concentrates on the major subjects in operational databases rather the full details. An enterprise data model can be developed for data planning (what databases to develop) or business intelligence (how to integrate and summarize existing databases). Section 16.3 describes the details of data planning. In Chapters 13 and 14, you learned details of data warehouse design and data integration, important implementations of an enterprise data model for business intelligence.

Enterprise Data Model
a conceptual data model of an organization. An enterprise data model can be used for data planning or business intelligence.

Large organizations may offer much specialization in data administration and database administration. For data administration, specialization can occur by task and environment. On the task side, data administrators can specialize in planning versus policy establishment. On the environment side, data administrators can specialize in environments such as business intelligence, operations, and nontraditional data such as images, text, and video. For database administration, specialization can occur by DBMS, task, and environment. Because of the complexities of learning a DBMS, DBAs typically specialize in one product. Task specialization is usually divided between

TABLE 16-2
Responsibilities of Data Administrators and Database Administrators

Position	Responsibilities
Data administrator	Develops an enterprise data model Establishes inter-database standards and policies about naming, data sharing, and data ownership Negotiates contractual terms with information technology vendors Develops long-range plans for information technology Supports data governance activities
Database administrator	Develops detailed knowledge of individual DBMSs Consults on application development Performs data modeling, logical database design, and physical database design Enforces data administration standards Monitors database performance Performs technical evaluation of DBMSs Creates security, integrity, and rule-processing statements Devises standards and policies related to individual databases and DBMSs

data modeling, application development support, and performance evaluation. Environment specialization is usually divided between transaction processing and data warehouses.

In large organizations, various titles are used for database specialists. The following list explains some common titles used in large organizations.

- Database architect: primarily specializes in data modeling and logical database design
- System DBA: interfaces with system administration and analyzes database impact on hardware and operating system
- Application DBA: specializes in management and usage of procedural objects including triggers, stored procedures, and transaction design
- Senior DBA: a highly experienced DBA who supervises junior DBAs and provides expert trouble shooting
- Performance DBA: specializes in physical database design and performance tuning
- Data warehouse administrator: specializes in operation and development of data warehouses

In small organizations, the boundary between data administration and database administration is fluid. There may not be separate positions for data administrators and database administrators. The same person may perform duties from both positions. As organizations grow, specialization usually develops so that separate positions are created.

16.1.4 Challenges of Big Data

In many organizations, data specialists confront the problem of exploding data growth. According to the 2014 Digital Universe Study by IDC, the volume of digital data will continue to double every two years, reaching 44 trillion gigabytes by 2020. IDC predicts that the volume of digital data will increase from 10 trillion gigabytes in 2015 to 180 trillion gigabytes in 2025. To digest this increasing deluge of data, IDC predicts that worldwide revenues for big data management and analytics will grow at a compound growth rate of 11.7% from 2016 to 2020.

The growth in data comes from a variety of sources such as sensors in smart phones, energy meters, and automobiles, interaction of individuals in social media websites, radio frequency identification tags in retail, and digitized multimedia content in medicine, entertainment, and security. This data growth breaks existing systems and business processes, providing challenges and opportunities for both vendors developing database technology and organizations using database technology.

Big Data
the phenomenon of exploding data growth. Big data has three dimensions, volume, velocity, and variety. The volume of big data that exceeds the limits of database software depends on technology, industry sector, and time.

The phenomenon of explosive data growth known as **big data** was first stated by Doug Laney of the Meta Group in 2001[1]. According to Laney's report, big data contains three dimensions: volume (amount of data), velocity (rate of generating and processing data), and variety (type of data especially the distinction between structured and unstructured data). Most attention is focused on data volumes but the other two dimensions must be managed effectively to deal with problems of big data. Complementing the dimensions of big data, the McKinsey Global Institute defines big data as "datasets whose size is beyond the ability of typical database software to capture, store, manage, and analyze." This definition provides flexibility to vary volumes considered as big data by technology, industry sector, and time.

Big data creates opportunities if managed well. Organizations can unlock the value of big data through increasing velocity on application processing speeds, improving accuracy of forecasts and performance of business units, narrowing segmentation of customers, improving decision making through analytics, and developing new generations of products and services.

[1] Laney, Doug, "3D Data Management: Controlling Data Volume, Velocity and Variety," META Group (now Gartner), February 2001.

TABLE 16-3
Data Unit Sizes for Big Data

Data Unit	Big Data Example
Terabyte (TB) 1,024 (1,000) GB	Typical hard drive size on a laptop computer in 2017 is 1 TB.
Petabyte (PB) 1,024 (1,000) TB	Teradata Database 14 has a capacity of 50 PB of compressed data.
Exabtye (EB) 1,024 (1,000) PB	Estimate of global IP traffic in 2021 by Cisco is 278 EB per month.
Zettabyte (ZB) 1,024 (1,000) EB	Cisco estimate of total volume of IP traffic in 2021 is 3.3 ZB. IDC estimate of digital universe in 2020 is 40 ZB.
Yottabyte (YB) 1,024 (1,000) ZB	High definition video of all human activity would be approximately 100 YB. Estimate of storage capacity of U.S. National Security data center is 1.0 YB.

To effectively manage big data, data specialists should have a clear understanding of data volume units. Basic units of data are the byte (one character or 8 bits), kilobyte (KB), megabyte (MB), and gigabyte (GB). A kilobyte is either 1,024 bytes (binary system measure) or 1,000 bytes (metric system measure). Since computers are binary machines, kilo means 1,024 for digital storage. Storage manufacturers often use the metric system measure perhaps because the public does not understand binary measures. However using the metric system measure provides less capacity than the binary system measure. Thus, a megabyte denotes either 1,000 or 1,024 KB while a gigabyte denotes either 1,000 or 1,024 MB.

Table 16-3 extends the basic data units to larger units along with big data examples. Terabyte was big data in previous decades. Now, it is the typical capacity of hard drives on personal computers. Petabyte is yesterday's big data as shown by efforts at Google and Teradata to manage petabyte levels. Exabyte is generally considered as the current big data level as demonstrated by internet traffic rates. Zettabyte and yottabyte are emerging big data levels with hypothetical examples of storage capacity needs.

16.2 TOOLS OF DATABASE ADMINISTRATION

To fulfill the responsibilities mentioned in the previous section, database administrators use a variety of tools. You already have learned about tools for data modeling, logical database design, view creation, physical database design, triggers, and stored procedures. Some of the tools are SQL statements (CREATE VIEW and CREATE INDEX) while others are part of CASE tools for database development. This section presents additional tools for security, integrity, and data dictionary access and discusses management of stored procedures and triggers.

16.2.1 Security

Security involves protecting a database from unauthorized access and malicious destruction. Because of the value of data in corporate databases, there is strong motivation for unauthorized users to gain access to corporate databases. Competitors have strong motivation to access sensitive information about product development plans, cost-saving initiatives, and customer profiles. Lurking criminals want to steal unannounced financial results, business transactions, and sensitive customer data such as credit card numbers. Social deviants and terrorists can wreak havoc by intentionally destroying database records. With growing use of the Web to conduct business, competitors, criminals, and social deviants have even more opportunity to compromise **database security**.

Database Security
protecting databases from unauthorized access and malicious destruction.

Security is a broad subject involving many disciplines. There are legal and ethical issues about who can access data and when data can be disclosed. There are network, hardware, operating system, and physical controls that augment the controls

provided by DBMSs. There are also operational problems about passwords, authentication devices, and privacy enforcement. These issues are not further addressed because they are beyond the scope of DBMSs and database specialists. The remainder of this subsection emphasizes access control approaches and SQL statements for authorization rules.

Authorization Rules
define authorized users, allowable operations, and accessible parts of a database. The database security system stores authorization rules and enforces them for each database access.

For access control, DBMSs support creation and storage of authorization rules and enforcement of authorization rules when users access a database. Figure 16.5 depicts the interaction of these elements. Database administrators create **authorization rules** that define allowable operations on database objects for users. Enforcement of authorization rules involves authenticating a user and ensuring that authorization rules are not violated by access requests (database retrievals and modifications). Authentication occurs when a user first connects to a DBMS. Authorization rules must be checked for each access request.

Discretionary Access Control
users are assigned access rights or privileges to specified parts of a database. Discretionary access control is the most common kind of security control supported by commercial DBMSs.

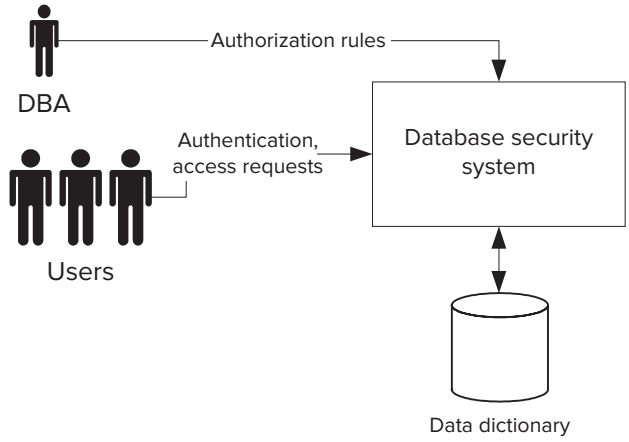
The most common approach to authorization rules is known as **discretionary access control**. In discretionary access control, users are assigned access rights or privileges to specified parts of a database. For precise control, privileges are usually specified for views rather than tables or fields. Users can be given the ability to read, update, insert, and delete specified parts of a database. To simplify the maintenance of authorization rules, privileges should be assigned to groups or roles rather than individual users. Because roles are more stable than individual users, authorization rules that reference roles require less maintenance than rules referencing individual users. Users are assigned to roles and given passwords. During the database login process, the database security system authenticates users and notes the roles to which they belong.

Mandatory Access Control
a database security approach for highly sensitive and static databases. A user can access a database element if the user's clearance level provides access to the classification level of the element.

Mandatory access controls are less flexible than discretionary access controls. In mandatory control approaches, each object is assigned a classification level and each user is given a clearance level. A user can access an object if the user's clearance level provides access to the classification level of the object. Typical clearance and classification levels are confidential, secret, and top secret. Mandatory access control approaches primarily have been applied to highly sensitive and static databases for national defense and intelligence gathering. Because of the limited flexibility of mandatory access controls, only a few DBMSs support them. DBMSs that are used in national defense and intelligence gathering must support mandatory controls, however.

In addition to access controls, DBMSs support encryption of databases. Encryption involves the encoding of data to obscure their meaning. An encryption algorithm changes the original data (known as the plaintext). To decipher the data, the user supplies an encryption key to restore the encrypted data (known as the ciphertext) to its original (plaintext) format. Enterprise DBMSs usually allow a choice of encryption algorithms such as the Advanced Encryption Standard (AES) and the Triple Data Encryption Standard (Triple DES). To protect data at the operating system level,

FIGURE 16.5
Database Security System



enterprise DBMSs such as Oracle, Microsoft, and IBM apply encryption to the file level in a technology known as Transparent Data Encryption (TDE). In addition, enterprise DBMSs typically support government security standards such as the Federal Information Processing Standard 140 of the U.S. government for database deployment in a government environment.

SQL:2016 Security Statements SQL:2016 supports discretionary authorization rules using the CREATE/DROP ROLE statements and the GRANT/REVOKE statements. When a role is created, the DBMS grants the role to either the current user or current role. In Example 16.1, the ISFaculty and ISAdvisor roles are granted to the current user while the ISAdministrator role is granted to the role of the current user. The WITH ADMIN clause means that a user assigned the role can assign the role to others. The WITH ADMIN option should be used sparingly because it provides wide latitude to the role. A role can be dropped with the DROP ROLE statement.

Example 16.1 (SQL:2016)

CREATE ROLE Statement Examples

```
CREATE ROLE ISFaculty;
CREATE ROLE ISAdministrator WITH ADMIN CURRENT_ROLE;
CREATE ROLE ISAdvisor;
```

In a GRANT statement, you specify the privileges (see Table 16-4), object (table, column, or view), and list of authorized users (or roles). In Example 16.2, SELECT access is given to three roles (ISFaculty, ISAdvisor, ISAdministrator) while UPDATE access is given only to the ISAdministrator. Individual users must be assigned to roles before they can access the *ISStudentGPA* view.

Example 16.2 (SQL:2016)

View Definition, GRANT, and REVOKE Statements

```
CREATE VIEW ISStudentGPA AS
  SELECT StdNo, StdFirstName, StdLastName, StdGPA
    FROM Student
    WHERE StdMajor = 'IS';
-- Grant privileges to roles
GRANT SELECT ON ISStudentGPA
  TO ISFaculty, ISAdvisor, ISAdministrator;
GRANT UPDATE ON ISStudentGPA.StdGPA TO ISAdministrator;
-- Assign users to roles
GRANT ISFaculty TO Mannino;
GRANT ISAdvisor TO Olson;
GRANT ISAdministrator TO Smith WITH GRANT OPTION;

REVOKE SELECT ON ISStudentGPA FROM ISFaculty RESTRICT;
```

The GRANT statement can also be used to assign users to roles as shown in the last three GRANT statements in Example 16.2. In addition to granting the privileges in Table 16-4, a user can be authorized to pass privileges to other users using the WITH GRANT OPTION keyword. In the last GRANT statement of Example 16.2, user Smith

TABLE 16-4
Explanation of Common SQL:2016 Privileges

Privilege	Explanation
SELECT	Query the object; can be specified for individual columns
UPDATE	Modify the value; can be specified for individual columns
INSERT	Add a new row; can be specified for individual columns
DELETE	Delete a row; cannot be specified for individual columns
TRIGGER	Create a trigger on a specified table
REFERENCES	Reference columns of a given table in integrity constraints
EXECUTE	Execute the stored procedure

can grant the ISAdministrator role to other users. The WITH GRANT option should be used sparingly because it provides wide latitude to the user.

To remove an access privilege, the REVOKE statement is used. In the last statement of Example 16.2, the SELECT privilege is removed from ISFaculty. The RESTRICT clause means the privilege is revoked only if the privilege has not been granted to the specified role by more than one user.

Security Specification in Oracle DBMS Oracle extends the SQL:2016 security statements with the CREATE USER statement, predefined roles, and additional privileges. In SQL:2016, user creation is an implementation issue. Since Oracle does not rely on the operating system for user creation, it provides the CREATE USER statement. Oracle provides predefined roles to simplify role definition including the CONNECT role to create a session, the RESOURCE role for creating tables and application objects such as stored procedures, and the DBA role for managing databases. Oracle discourages usage of these legacy roles although these roles continue to have substantial usage. Oracle provides a number of other predefined roles described in the Oracle documentation. In addition, Oracle provides the PUBLIC user group to support common privileges for all users.

For privileges, Oracle distinguishes between system privileges (independent of object) and object privileges. Granting system privileges usually is reserved for highly secure roles because of the far-reaching nature of system privileges as shown in Table 16-5. Typically accounts with DBA or SYS role should have system privileges. The ORACLE object privileges are similar to the SQL:2016 privileges except that Oracle provides more objects than SQL:2016, as shown in Table 16-6.

In addition to extensions to the standard SQL security statements, Oracle provides advanced security features (security policies, auditing, and profiles) with no counterpart in the SQL:2016 specification. Security policies support dynamic restrictions for fine-grained control to the row and column level. Security restrictions based on views

TABLE 16-5
Explanation of Common Oracle System Privileges

System Privilege	Explanation
CREATE X, CREATE ANY X	Create objects of kind X in one's schema[2]; CREATE ANY allows creating objects in other schemas
ALTER X, ALTER ANY X	Alter objects of kind X in one's schema; ALTER ANY X allows altering objects in other schemas
INSERT ANY, DELETE ANY, UPDATE ANY, SELECT ANY	Insert, delete, update, and select from a table in any schema
DROP X, DROP ANY X	DROP objects of kind X in one's schema; DROP ANY allows dropping of objects in other schemas
ALTER SYSTEM, ALTER DATABASE, ALTER SESSION	Issue ALTER SYSTEM commands, ALTER DATABASE commands, and ALTER SESSION commands
ANALYZE ANY	Analyze any table, index, or cluster

[2] A schema is a collection of related tables and other Oracle objects that are managed as a unit.

TABLE 16-6
Mapping between Common Oracle Privileges and Objects

Privilege	Object				
	Table	View	Sequence[3]	Procedure, Function, Package, Library, Operator, Index, Type	Materialized View[4]
ALTER	X		X		
DELETE	X	X			X
EXECUTE				X	
INDEX	X				
INSERT	X	X			X
REFERENCES	X	X			
SELECT	X	X	X		X
UPDATE	X	X			X

alone cannot be customized to the individual user level without a large number of views, one per user. Security policies in Oracle support dynamic generation of conditions so that access can be restricted based on individual user characteristics such as customer numbers and employee departments. Oracle provides auditing to record user database actions. Auditing can be triggered by combinations of user name, statement type, time, and database object. In addition, security policies can trigger auditing when specified elements in an Oracle database are accessed or changed. Profiles specify resource limits for roles and users. Profiles can restrict CPU time, memory usage, data block accesses, idle time, elapsed time, and concurrent sessions.

Authorization restrictions by application objects such as forms and reports should also be supported in addition to the database objects permissible in the GRANT statement. These additional security constraints are usually specified in proprietary interfaces or in application development tools, rather than in SQL. For example, Microsoft Access 2003 allows definition of authorization rules for database objects (tables and stored queries) as well as application objects (forms and reports). However, Access 2007 and later versions dropped support for application-level security, instead relying on SQL security constraints and improved control of components that may pose security risks.

The fundamental problem with application level security is the ease of bypass. Any usage outside of the application bypasses application level security. Thus, enterprise DBMS vendors emphasize database level security, not application level security.

16.2.2 Integrity Constraints

You have already seen integrity constraints presented in previous chapters. In Chapter 3, you learned about primary keys, foreign keys, candidate keys, and nonnull constraints along with the corresponding SQL syntax. In Chapter 5, you studied cardinality constraints and generalization hierarchy constraints. In Chapter 7, you studied functional and multivalued dependencies as part of the normalization process. Chapter 8 described indexes that can be used to enforce primary and candidate key constraints efficiently. Chapter 11 presented triggers that can be used to specify complex integrity constraints. This subsection describes additional kinds of integrity constraints and the corresponding SQL syntax.

SQL Domains In Chapter 3, standard SQL data types were defined. A data type indicates the kind of data (character, numeric, yes/no, etc.) and permissible operations (numeric operations, string operations, etc.) for columns using the data type.

[3] A sequence is a collection of values maintained by Oracle. Sequences typically are used for system-generated primary keys.

[4] Recall from Chapter 15 that a materialized view is stored rather than derived. Materialized views are useful in data warehouses as presented in Chapter 15.

SQL:2016 provides a limited ability to define new data types using the CREATE DOMAIN statement. A domain can be created as a subset of a standard data type. Example 16.3 demonstrates the CREATE DOMAIN statement along with usage of the new domains in place of standard data types. The CHECK clause defines a constraint for the domain limiting the domain to a subset of the standard data type.

Example 16.3 (SQL:2016)

CREATE DOMAIN Statements and Usage of the Domains

```
CREATE DOMAIN StudentClass AS CHAR(2)
  CHECK ( VALUE IN ('FR', 'SO', 'JR', 'SR') )

CREATE DOMAIN CourseUnits AS SMALLINT
  CHECK ( VALUE BETWEEN 1 AND 9 )
```

In the CREATE TABLE statement for the *Student* table, the domain can be referenced in the *StdClass* column.

```
   StdClass    StudentClass  NOT NULL
```

In the CREATE TABLE statement for the *Course* table, the domain can be referenced in the *CrsUnits* column.

```
   CrsUnits    CourseUnits    NOT NULL
```

SQL:2016 provides a related feature known as a distinct type. Like a domain, a distinct type is based on a primitive type. Unlike a domain, a distinct type cannot have constraints. However, the SQL specification provides improved type checking for distinct types as compared to domains. A column having a distinct type can be compared only with another column using the same distinct type. Example 16.4 demonstrates distinct type definitions and a comparison among columns based on the types.

Example 16.4 (SQL:2016)

Distinct Types and Usage of the Distinct Types

```
-- USD distinct type and usage in a table definition
CREATE DISTINCT TYPE USD AS DECIMAL(10,2);
USProdPrice  USD

CREATE DISTINCT TYPE Euro AS DECIMAL(10,2);
EuroProdPrice  Euro

-- Type error: columns have different distinct types
USProdPrice > EuroProdPrice
```

For object-oriented databases, SQL:2016 provides user-defined types, a more powerful capability than domains or distinct types. User-defined data types can be defined with new operators and functions. In addition, user-defined data types can be defined using other user-defined data types. Chapter 19 describes user-defined data types as part of the presentation of the object-oriented features of SQL:2016. Because of the limitations, most DBMSs no longer support domains and distinct types. For example, Oracle supports user-defined types but does not support domains or distinct types.

CHECK Constraints in the CREATE TABLE Statement When a constraint involves row conditions on columns of the same table, a CHECK constraint may be used. CHECK constraints are specified as part of the CREATE TABLE statement as shown in Example 16.5. For easier traceability, you should always use constraint names. When a constraint violation occurs, most DBMSs will display the constraint name.

Example 16.5 (SQL:2016)

CHECK Constraint Clauses

Here is a CREATE TABLE statement with CHECK constraints for the valid GPA range and upper-class students (juniors and seniors) having a declared (non-null) major.

```
CREATE TABLE Student
( StdNo        CHAR(11),
  StdFirstName VARCHAR(50) CONSTRAINT StdFirstNameRequired NOT NULL,
  StdLastName  VARCHAR(50) CONSTRAINT StdLastNameRequired NOT NULL,
  StdCity      VARCHAR(50) CONSTRAINT StdCityRequired NOT NULL,
  StdState     CHAR(2)     CONSTRAINT StdStateRequired NOT NULL,
  StdZip       CHAR(9)     CONSTRAINT StdZipRequired NOT NULL,
  StdMajor     CHAR(6),
  StdClass     CHAR(6),
  StdGPA       DECIMAL(3,2),
CONSTRAINT PKStudent PRIMARY KEY (StdSSN),
CONSTRAINT ValidGPA CHECK ( StdGPA BETWEEN 0 AND 4 ),
CONSTRAINT MajorDeclared CHECK
  ( StdClass IN ('FR','SO') OR StdMajor IS NOT NULL ) )
```

Although CHECK constraints are widely supported, most DBMSs limit the conditions inside CHECK constraints. The SQL:2016 specification allows any condition that could appear in a SELECT statement including conditions that involve SELECT statements. Most DBMSs do not permit conditions involving SELECT statements in a CHECK constraint. For example, Oracle prohibits SELECT statements in CHECK constraints as well as references to columns from other tables. For these complex constraints, assertions may be used (if supported by the DBMS) or triggers if assertions are not supported.

SQL:2016 Assertions SQL:2016 assertions are more powerful than constraints about domains, columns, primary keys, and foreign keys. Unlike CHECK constraints, assertions are not associated with a specific table. An assertion can involve a SELECT statement of arbitrary complexity. Thus, assertions can be used for constraints involving multiple tables and statistical calculations, as demonstrated in Examples 16.6 through 16.8. However, complex assertions should be used sparingly because they can be inefficient to enforce. There may be more efficient ways to enforce assertions such as through event conditions in a form and stored procedures. As a DBA, you are advised to investigate the event programming capabilities of application development tools before using complex assertions.

Assertions are checked after related modification operations complete. For example, the *OfferingConflict* assertion in Example 16.7 would be checked for each insertion of an *Offering* row and for each change to one of the columns in the WHERE clause of the assertion. In some cases, an assertion should be delayed until other statements complete. The keyword DEFERRABLE can be used to allow an assertion to be tested at the end of a transaction rather than immediately. Deferred checking is an issue with transaction design discussed in Chapter 17.

Example 16.6 (SQL:2016)

CREATE ASSERTION Statement.

This assertion statement ensures that each faculty has a course load between three and nine units.

```
CREATE ASSERTION FacultyWorkLoad
 CHECK ( NOT EXISTS
  ( SELECT Faculty.FacNo, OffTerm, OffYear
     FROM Faculty, Offering, Course
     WHERE Faculty.FacNo = Offering.FacNo
       AND Offering.CourseNo = Course.CourseNo
     GROUP BY Faculty.FacNo, OffTerm, OffYear
     HAVING SUM(CrsUnits) < 3 OR SUM(CrsUnits) > 9 ) )
```

Example 16.7 (SQL:2016)

CREATE ASSERTION Statement

This assertion statement ensures that no two courses are offered at the same time and place. The conditions involving the *OffTime* and *OffDays* columns should be refined to check for any overlap, not just equality. Because these refinements would involve string and date functions specific to a DBMS, they are not shown.

```
CREATE ASSERTION OfferingConflict
 CHECK ( NOT EXISTS
  ( SELECT O1.OfferNo
     FROM Offering O1, Offering O2
     WHERE O1.OfferNo <> O2.OfferNo
       AND O1.OffTerm = O2.OffTerm
       AND O1.OffYear = O2.OffYear
       AND O1.OffDays = O2.OffDays
       AND O1.OffTime = O2.OffTime
       AND O1.OffLocation = O2.OffLocation  ) )
```

Example 16.8 (SQL:2016)

Assertion Statement to Ensure that Full-Time Students Have at Least Nine Units

```
CREATE ASSERTION FullTimeEnrollment
 CHECK ( NOT EXISTS
  ( SELECT Enrollment.RegNo
     FROM Registration, Offering, Enrollment, Course
     WHERE Offering.OfferNo = Enrollment.OfferNo
       AND Offering.CourseNo = Course.CourseNo
       AND Offering.RegNo = Registration.RegNo
       AND RegStatus = 'F'
     GROUP BY Enrollment.RegNo
     HAVING SUM(CrsUnits) >= 9 ) )
```

Assertions are not widely supported because assertions overlap with triggers. An assertion is a limited kind of trigger with an implicit condition and action. Because assertions are simpler than triggers, they are usually easier to create and more efficient to execute. However, no enterprise DBMS supports assertions so triggers must be used in places where assertions would be more appropriate.

16.2.3 Management of Triggers and Stored Procedures

In Chapter 11, you learned about the concepts and coding details of stored procedures and triggers. Although a DBA writes stored procedures and triggers to help manage databases, the primary responsibilities for a DBA are to manage stored procedures and triggers, not to write them. The DBA's responsibilities include setting standards for coding practices, monitoring dependencies, and understanding trigger interactions.

For coding practices, a DBA should consider documentation standards, parameter usage, and content, as summarized in Table 16-7. Documentation standards may include naming standards, explanations of parameters, and descriptions of pre- and post-conditions of procedures. Parameter usage in procedures and functions should be monitored. Functions should use only input parameters and not have side effects. For content, triggers should not perform integrity checking that can be coded as declarative integrity constraints (CHECK constraints, primary keys, foreign keys, …). To reduce maintenance, triggers and stored procedures should reference the data types of associated database columns. In Oracle, this practice involves anchored data types. Because most application development tools support triggers and event procedures for forms and reports, the choice between a database trigger/procedure versus an application trigger/procedure is not always clear. A DBA should participate in setting standards that provide guidance between using database triggers and procedures as opposed to application triggers and event procedures.

A stored procedure or trigger depends on the tables, views, procedures, and functions that it references as well as on access plans created by the SQL compiler. When a referenced object changes, its dependents should be recompiled. In Figure 16.6, trigger X needs recompilation if changes are made to the access plan for the UPDATE statement in the trigger body. Likewise, the procedure needs recompilation if the access plan for the SELECT statement becomes outdated. Trigger X may need recompilation if changes are made to table A or to procedure pr_LookupZ. Most DBMSs maintain dependencies to ensure that stored procedures and triggers work correctly. If a procedure or trigger uses an SQL statement, most DBMSs will automatically recompile the procedure or trigger if the associated access plan becomes obsolete.

A DBA should be aware of the limitations of DBMS-provided tools for dependency management. Table 16-8 summarizes the dependency management issues of access plan obsolescence, modification of referenced objects, and deletion of referenced objects. For access plans, a DBA should understand that manual recompilation may be necessary if optimizer statistics become outdated. For remotely stored procedures and functions, a DBA can choose between timestamp and signature dependency maintenance. With timestamp maintenance, a DBMS will recompile a dependent object for any change in referenced objects. Timestamp maintenance may lead to excessive recompilation because many changes to referenced objects do not require recompilation of the dependent objects. Signature maintenance involves recompilation when a signature (parameter name or usage) changes. A DBA also should be aware that a DBMS will not recompile a procedure or trigger if a referenced object is deleted. The dependent procedure or trigger will be marked as invalid because recompilation is not possible.

Trigger interactions were discussed in Chapter 11 as part of trigger execution procedures. Triggers interact when one trigger fires other triggers and when triggers

TABLE 16-7
Summary of Coding Practice Concerns for a DBA

Coding Practice Area	Concerns
Documentation	Procedure and trigger naming standards; explanation of parameters; comments describing pre- and post-conditions
Parameter usage	Only input parameters for functions; no side effects for functions
Trigger and procedure content	Do not use triggers for standard integrity constraints; usage of anchored data types for variables; standards for application triggers and event procedures versus database triggers and procedures

FIGURE 16.6
Dependencies among Database Objects

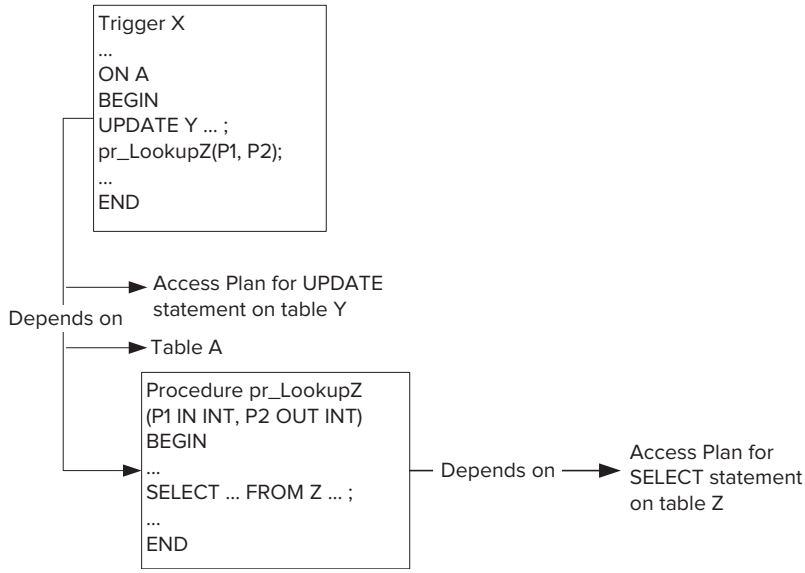


TABLE 16-8
Summary of Dependency Concerns for a DBA

Dependency Area	Concerns
Access plan obsolescence	DBMS should automatically recompile. DBA may need to recompile when optimizer statistics become outdated.
Modification of referenced objects	DBMS should automatically recompile. DBA should choose between timestamp and signature maintenance for remote procedures and functions.
Deletion of referenced objects	DBMS marks procedure/trigger as invalid if referenced objects are deleted.

overlap leading to firing in arbitrary order. A DBA can use trigger analysis tools provided by a DBMS vendor or manually analyze trigger interactions if no tools are provided. A DBA should require extra testing for interacting triggers. To minimize trigger interaction, a DBA should implement guidelines like those summarized in Table 16-9.

16.2.4 Data Dictionary Manipulation

Metadata
data that describe other data including the source, use, value, and meaning of the data.

The data dictionary is a special database that describes individual databases and the database environment. The data dictionary contains data descriptors called **metadata** that define the source, use, value, and meaning of data. DBAs typically deal with two kinds of data dictionaries to track the database environment. Each DBMS provides a data dictionary to track tables, columns, triggers, indexes, and other objects managed by the DBMS. Independent CASE tools provide a data dictionary known as the information resource dictionary that tracks a broader range of objects relating to information systems development. This subsection provides details about both kinds of data dictionaries.

TABLE 16-9
Summary of Guidelines to Control Trigger Complexity

Guideline	Explanation
BEFORE ROW triggers	Do not use data manipulation statements in BEFORE ROW triggers to avoid firing other triggers.
UPDATE triggers	Use a list of columns for UPDATE triggers to reduce trigger overlap.
Actions on referenced rows	Be cautious about triggers on tables affected by actions on referenced rows. These triggers will fire as a result of actions on parent tables.
Overlapping triggers	Do not depend on a specific firing order.

Catalog Tables in SQL:2016 and Oracle SQL:2016 contains catalog tables in the Definition_Schema as summarized in Table 16-10. The Definition_Schema contains one or more catalog tables corresponding to each object that can be created in an SQL data definition or data control statement. The base catalog tables in the Definition_Schema are not meant to be accessed in applications. For access to metadata in applications, SQL:2016 provides the Information_Schema that contains views of the base catalog tables of the Definition_Schema.

The SQL:2016 Definition_Schema and Information_Schema have few implementations because most DBMSs already had proprietary catalog tables long before the standard was released. Thus, you will need to learn the catalog tables of each DBMS with which you work. Typically, a DBMS may have hundreds of catalog tables. However, for any specific task such as managing triggers, a DBA needs to use a small number of catalog tables. Table 16-11 lists some of the most important catalog tables in Oracle.

A DBA implicitly modifies catalog tables when using data definition commands such as the CREATE TABLE statement. The DBMS uses catalog tables to process queries, authorize users, check integrity constraints, and perform other database processing. The DBMS consults catalog tables before performing almost every action. Thus, integrity of catalog tables is crucial to the operation of the DBMS. Only the most-authorized users should be permitted to modify catalog tables. To improve security and reliability, the data dictionary is usually a separate database stored independently of user databases.

A DBA can query the catalog tables through proprietary interfaces and SELECT statements. Proprietary interfaces such as the Table Definition window of Microsoft Access and the Oracle Enterprise Manager are easier to use than SQL but are not portable across DBMSs. SELECT statements provide more control over the information retrieved than do proprietary interfaces.

Information Resource Dictionary
a database of metadata that describes the entire information systems life cycle. The information resource dictionary system manages access to an IRD.

Information Resource Dictionary An information resource dictionary contains a much broader collection of metadata than does a data dictionary for a DBMS. An information resource dictionary (IRD) contains metadata about individual databases,

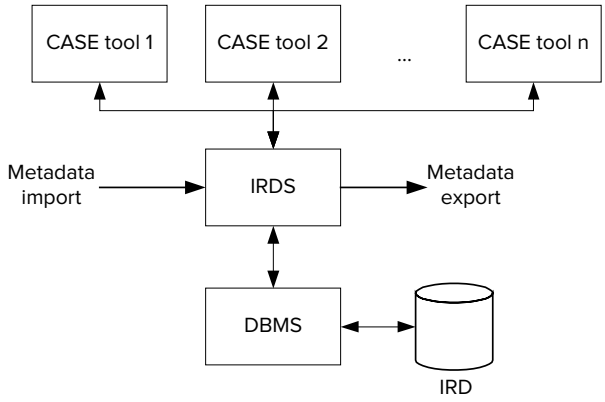
TABLE 16-10
Summary of Important Catalog Tables in SQL:2016

Table	Contents
USERS	One row for each user
DOMAINS	One row for each domain
DOMAIN_CONSTRAINTS	One row for each domain constraint on a table
TABLES	One row for each table and view
VIEWS	One row for each view
COLUMNS	One row for each column
TABLE_CONSTRAINTS	One row for each table constraint
REFERENTIAL_CONSTRAINTS	One row for each referential constraint

TABLE 16-11
Common Catalog Tables for Oracle

Table Name	Contents
USER_CATALOG	Contains basic data about each table and view defined by a user.
USER_OBJECTS	Contains data about each object (functions, procedures, indexes, triggers, etc.) defined by a user. This table contains the time created and the last time changed for each object.
USER_TABLES	Contains extended data about each table such as space allocation and statistical summaries.
USER_TAB_COLUMNS	Contains basic and extended data for each column such as the column name, the table reference, the data type, and a statistical summary.
USER_VIEWS	Contains the SQL statement defining each view.

FIGURE 16.7
IRDS Architecture



computerized and human processes, configuration management, version control, human resources, and the computing environment. Conceptually, an IRD defines metadata used throughout the information systems life cycle. Both DBAs and DAs can use an IRD to manage information resources. In addition, other information systems professionals can use an IRD during selected tasks in the information systems life cycle.

Because of its broader role, an IRD is not consulted by a DBMS to conduct operations. Rather, an information resource dictionary system (IRDS) manages an IRD. Many CASE tools can use the IRDS to access an IRD as depicted in Figure 16.7. CASE tools can access an IRD directly through the IRDS or indirectly through the import/export feature. The IRD has an open architecture so that CASE tools can customize and extend its conceptual schema.

There are two primary proposals for the IRD and the IRDS. The IRD and the IRDS were originally developed as standards by the International Standards Organization (ISO) in the early 1990s. The implementation of the standards, however, was limited. Microsoft and Texas Instruments jointly developed the Microsoft Repository, which supported many of the goals of the IRD and the IRDS although it did not conform to the standard. However, the Microsoft Repository has been phased out after initially gaining some acceptance among CASE tool vendors.

As an alternative to the IRD and IRDS, the Object Management Group (OMG) developed the Model Driven Architecture (MDA) in the early 2000s. The MDA provides an open specification that supports formal modeling of all aspects of the software life cycle including business processes, software architectures, data warehousing, metadata repositories, tool integration and even the software development process itself. The MDA uses multiple standards, including the Unified Modeling Language (UML), the Meta-Object Facility (MOF), XML Metadata Interchange (XMI), Enterprise Distributed Object Computing (EDOC), the Software Process Engineering Metamodel (SPEM), and the Common Warehouse Metamodel (CWM). The OMG relies on commercial and open source developers to implement tools for the MDA. However, the MDA has not gained enough commercial acceptance to be considered an important standard.

Thus, the IRD and IRDS remain idealized concepts without a widely accepted commercial standard. Data specialists must deal with proprietary data dictionary interfaces in commercial CASE tools to compliment the dictionary tables available with enterprise DBMSs.

16.3 PROCESSES FOR DATABASE SPECIALISTS

This section describes processes performed by data administrators and database administrators. Data administrators perform data planning as part of the information systems planning process. Data administrators participate in data governance

sometimes serving on the data governance committee and other times consulting on activities managed by the data governance committee. Both data administrators and database administrators may perform tasks in the process of selecting and evaluating DBMSs. This section presents the details of all three processes.

16.3.1 Data Planning

Despite large expenditures on information technology, many organizations feel disappointed in the payoff. Many organizations have created islands of automation that support local objectives but not the global objectives for the organization. The islands-of-automation approach can lead to a misalignment of the business and information technology objectives. One result of the misalignment is the difficulty in extracting the decision-making value from operational databases.

As a response to problems with islands of automation, many organizations perform a detailed planning process for information technology and systems. The planning process is known under various names such as **information systems planning**, business systems planning, information systems engineering, and information systems architecture. All of these approaches provide a process to achieve the following objectives:

Information Systems Planning
the process of developing enterprise models of data, processes, and organizational roles. Information systems planning evaluates existing systems, identifies opportunities to apply information technology for competitive advantage, and plans new systems.

- Evaluation of current information systems with respect to the goals and objectives of the organization
- Determination of the scope and the timing of developing new information systems and utilization of new information technology
- Identification of opportunities to apply information technology for competitive advantage

The information systems planning process involves the development of enterprise models of data, processes, and organizational roles, as depicted in Figure 16.8. In the first part of the planning process, broad models are developed. Table 16-12 shows the initial level of detail for the data, process, and organization models. Because the enterprise data model is usually more stable than the process model, it is usually developed first. To integrate these models, interaction models are developed as shown in Table 16-12. If additional detail is desired, the process and the data models are further expanded. These models should reflect the current information systems infrastructure as well as planned future directions.

Data administrators play an important part in the development of information system plans. Data administrators conduct numerous interviews to develop the enterprise data model and coordinate with other planning personnel to develop the interaction models. To improve the likelihood that plans will be accepted and used, data administrators should involve senior management. By emphasizing the decision-making potential of integrated information systems, senior management will be motivated to support the planning process.

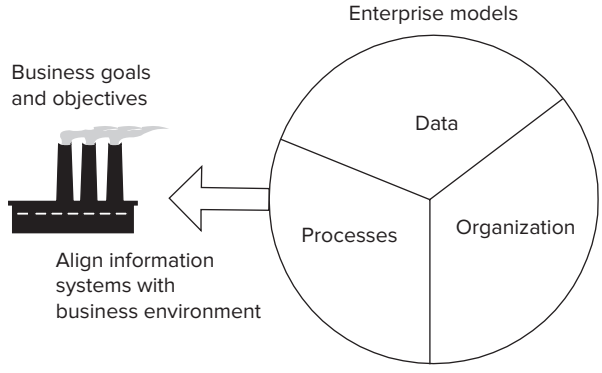


FIGURE 16.8
Enterprise Models Developed in the Information Systems Planning Process

TABLE 16-12
Level of Detail of Enterprise Models

Model	Levels of Detail
Data	Subject model (initial level), entity model (detailed level)
Process	Functional areas and business processes (initial level), activity model (detailed level)
Organization	Role definitions and role relationships
Data-process interaction	Matrix and diagrams showing data requirements of processes
Process-organization interaction	Matrix and diagrams showing role responsibilities
Data-organization	Matrix and diagrams showing usage of data by roles

16.3.2 Data Governance Processes and Tools

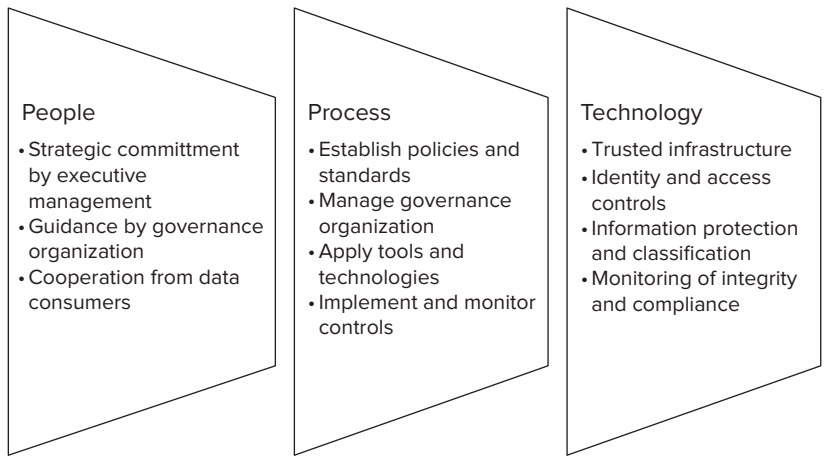
Data governance overlaps somewhat with data planning. Data governance emphasizes controls and accountability while data planning emphasizes enterprise models to guide usage and development of information technology. For example, data planning involves models of an organization and data usage within an organization but does not typically involve the controls and compliance representation in data governance models.

Microsoft and the Data Governance Institute have developed data governance approaches. The Microsoft approach provides some useful modeling tools so its approach is covered in this section. Otherwise, the approaches are reasonably similar. For more details about both approaches, you should see references listed in this section and at the end of the chapter.

The Microsoft Data Governance for Privacy, Confidentiality, and Compliance (DGPC) Framework contains components[5] for people, process, and technology as depicted in Figure 16.9. The people part of the framework involves a pyramid with a small group of executive management providing strategic direction to a data governance organization. The data governance organization contains a diverse group of data stewards organized into steering committees and working groups. A **data steward**, typically a manager of a business area or a subject area expert, assumes responsibility for policies and standards applying to selected data. At the bottom of the pyramid, data consumers adhere to the policies and standards established by the data governance organization. Data consumers also report on violations of policies and standards for corrective action performed by data stewards and the data governance organization.

Data Steward
an individual responsible for policies and standards applying to selected data. Typically managers from business areas or experts from subject areas serve as data stewards.

FIGURE 16.9
Components of the Microsoft DGPC Framework (Adapted from Salido and Voon, 2010, Part 2)



[5] Salido, J. and Voon, P. "A Guide to Data Governance for Privacy, Confidentiality, and Compliance (Part 2): People and Process," Microsoft Corporation, Whitepaper, January 2010.

The DGPC Framework contains four core processes for managing the governance organization, requirements, strategies/policies, and controls as depicted in Figure 16.10. Managing the data governance organization involves processes for appointing members, defining roles and responsibilities, creating working groups, and reporting status and performance of data governance initiatives. Managing requirements involves translating business strategy into data quality and compliance requirements and collecting and integrating authority documents including regulations, standards, and policies. Managing strategies and policies involves processes to review, approve, publish, and implement authority documents. Managing the control environment involves processes for developing and monitoring controls and using tools to model information lifecycles and technology domains. The Microsoft information cycle is similar to the lifecycle shown in Figure 16.2. Table 16-13 lists functions of the four key technology domains: secure infrastructure, identity and access controls, information protection, and auditing and reporting.

Controls, tools for managing risks to data assets, can be classified according to their timing (preventative, detective, and corrective) and automation level (manual, technology-aided, and automatic). Preventative controls are applied before an event occurs to ensure compliance. Typical preventative controls are segregation of duties and approval levels. Detective controls are applied after an event occurs to determine errors and irregularities. Exception reports, reconciliations, and audits are typical detective controls. Corrective controls are applied after detection of errors and noncompliance. Malware removal and backup restore are typical corrective controls. The

Control
a tool to manage risks to data assets. Controls can be classified according to their timing (preventative, detective, and corrective) and automation level (manual, technology-aided, and automatic).

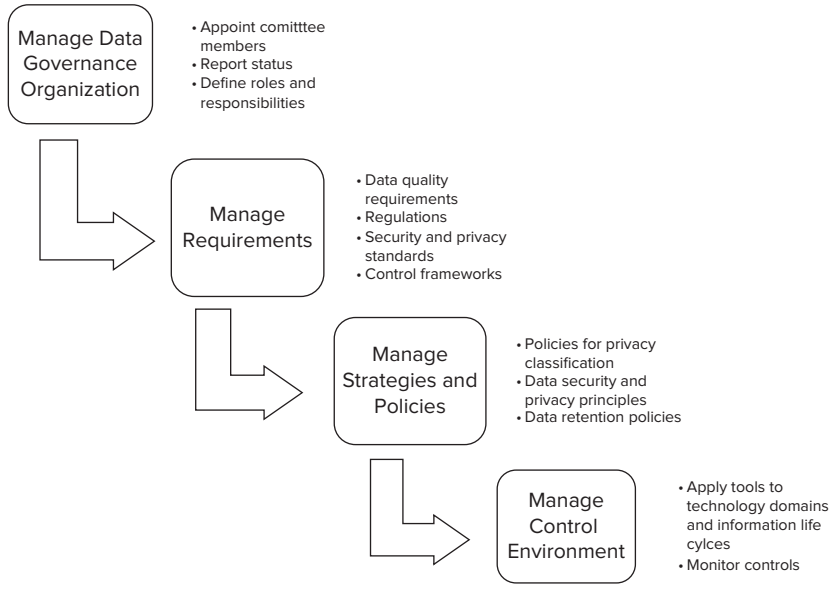


FIGURE 16.10
Core Processes in the Microsoft DGPC Framework (Adapted from Salido and Voon, 2010, Part 2)

Security Domain	Functions
Secure infrastructure	Stop malware and intrusions Monitor systems from evolving threats
Identity and access control	Safeguard sensitive data from unauthorized access or use Support controls for identity, access, and provisioning
Information protection	Protect sensitive data in databases, documents, messages, and records Safeguard data in motion Automate data classification for privacy
Auditing and reporting	Monitor integrity of systems and data Monitor compliance of data privacy and confidentiality with policies

TABLE 16-13
List of Functions in Microsoft Security Domains

Adapted from Salido and Voon, 2010, Part 2

level of automation depends on the type of control. Corrective controls are easiest to automate. Some human involvement is usually necessary for preventative and detective controls to perform procedures or review control results.

The Risk-Gap Analysis Matrix[6] combines the information lifecycle and technology domains. The rows of the matrix are the data lifecycle stages while the columns are the technology domains as shown in Table 16-14. The last column identifies manual controls, not part of the technology domains. The cells show gaps in measures to protect data and manage risks for combinations of lifecycle activity and security domain. Gaps should be evaluated using compliance, data privacy, and confidentiality principles applicable to the combination of a lifecycle stage and technology domain.

Microsoft provides a risk-gap analysis process to support identification and resolution of gaps as depicted in Figure 16.11. In the first step, the context is established through defining the business purpose and listing the specific privacy, security, and compliance objectives. In the second step, threats are identified using a threat modeling tool such as the Microsoft Security Development Lifecycle tool. In the third step, risks for each cell of the risk-gap matrix are determined using the security and privacy principles. In the fourth step, risk mitigation techniques are chosen subject to cost-benefit analyses. In the fifth step, the effectiveness of mitigation techniques are evaluated using data collected from data consumers.

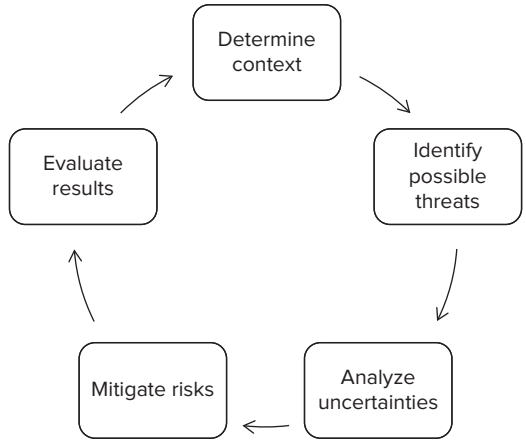
To gauge an organization's relative progress in data governance, Microsoft provides the DGPC Capability Maturity Model (CMM)[7]. The CMM helps organizations

TABLE 16-14
Template Risk-Gap Matrix in the Microsoft DGPC Framework

	Technology Domain				
Lifecycle Stage	**Secure Infrastructure**	**Identity and Access Control**	**Information Protection**	**Auditing and Reporting**	**Manual Controls**
Collect					
Update					
Process					
Delete					
Storage					
Transfer					

Adapted from Salido and Voon, 2010, Part 3

FIGURE 16.11
Microsoft Risk Gap Analysis Process
(Adapted from Salido and Voon, 2010, Part 3)



[6] Salido, J. and Voon, P. "A Guide to Data Governance for Privacy, Confidentiality, and Compliance (Part 3): Managing Technological Risk," Microsoft Corporation, Whitepaper, March 2010.
[7] Salido, J. and Voon, P. "A Guide to Data Governance for Privacy, Confidentiality, and Compliance (Part 4): A Capability Maturity Model," Microsoft Corporation, Whitepaper, April 2010.

determine their current status, target future status goals, and create action plans to reach maturity targets. The CMM provides a timeline showing stages of maturity, a development process to determine activities to reach future target maturity targets, and a table with details about capabilities in each core area. As depicted in Table 16-15, the stages of the CMM depend on the levels of training for personnel, development and integration of processes, and automation and integration of controls. The CMM development process involves understanding CMM capabilities, determining target levels, prioritizing development of new capabilities, and measuring progress. The main artifact of the CMM development process is a detailed table with main sections for people, process, and technology. Each area is evaluated on the four maturity levels with capabilities noted as current, planned, in progress, delivered, and adopted.

To support the DGPC for Microsoft Office 365, Microsoft provides some software tools. The initial release of Office 365 Advanced Data Governance supports automated recommendations of retention policies, data profiling for retention, and detective controls for suspicious user interactions. Policy recommendations involve retention and deletion rules consistent with organization and industry norms. Data profiling classifies data by type, age, and user interaction. Detective controls provide standard alerts indicating unusual volumes of file operations and custom alerts with customized matching conditions and thresholds.

Data administrators and database administrators are usually involved in data governance. Data administrators participate through membership in the data governance office and consulting on activities managed by the data governance office. With their broad outlook on corporate data assets and interaction with various data stewards, data administrators are well-suited to manage a data governance office. Data administrators can also be heavily involved in setting policies for data definitions, data quality, and controls. Database administrators usually serve a support role to help implement technology-aided and automated controls.

16.3.3 Selection and Evaluation of Database Management Systems

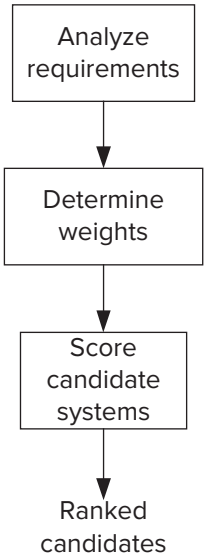
Selection and evaluation of a DBMS can be a very important task for an organization. DBMSs provide an important part of the computing infrastructure. As organizations strive to conduct electronic commerce over the Internet and extract value from operational databases, DBMSs play an even greater role. The selection and evaluation process is important because of the impacts of a poor choice. The immediate impacts may be slow database performance and loss of the purchase price. A poorly performing information system can cause lost sales and higher costs. The longer-term impacts are high switching costs. To switch DBMSs, an organization may need to convert data, recode software, and retrain employees. The switching costs can be much larger than the original purchase price.

TABLE 16-15
Stages of the Microsoft Capability Maturity Model

Stage	People	Process	Technology
Basic	Lack of training and awareness	Few, immature processes	Lack of tools without integration
Standardized	Formal training	Established and communicated processes	Minimal tools for foundation goals
Rationalized	Formal training with compliance metrics	Process improvement	Increased usage of automated controls and some integration
Dynamic	Formal training with compliance metrics	Integrated compliance efforts	Full usage of automated and technology-aided controls with integration

Adapted from Salido and Voon, 2010, Part 4

FIGURE 16.12
Overview of the Selection and Evaluation Process



Selection and Evaluation Process The selection and evaluation process involves a detailed assessment of an organization's needs and features of candidate DBMSs. The goal of the process is to determine a small set of candidate systems that will be investigated in more detail. Because of the detailed nature of the process, a DBA performs most of the tasks. Therefore, a DBA needs a thorough knowledge of DBMSs to perform the process.

Figure 16.12 depicts the steps of the selection and evaluation process. In the first step, a DBA conducts a detailed analysis of the requirements. Because of the large number of requirements, it is helpful to group them. Table 16-16 lists major groupings of requirements while Table 16-17 shows some individual requirements in one group. Each individual requirement should be classified as essential, desirable, or optional to the requirement group. In some cases, several levels of requirements may be necessary. A DBA should be able to objectively measure individual requirements in candidate systems.

After determining the groupings, the DBA should assign weights to the major requirement groups and score candidate systems. With more than a few major requirement groups, assigning consistent weights is very difficult. The DBA needs a tool to help assign consistent weights and to score candidate systems. The **Analytic Hierarchy Process** (AHP) provides a simple approach that achieves a reasonable level of consistency. The AHP has been used in a variety of management decision-making situations. As evidence of AHP's acceptance, a number of commercial software tools support the AHP.

Analytic Hierarchy Process a decision theory technique to evaluate problems with multiple objectives. The process supports the selection and evaluation process by allowing a systematic assignment of weights to requirements and scores to features of candidate DBMSs.

Using the AHP, you assign weights to pairwise combinations of requirement groups using the nine-point scale in Table 16-18. For example, you should assign a weight that represents the importance of conceptual data definition as compared to nonprocedural retrieval. As shown in Table 16-19, the ranking of 5 in row 2, column 1 means that nonprocedural retrieval is significantly more important than conceptual data definition. For consistency, if entry $A_{ij} = x$, then $A_{ji} = 1/x$. Thus, the entry in row 1, column 2 is 1/5. The diagonal elements (comparing a requirement group to itself) should always be 1. Thus, it is necessary to complete only half the rankings in Table 16-19. The final row in the Table 16-19 shows column sums used to normalize weights and determine importance values.

After assigning pairwise weights to the requirement groups, the weights are combined to determine an importance weight for each requirement group. The cell values are normalized by dividing each cell by its column sum as shown in Table 16-20. The final importance value for each requirement group is the average of the normalized weights in each row as shown in Table 16-21.

TABLE 16-16
Some Major Requirement Groups

Category
Data definition (conceptual)
Nonprocedural retrieval
Data definition (internal)
Application development
Procedural language
Concurrency control
Recovery management
Parallel database processing
Distributed database support
Vendor support
Query optimization

Requirement (Importance)	Explanation
Entity integrity (essential)	Declaration and enforcement of primary keys
Candidate keys (desirable)	Declaration and enforcement of candidate keys
Referential integrity (essential)	Declaration and enforcement of referential integrity
Referenced rows (desirable)	Declaration and enforcement of rules for referenced rows
Standard data types (essential)	Support for whole numbers (several sizes), floating-point numbers (several sizes), fixed-point numbers, fixed-length strings, variable-length strings, and dates (date, time, and timestamp)
User-defined data types (desirable)	Support for new data types or a menu of optional data types
User interface (desirable)	Graphical user interface to manipulate dictionary tables
General assertions (optional)	Declaration and enforcement of multitable constraints
CHECK constraints (essential)	Declaration and enforcement of intra table constraints

TABLE 16-17
Some Detailed Requirements for the Conceptual Data Definition Category

Ranking Value of A_{ij}	Meaning
1	Requirements *i* and *j* are equally important.
3	Requirement *i* is slightly more important than requirement *j*.
5	Requirement *i* is significantly more important than requirement *j*.
7	Requirement *i* is very significantly more important than requirement *j*.
9	Requirement *i* is extremely more important than requirement *j*.

TABLE 16-18
Interpretation of Rating Values for Pairwise Comparisons

	Data Definition (Conceptual)	Nonprocedural Retrieval	Application Development	Concurrency Control
Data Definition (conceptual)	1	1/5 (0.20)	1/3 (0.33)	1/7 (0.14)
Nonprocedural Retrieval	5	1	3	1/3 (0.33)
Application Development	3	1/3 (0.33)	1	1/5 (0.20)
Concurrency Control	7	3	5	1
Column Sum	16	4.53	9.33	1.67

TABLE 16-19
Sample Weights for Some Requirement Groups

	Data Definition (Conceptual)	Nonprocedural Retrieval	Application Development	Concurrency Control
Data Definition (conceptual)	0.06	0.04	0.04	0.08
Nonprocedural Retrieval	0.31	0.22	0.32	0.20
Application Development	0.19	0.07	0.11	0.12
Concurrency Control	0.44	0.66	0.54	0.60

TABLE 16-20
Normalized Weights for Some Requirement Groups

Requirement Group	Importance
Data Definition (conceptual)	0.06
Nonprocedural Retrieval	0.26
Application Development	0.12
Concurrency Control	0.56

TABLE 16-21
Importance Values for Some Requirement Groups

Importance weights must be computed for each subcategory of requirement groups in the same manner as for requirement groups. For each subcategory, pairwise weights are assigned before normalizing the weights and computing final importance values.

After computing importance values for the requirements, candidate DBMSs are assigned scores. Scoring candidate DBMSs can be complex because of the number of individual requirements and the need to combine individual requirements into an overall score for the requirement group. As the first part of the scoring process, a DBA should carefully investigate the features of each candidate DBMS.

Many approaches have been proposed to combine individual feature scores into an overall score for the requirement group. The Analytic Hierarchy Process supports pairwise comparisons among candidate DBMSs using the rating values in Table 16-18. The interpretations change slightly to reflect comparisons among candidate DBMSs rather than the importance of requirement groups. For example, a value 3 should be assigned if DBMS *i* is slightly better than DBMS *j*. For each requirement subcategory, a comparison matrix should be created to compare the candidate DBMSs. Scores for each DBMS are computed by normalizing the weights and computing the row averages as for requirement groups.

After scoring the candidate DBMSs for each requirement group, the final scores are computed by combining the requirement group scores with the importance of requirement groups. For details about computing the final scores, you should consult the references at the end of the chapter about the Analytic Hierarchy Process.

DBMS Evaluation using Benchmarks After the selection and evaluation process completes, the top two or three candidate DBMSs should be evaluated in more detail. Benchmarks can be used to provide a more detailed evaluation of candidate DBMSs. A **benchmark** is a workload to evaluate the performance of a system or product. A good benchmark should be relevant, portable, scalable, and understandable. Because developing good benchmarks requires significant expertise, most organizations should not attempt to develop a benchmark. Fortunately, the Transaction Processing Performance Council (TPC) has developed a number of standard, enterprise benchmarks as summarized in Table 16-22. Each enterprise benchmark was developed over an extended time period with input from a diverse group of contributors.

Benchmark
a workload to evaluate the performance of a system or product. A good benchmark should be relevant, portable, scalable, and understandable.

The TPC benchmarks have been extended with specifications for pricing, energy efficiency, and virtualization measurement. The pricing specification was developed

TABLE 16-22
Summary of TPC Enterprise[8] Benchmarks

Benchmark	Description	Performance Measures
TPC-C	Online order entry benchmark	Transactions per minute, price per transactions per minute
TPC-DI	Data integration benchmark for a fictitious brokerage firm with multiple data sources	Combined throughput (elapsed time) measure for historical load and incremental update, price per throughput
TPC-DS	Decision support benchmark for a retail product supplier; uses both queries and data integration processing	Effective query throughput (queries per elapsed time), price per query throughput, system availability date
TPC-E	Transaction workload of a brokerage firm	Transactions per second, price per transactions per second
TPC-H	Decision support for ad hoc queries	Composite queries per hour, price per composite queries per hour
TPC-VMS	Virtualized database benchmark for base benchmarks (TPC-C, TPC-DS, TPC-E, or TPC-H)	Minimum base benchmark performance, price per base benchmark performance, system availability date

[8] In addition to the currently supported enterprise benchmarks, the TPC has three express benchmarks and six obsolete benchmarks.

to reduce confusion caused by special pricing policies offered by vendors especially benchmark pricing. Vendor prices should be generally available, published prices for commercial products to evaluate price/performance for a three year period. Each vendor must disclose the purchase price of associated hardware, software licensing costs, and maintenance contracts. The pricing specification stipulates auditing requirements for the full disclosure agreement about prices provided in a benchmark result.

In response to the growing importance of energy efficiency in information technology selection, the TPC developed the TPC-Energy specification. Customers have increasingly demanded both price/performance and energy/performance results for information technology purchasing decisions. The TPC-Energy specification provides standards for energy metrics and a software tool to support energy measurement and reporting by benchmark vendors. The energy metric standard stipulates components of the system under test and aspects of the physical environment (such as temperature, humidity, and altitude) for execution of a benchmark. The Energy Metric System supports a Web interface for power system instrumentation, power and temperature logging, and report generation.

The virtualized database benchmark (TPC-VMS) supports benchmark execution and reporting on virtualized databases. Organizations increasingly utilize virtualization environments leading to demands for benchmark measurements for virtualized databases.

A DBA can use the TPC results to obtain reasonable estimates about the performance of a DBMS in a specific hardware/software environment. The TPC performance results involve total system performance, not just DBMS performance so that results are not inflated when a customer uses a DBMS in a specific hardware/software environment. To facilitate price performance trade-offs, the TPC publishes the performance measure along with price/performance for each benchmark. The price covers all cost dimensions of an entire system environment including workstations, communications equipment, system software, computer system or host, backup storage, and three years' maintenance cost. The TPC audits the benchmark results prior to publication to ensure that vendors have not manipulated results.

To augment the published TPC results, an organization may want to evaluate a DBMS on a trial basis. Customized benchmarks can be created to gauge the efficiency of a DBMS for its intended usage. In addition, the user interface and the application development capabilities can be evaluated by building small applications.

Final Selection Process The final phase of the selection process typically involves nontechnical considerations performed by data administrators along with senior management and legal staff. Assessment of each vendor's future prospects is important because information systems can have a long life. If the underlying DBMS does not advance with the industry, it may not support future initiatives and upgrades to the information systems that use it. Because of the high fixed and variable costs (maintenance fees) of a DBMS, negotiation is often a critical element of the final selection process. The final contract terms along with one or two key advantages often make the difference in the final selection.

Open source DBMS software is a recent development that complicates selection and evaluation decisions. Organizations considering open source DBMS software should understand distinctions between **open source and open core license models**. An open source license grants users the right to use, copy, share, inspect, and alter software without payment. License variations place conditions in which software can be shared and altered. Organizations using open source DBMS software (such as PostgreSQL) typically pay for product support from an external vendor although the DBMS does not have any license costs.

Open source versus open core license model
an open source license grants users rights to use, copy, share, inspect, and alter software without license payments. An open core model involves a core product under an open source license and an enhanced product under a commercial license.

In contrast, the open core license model provides a core software product under an open source license and an enhanced version as a commercial product. The commercial product in an open core license typically has a reduced price as compared to traditional commercial software. Essentially the open core model provides a way

to monetize open source software development. As an example, MySQL uses an open core license with a core product available under an open source license and an enhanced product under a commercial license.

For open source DBMSs, the lack of profit motive may hinder future product development. Usage of DBMS software for mission critical systems can lead to high switching costs if an organization's needs change and DBMS product development stalls. The open core model provides organizations an upgrade path with a free core product and a commercial product with more certain future development. Despite this uncertainty, many organizations utilize open source DBMS software especially for non-mission-critical systems.

16.4 MANAGING DATABASE ENVIRONMENTS

DBMSs operate in several different processing environments. Data specialists must understand the environments to ensure adequate database performance and set standards and policies. This section provides an overview of the processing environments with an emphasis on the tasks performed by database administrators and data administrators. Chapters 12 to 15 provided details about data warehouse environments. The other chapters in Part 7 provide the details about the other processing environments.

16.4.1 Transaction Processing

Transaction processing involves the daily operations of an organization. Every day, organizations process large volumes of orders, payments, cash withdrawals, airline reservations, insurance claims, and other kinds of transactions. DBMSs provide essential services to perform transactions in an efficient and reliable manner. Organizations such as banks with automatic tellers, airlines with online reservation systems, and universities with online registration could not function without reliable and efficient transaction processing. With exploding interest to conduct business over the Internet, the importance of transaction processing will grow even larger.

Data specialists have many responsibilities for transaction processing, as listed in Table 16-23. Data administrators may perform planning responsibilities involving infrastructure and disaster recovery. Database administrators usually perform the more detailed tasks such as consulting on transaction design and monitoring performance. Because of the importance of transaction processing, database administrators often must be on call to troubleshoot problems. Chapter 17 presents details of transaction processing for concurrency control and recovery management. After you read Chapter 17, you may want to review Table 16-23 again.

16.4.2 Data Warehouse Processing

As you learned in Chapters 12 to 15, data warehouses support business intelligence requirements. Because many organizations have not been able to use operational databases directly to support management decision making, the idea of a data

TABLE 16-23
Responsibilities of Database Specialists for Transaction Processing

Area	Responsibilities
Transaction design	Consult about design to balance integrity and performance; educate about design issues and DBMS features
Performance monitoring	Monitor transaction performance and troubleshoot performance problems; modify resource levels to improve performance
Transaction processing infrastructure	Determine resource levels of disks, memory, RAID devices, and servers for efficient and reliable processing
Disaster recovery	Provide contingency plans for various kinds of database failures

warehouse was conceived. A data warehouse is a logically centralized database in which enterprise-wide data are stored to facilitate business intelligence activities by user departments. Data from operational databases and external sources are extracted, cleaned, integrated, and then loaded into a data warehouse. Because a data warehouse contains secondary data, most activity involves retrievals of summarized data.

Data specialists have many responsibilities for data warehouses, as listed in Table 16-24. Data administrators may perform planning responsibilities involving the data warehouse architecture and the enterprise data model. Database administrators usually perform the more detailed tasks such as data warehouse design, performance monitoring, and consulting. To support a large data warehouse, additional software products separate from a DBMS may be necessary for data integration, reporting, and analytical processing. A selection and evaluation process should be conducted to choose the most appropriate product for each need. Chapters 12 to 15 presented details of data warehouses. After you read Chapters 12 and 15, you may want to review Table 16-24 again.

16.4.3 Distributed Environments

DBMSs can operate in distributed environments to support both transaction processing and data warehouses. In distributed environments, DBMSs can provide the ability to distribute processing and data among computers connected by a network. For distributed processing, a DBMS may allow the distribution of functions provided by the DBMS as well as application processing to be distributed among different computers in a network. For distributed data, a DBMS may allow tables to be stored and possibly replicated at different sites in a network. The ability to distribute processing and data provides the promise of improved flexibility, scalability, performance, and reliability. However, these improvements only can be obtained through careful design.

Data specialists have many responsibilities for distributed database environments, as shown in Table 16-25. Data administrators usually perform planning responsibilities involving setting goals and determining architectures. Because distributed environments do not increase functionality, they must be justified by improvements in the underlying operations. Database administrators perform more detailed tasks such as performance monitoring and distributed database design. To support distributed environments, other software products along with major extensions to a DBMS may be necessary. A selection and evaluation process should be conducted to choose the most appropriate products. Chapter 18 presents the details of distributed processing and distributed data. After you read Chapter 18, you may want to review Table 16-25 again.

TABLE 16-24
Responsibilities of Database Specialists for Data Warehouses

Area	Responsibilities
Data warehouse usage	Educate and consult about application design and DBMS features for data warehouse processing
Performance monitoring	Monitor data warehouse refresh and query performance; troubleshoot integrity problems; modify resource levels to improve performance
Data warehouse refresh	Determine the frequency of refreshing the data warehouse and the schedule of activities to refresh the data warehouse; design and implement data integration workflows and procedures
Data warehouse architecture	Determine architecture to support decision-making needs; select database and data integration products to support architecture determine resource levels for efficient processing
Enterprise data model	Provide expertise about operational database content; design conceptual data models for data warehouses; promote data quality to support data warehouse development

TABLE 16-25
Responsibilities of Database Specialists for Distributed Environments

Area	Responsibilities
Application development	Educate and consult about impacts of distributed environments for transaction processing and data warehouses
Performance monitoring	Monitor performance and troubleshoot problems with a special emphasis on distributed environments
Distributed environment architectures	Identify goals for distributed environments; choose distributed processing, parallel database, big data, and distributed database architectures to meet goals; select additional software products to support architectures
Distributed environment design	Design distributed databases; determine resource levels for efficient processing

16.4.4 Object Databases and NoSQL Databases

Object DBMSs support additional functionality for transaction processing and business intelligence processing. Many information systems use a richer set of data types than provided by relational DBMSs. For example, many financial databases need to manipulate time series, a data type not provided by most relational DBMSs. With the ability to convert any kind of data to a digital format, the need for new data types is even more pronounced. Business databases often need to integrate traditional data with nontraditional data based on new data types. For example, information systems for processing insurance claims must manage traditional data such as account numbers, claim amounts, and accident dates as well as nontraditional data such as images, maps, and drawings. Because of these needs, existing relational DBMSs have been extended with object capabilities.

NoSQL (Not Only SQL) databases are the opposite side of object database management. NoSQL databases support simplified and flexible data representations, limited constraint checking, non-standard query languages, and highly efficient processing for emerging big data applications. The market for NoSQL DBMSs has grown rapidly from a small base as organizations invest in big data applications.

Data specialists have many responsibilities for object databases and NoSQL databases as shown in Table 16-26. Data administrators usually perform planning responsibilities involving setting goals and determining architectures. Database administrators perform more detailed tasks such as performance monitoring, consulting, and database design. An object DBMS extends an existing relational DBMS. A NoSQL database can extend an existing DBMS or be a new DBMS without relational database features. A selection and evaluation process should be conducted to choose the most appropriate product. Chapter 19 presents the details of object and NoSQL DBMSs. After you read Chapter 19, you may want to review Table 16-26 again.

TABLE 16-26
Responsibilities of Database Specialists for Object and NoSQL Databases

Area	Responsibilities
Application development	Educate and consult about extended data types, creating new data types, inheritance for data types and tables, and other object features as well as simplified data representations in NoSQL databases
Performance monitoring	Monitor performance and troubleshoot problems with new data types for object databases and processing approaches for NoSQL databases
NoSQL database architectures	Choose data representation and processing approaches; Identify applications that require NoSQL technology
Object database design	Design object databases; select data types; create new data types, design functions and procedures for new data types; For NoSQL databases, convert conventional design into simplified NoSQL representation

CLOSING THOUGHTS

This chapter has described the responsibilities, tools, and processes used by data specialists to manage databases and support management decision making. Many organizations provide two roles for managing information resources. Data administrators perform broad planning and policy setting, while database administrators perform detailed oversight of individual databases and DBMSs. To provide a context to understand the responsibilities of data specialists, this chapter described frameworks for managing organizational information and knowledge as well as governance of information resources. In many organizations, data specialists participate in these frameworks in an environment dominated by challenges and opportunities from exploding data growth.

This chapter described a number of tools to support database administrators. Database administrators use security rules to restrict access and integrity constraints to improve data quality. This chapter described security rules and integrity constraints along with associated SQL:2016 syntax and additional Oracle security features. For triggers and stored procedures, this chapter described managerial responsibilities of DBAs to complement the coding details in Chapter 11. The data dictionary is an important tool for managing individual databases as well as integrating database development with information systems development. This chapter presented two kinds of data dictionaries: catalog tables used by DBMSs and the information resource dictionary used by CASE tools.

Database specialists need to understand three important processes to manage information technology. Data administrators participate in a detailed planning process that determines new directions for information systems development. This chapter described the data planning process as an important component of the information systems planning process. Data administrators often serve in key roles in data governance, a process to control risks to data assets and improve compliance with privacy and confidentiality policies. Database administrators may support data governance by helping to implement technology-aided controls. This chapter described the Microsoft Framework for Data Governance, Privacy, Confidentiality, and Compliance, a prominent approach involving people, processes, and technology components to support data governance activities. Both data administrators and database administrators participate in the selection and evaluation of DBMSs. Database administrators perform detailed tasks while data administrators often make final selection decisions based on a detailed recommendation and negotiation with vendors. This chapter described the steps of the selection and evaluation process and the tasks performed by database administrators and data administrators in the process.

This chapter provides a context for the data warehouse chapters (Chapters 12 to 15) and other chapters in Part 7. The other chapters provide details about different database environments including transaction processing, distributed environments, and object and NoSQL databases. This chapter has emphasized the responsibilities, tools, and processes of database specialists for managing these environments. After completing the other chapters in Part 7, you are encouraged to reread this chapter to help you integrate the details with management concepts and techniques.

REVIEW CONCEPTS

- Information resource management: management philosophy to control information resources and apply information technology to support management decision-making
- Data governance involving the application of decision-making and authority for data-related issues

- Database administrator: support position for managing individual databases and DBMSs
- Data administrator: management position with planning and policy responsibilities for information technology
- Challenges and opportunities dealing with big data, the phenomenon of exploding growth of data volumes, velocity, and variety
- Discretionary access controls for assigning access rights to groups and users
- Mandatory access controls for highly sensitive and static databases used in intelligence gathering and national defense
- SQL CREATE/DROP ROLE statements and GRANT/REVOKE statements for discretionary authorization rules

```
CREATE ROLE ISFaculty

GRANT SELECT ON ISStudentGPA
  TO ISFaculty, ISAdvisor, ISAdministrator

REVOKE SELECT ON ISStudentGPA FROM ISFaculty
```

- Oracle system and object privileges for discretionary access control
- Advanced Oracle security tools for fine grained control of data access, auditing, and resource limitations
- SQL CREATE DOMAIN statement for data type constraints

```
CREATE DOMAIN StudentClass AS CHAR(2)
  CHECK (VALUE IN ('FR', 'SO', 'JR', 'SR') )
```

- SQL distinct types for improved type checking
- Limitations of SQL domains and distinct types compared to user-defined data types
- SQL CREATE ASSERTION statement for complex integrity constraints

```
CREATE ASSERTION OfferingConflict
 CHECK (NOT EXISTS
  ( SELECT O1.OfferNo
      FROM Offering O1, Offering O2
      WHERE O1.OfferNo <> O2.OfferNo
        AND O1.OffTerm = O2.OffTerm
        AND O1.OffYear = O2.OffYear
        AND O1.OffDays = O2.OffDays
        AND O1.OffTime = O2.OffTime
        AND O1.OffLocation = O2.OffLocation  ) )
```

- CHECK constraints in the CREATE TABLE statement for constraints involving row conditions on columns of the same table

```
CREATE TABLE Student
( StdSSN      CHAR(11),
  StdFirstName VARCHAR(50) CONSTRAINT StdFirstNameRequired
                           NOT NULL,
  StdLastName  VARCHAR(50) CONSTRAINT StdLastNameRequired
                           NOT NULL,
  StdCity      VARCHAR(50) CONSTRAINT StdCityRequired NOT
                           NULL,
  StdState     CHAR(2)     CONSTRAINT StdStateRequired NOT
                           NULL,
  StdZip       CHAR(9)     CONSTRAINT StdZipRequired NOT
                           NULL,
  StdMajor     CHAR(6),
```

```
  StdClass    CHAR(6),
  StdGPA      DECIMAL(3,2),
CONSTRAINT PKStudent PRIMARY KEY (StdSSN),
CONSTRAINT ValidGPA CHECK ( StdGPA BETWEEN 0 AND 4 ),
CONSTRAINT MajorDeclared CHECK
  ( StdClass IN ('FR','SO') OR StdMajor IS NOT NULL ) )
```

- Management of trigger and procedure coding practices: documentation standards, parameter usage, and content
- Management of object dependencies: access plan obsolescence, modification of referenced objects, deletion of referenced objects
- Controlling trigger complexity: identifying trigger interactions, minimizing trigger actions that can fire other triggers, removing dependence on a specific firing order for overlapping triggers
- Catalog tables for tracking the objects managed by a DBMS
- Information resource dictionary for managing the information systems development process
- Development of an enterprise data model as an important part of the information systems planning process
- Establishment of an organization to develop, implement, and monitor policies and standards for data governance
- Developing controls to detect, prevent, and correct violations of integrity, security, and privacy policies
- Selection and evaluation process for matching organization needs to DBMS features
- Using a tool such as the Analytic Hierarchy Process for consistently assigning importance weights and scoring candidate DBMSs
- Using TPC benchmark results to gauge the performance of DBMSs
- Open source versus open core license models to reduce licensing costs of DBMSs
- Responsibilities of database specialists for managing transaction processing, data warehouses, distributed environments, and object and NoSQL DBMSs

QUESTIONS

1. Why is it difficult to use operational databases for management decision making?
2. How must operational databases be transformed for management decision making?
3. What are the phases of the information life cycle?
4. What does it mean to integrate information life cycles?
5. What data quality dimension is important for management decision making but not for operational decision making?
6. How does knowledge management differ from information resource management?
7. What are the three pillars of knowledge management?
8. What kind of position is the data administrator?
9. What kind of position is the database administrator?
10. Which position (data administrator versus database administrator) takes a broader view of information resources?

11. What is an enterprise data model?
12. For what reasons is an enterprise data model developed?
13. What kinds of specialization are possible in large organizations for data administrators and database administrators?
14. What is discretionary access control?
15. What is mandatory access control?
16. What kind of database requires mandatory access control?
17. What are the purposes of the GRANT and REVOKE statements in SQL?
18. Why should authorization rules reference roles instead of individual users?
19. Why do authorization rules typically use views rather than tables or columns?
20. What are the two uses of the GRANT statement?
21. Why should a DBA cautiously use the WITH ADMIN clause in the CREATE ROLE statement and the WITH GRANT OPTION clause in the GRANT statement?
22. What is the difference between system privileges and object privileges in Oracle? Provide an example of a system privilege and an object privilege.
23. What other disciplines does computer security involve?
24. What is the purpose of the CREATE DOMAIN statement? Compare and contrast an SQL domain with a distinct type.
25. What additional capabilities does SQL:2016 add for user-defined types as compared to domains?
26. What is the purpose of assertions in SQL?
27. What does it mean to say that a constraint is deferrable?
28. What are alternatives to SQL assertions? Why would you use an alternative to an assertion?
29. What are the coding issues about which a DBA should be concerned?
30. How does a stored procedure or trigger depend on other database objects?
31. What are the responsibilities for a DBA for managing dependencies?
32. What is the difference between timestamp and signature dependency maintenance?
33. List at least three ways that a DBA can control trigger interactions.
34. What kind of metadata does a data dictionary contain?
35. What are catalog tables? What kind of catalog tables are managed by DBMSs?
36. What is the difference between the Information_Schema and the Definition_Schema in SQL:2016?
37. Why is it necessary to learn the catalog tables of a specific DBMS?
38. How does a DBA access catalog tables?
39. What is the purpose of an information resource dictionary?
40. What functions does an information resource dictionary system perform?
41. What are the purposes of information systems planning?
42. Why is the enterprise data model developed before the process model?
43. Why is the selection and evaluation process important for DBMSs?
44. What are some difficulties in the selection and evaluation process for a complex product like a DBMS?
45. What are the steps in the selection and evaluation process?
46. How is the Analytic Hierarchy Process used in the selection and evaluation process?

47. What responsibilities does the database administrator have in the selection and evaluation process?
48. What responsibilities does the data administrator have in the selection and evaluation process?
49. What are the responsibilities of database administrators for transaction processing?
50. What are responsibilities of database administrators for managing data warehouses?
51. What are the responsibilities of database administrators for managing databases in distributed environments?
52. What are the responsibilities of database administrators for managing object databases?
53. What are the responsibilities of data administrators for transaction processing?
54. What are the responsibilities of data administrators for managing data warehouses?
55. What are the responsibilities of data administrators for managing databases in distributed environments?
56. What are the responsibilities of data administrators for managing object databases?
57. What are the characteristics of a good benchmark?
58. Why does the Transaction Processing Performance Council publish total system performance measures rather than component measures?
59. Why does the Transaction Processing Performance Council publish price/performance results?
60. How does the Transaction Processing Performance Council ensure that benchmark results are relevant and reliable?
61. What is the status of standards and implementations of the Information Resource Dictionary?
62. Briefly describe the Model Driven Architecture (MDA) and its current status.
63. What are the choices for information resource tools beyond the dictionary tables provided by an enterprise DBMS?
64. Briefly describe the advanced tools provided by Oracle for data security.
65. What is the TPC pricing specification?
66. What is the TPC energy efficiency specification?
67. What is a data steward? What role do data stewards play in data governance?
68. Briefly describe the types of controls and discuss the role of controls in the data governance process.
69. How is the Risk-Gap Matrix used in Microsoft's data governance framework?
70. How is the Capability Maturity Model used in Microsoft's data governance framework?
71. Provide two definitions for big data.
72. List the major sources of big data.
73. Briefly explain the ways that big data creates business opportunities.
74. Provide examples of the units of big data.
75. What is application level security? Why do enterprise DBMSs usually not provide application level security?
76. What tools does Microsoft provide to support Data Governance for Privacy, Confidentiality, and Compliance framework?
77. What is an open source license?

78. What is the open core license model?
79. Briefly compare the open source versus open core models for DBMS software.

PROBLEMS

Due to the nature of this chapter, the problems are more open-ended than other chapters. More detailed problems appear at the end of the other chapters in Part 7.

1. Prepare a short presentation (6 to 12 slides) about the TPC-C benchmark. You should provide details about its history, database design, application details, and recent results.
2. Prepare a short presentation (6 to 12 slides) about the TPC-H benchmark. You should provide details about its history, database design, application details, and recent results.
3. Prepare a short presentation (6 to 12 slides) about the TPC-E benchmark. You should provide details about its history, database design, application details, and recent results.
4. Prepare a short presentation (6 to 12 slides) about the TPC-DS benchmark. You should provide details about its history, database design, application details, and recent results.
5. Compare and contrast the software licenses for MySQL and another open source DBMS product.
6. Develop a list of detailed requirements for nonprocedural retrieval. You should use Table 16-17 as a guideline.
7. Provide importance weights for your list of detailed requirements from problem 6 using the AHP criteria in Table 16-19.
8. Normalize the weights and compute the importance values for your detailed requirements using the importance weights from problem 7.
9. Write named CHECK constraints for the following integrity rules. Modify the CREATE TABLE statement to add the named CHECK constraints.

```
CREATE TABLE Customer
( CustNo            CHAR(8),
  CustFirstName     VARCHAR2(20) CONSTRAINT
                    CustFirstNameRequired NOT NULL,
  CustLastName      VARCHAR2(30) CONSTRAINT
                    CustLastNameRequired NOT NULL,
  CustStreet        VARCHAR2(50),
  CustCity          VARCHAR2(30),
  CustState         CHAR(2),
  CustZip           CHAR(10),
CustBal             DECIMAL(12,2) DEFAULT 0,
CONSTRAINT PKCustomer PRIMARY KEY (CustNo)  )
```

   - Customer balance is greater than or equal to 0.
   - Customer state is one of CO, CA, WA, AZ, UT, NV, ID, or OR.

10. Write named CHECK constraints for the following integrity rules. Modify the CREATE TABLE statement to add the named CHECK constraints.

```
CREATE TABLE Purchase
(  PurchNo          CHAR(8),
   PurchDate        DATE CONSTRAINT PurchDateRequired NOT
                    NULL,
   SuppNo           CHAR(8) CONSTRAINT SuppNo2Required NOT
                    NULL,
```

```
  PurchPayMethod    CHAR(6) DEFAULT 'PO',
  PurchDelDate      DATE,
CONSTRAINT PKPurchase PRIMARY KEY (PurchNo),
CONSTRAINT SuppNoFK2 FOREIGN KEY (SuppNo) REFERENCES
                                  Supplier )
```

- Purchase delivery date is either later than the purchase date or null.
- Purchase payment method is not null when purchase delivery date is not null.
- Purchase payment method is PO, CC, DC, or null.

11. In this problem, you should create a view, several roles, and then grant specific kinds of access of the view to the roles.
    - Create a view of the *Supplier* table in the extended Order Entry Database introduced in the problems section of Chapter 10. The view should include all columns of the *Supplier* table for suppliers of printer products (*Product. ProdName* column containing the word "Printer"). Your view should be named "PrinterSupplierView."
    - Define three roles: PrinterProductEmp, PrinterProductMgr, and StoreMgr.
    - Grant the following privileges of PrinterSupplierView to PrinterProductEmp: retrieval of all columns except supplier discount.
    - Grant the following privileges of PrinterSupplierView to PrinterProductMgr: retrieval and modification of all columns of PrinterSupplierView except supplier discount.
    - Grant the following privileges of PrinterSupplierView to StoreMgr: retrieval for all columns, insert, delete, and modification of supplier discount.
12. Identify important privileges in an enterprise DBMS for data warehouses and database statistics. The privileges are vendor specific so you need to read the documentation of an enterprise DBMS.
13. Identify and briefly describe dictionary tables for database statistics in an enterprise DBMS. The dictionary tables are vendor specific so you need to read the documentation of an enterprise DBMS.
14. Write a short summary (one page) about DBA privileges in an enterprise DBMS. Identify predefined roles and/or user accounts with DBA privileges and the privileges granted to these roles.
15. Investigate the data governance practices of a profit or government organization. You will need to obtain permission to interview individuals with knowledge of data governance practices at your chosen organization. You should investigate the structure of the data governance organization, processes used by the data governance organization, and projects in which the data governance organization has attempted. You should prepare a slide show presentation to communicate your findings. You may need to disguise the organization and individuals interviewed for privacy considerations.

REFERENCES FOR FURTHER STUDY

The book by Jay Louise Weldon (1981) remains the classic book on database administration despite its age. Mullins (2012) provides a more recent comprehensive reference about database administration as well as a website (www.craigsmullins.com) with many articles about database administration. The Database Trends and Applications website (www.dbta.com) contains current details about products and industry developments. The Information Resource Management section of the online list of web resources provides links to information resource management and knowledge management sources. Numerous SQL books provide additional details about

security and integrity features in SQL. Inmon (1986) and Martin (1982) have written detailed descriptions of information systems planning. Castano et al. (1995) is a good reference for additional details about database security. Microsoft (www.microsoft.com/privacy/datagovernance.aspx) and the Data Governance Institute (www.datagovernance.com) provide whitepapers, case studies and other resources about data governance. Gartner (www.gartner.com) and the McKinsey Global Institute (www.mckinsey.com/insights/mgi.aspx) provide analysis of big data challenges and opportunities. For more details about the Analytic Hierarchy Process mentioned in Section 16.3.2, you should consult Saaty (1988) and Zahedi (1986). Some excellent online AHP tutorials can be found by searching the Web using "Analytical Hierarchy Process Tutorials". Su et al. (1987) describe the Logic Scoring of Preferences, an alternative approach to DBMS selection. The Transaction Processing Council (www.tpc.org) provides an invaluable resource about domain-specific benchmarks for DBMSs. Details about the Model Driven Architecture (MDA) can be found in the website of the Object Management Group at www.omg.org/mda.

chapter

17

Transaction Management

Learning Objectives

This chapter describes transaction management features to support concurrent usage of a database and recovery from failures. After this chapter, the student should have acquired the following knowledge and skills:

- Describe transaction examples using the ACID properties
- Explain concepts of concurrency and recovery transparency
- Understand the role of locking to prevent interference problems among multiple users
- Explain the two-phase locking protocol for applying locks
- Understand the role of recovery tools and processes to deal with database failures
- Understand transaction design issues that affect performance
- Describe the relationship of workflow management to transaction management

OVERVIEW

Chapter 16 presented a context for managing databases and an overview of the different processing environments for databases. You learned about the responsibilities of database specialists and the tools and processes used by database specialists. The most prevalent and important database environment is transaction processing that supports the daily operations of an organization. This chapter begins the details of Part 7 by describing DBMS support for transaction processing.

This chapter presents a broad coverage of transaction management. Before you can understand DBMS support for transaction processing, you need a more detailed understanding of transaction concepts. This chapter describes properties of transactions, SQL statements to define transactions, and properties of transaction processing. After learning about transaction concepts, you are ready to study concurrency control and recovery management, two major services to support transaction processing. For concurrency control, this chapter describes the objective, interference problems, and tools of concurrency control. For recovery management, this chapter describes failure types, recovery tools, and recovery processes.

Besides knowing the transaction management services provided by a DBMS, you should understand the issues of transaction design. This chapter describes

important issues of transaction design including hot spots, transaction boundaries, isolation levels, and integrity constraint enforcement. To broaden your background, you should understand how database transactions fit into the larger context of collaborative work. The final section describes workflow management and contrasts it with transaction management in DBMSs.

17.1 BASICS OF DATABASE TRANSACTIONS

Transaction processing involves the operating side of databases. Whereas operations management involves the production of physical goods, transaction management involves the control of information goods or transactions. Transaction management, like management of physical goods, is enormously important to modern organizations. Organizations such as banks with automatic tellers, airlines with online reservation systems, and universities with online registration could not function without reliable and efficient transaction processing. Large organizations now conduct thousands of transactions per minute. With continued growth in electronic commerce, the importance of transaction processing will increase.

Transaction
a unit of work processed together in a reliable manner. DBMSs provide recovery and concurrency control services to process transactions efficiently and reliably.

In common discourse, a transaction is an interaction among two or more parties for the conduct of business such as buying a car from a dealership. Database transactions have a more precise meaning. A database **transaction** involves a collection of operations that must be processed as one unit of work. Transactions should be processed reliably so that there is no loss of data due to multiple users and failures. To help you grasp this more precise meaning, this section presents examples of transactions and defines properties of transactions.

17.1.1 Transaction Examples

A transaction is a user-defined concept. For example, making an airline reservation may involve reservations for the departure and return. To a traveler, the combination of the departure and the return is a transaction, not the departure and the return separately. If a reservation system considers a departure and return separately, a traveler may make a departure without obtaining a desired return flight. Thus, a DBMS must support a transaction as a user defined set of database operations. A transaction can involve any number of reads and writes to a database. To provide the flexibility for user-defined transactions, a DBMS cannot restrict transactions to only a specified number of reads and writes to a database.

An information system may have many different kinds of transactions. Table 17-1 depicts transactions of an order entry system. At any point in time, customers may be conducting business with each of these transactions. For example, some customers may be placing orders while other customers check on the status of their orders. As an additional example of transactions in an information system, Table 17-2 depicts typical transactions in a university payroll system. Some of the transactions are periodic while others are one-time events.

SQL Statements to Define Transactions To define a transaction, you can use some additional SQL statements. Figure 17.1 depicts additional SQL statements to

TABLE 17-1
Typical Transactions in an Order Entry System

Transaction	Description
Add order	Customer places a new order.
Update order	Customer changes details of an existing order.
Check status	Customer checks the status of an order.
Payment	Payment received from a customer.
Shipment	Goods sent to a customer.

Transaction	Description
Hire employee	Employee begins service with the university.
Pay employee	Periodic payment made to employee for service.
Submit time record	Hourly employees submit a record of hours worked.
Reappointment	Assign employee to a new position.
Evaluation	Periodic performance evaluation.
Termination	Employee leaves employment at the university.

TABLE 17-2
Typical Transactions in a University Payroll System

```
START TRANSACTION
    Get account number, pin, type, and amount
    SELECT account number, type, and balance
    If balance is sufficient Then
       UPDATE account by posting debit
       UPDATE account by posting credit
       INSERT history row
       Display final message and issue cash
    Else
       Write error message
    End If
    On Error: ROLLBACK
COMMIT
```

FIGURE 17.1
Pseudo Code for an ATM Transaction

define the prototypical automatic teller machine (ATM) transaction. The START TRANSACTION[1] and COMMIT statements define the statements in a transaction. Any other SQL statements between them are part of the transaction. Typically, a transaction involves a number of SELECT, INSERT, UPDATE, and DELETE statements. In Figure 17.1, an actual transaction would have valid SQL statements for the lines beginning with SELECT, UPDATE, and INSERT. Valid programming language statements should be substituted for lines with pseudo code such as "Display greeting."

Besides the START TRANSACTION and COMMIT statements, the ROLLBACK statement may be used. ROLLBACK behaves like an undo command in a word processor to remove effects of user actions. Unlike an undo command, ROLLBACK applies to a sequence of actions not just a single action. Thus, a ROLLBACK statement removes all actions of a transaction to restore a database to the state before execution of the transaction.

You can use a ROLLBACK statement in several contexts. In one situation, you use a ROLLBACK statement to allow a user to cancel a transaction. In a second situation, you use a ROLLBACK statement to respond to errors. In this situation, the ROLLBACK statement appears as part of exception-handling statements such as the "On Error" line in Figure 17.1. Exception-handling statements are part of programming languages such as Java and Visual Basic. Exception handling supports separate processing of unanticipated errors such as communication errors from the normal logic of a transaction.

As you will learn later in this chapter, transactions should have short duration. To shorten duration of the ATM transaction, a transaction designer should place user interaction outside of the transaction. In the ATM transaction, the START TRANSACTION statement should be moved after the first three lines to remove user interaction. Long-running transactions can cause excessive waiting among concurrent users of a database. However, the UPDATE and INSERT statements must remain in the same

[1] SQL:2016 specifies the START TRANSACTION and COMMIT statements. Some DBMSs use the keyword BEGIN instead of START. Other DBMSs such as Oracle do not use a statement to start a transaction. Rather, a new transaction begins with the next SQL statement following a COMMIT statement.

transaction because these statements are part of the same unit of work. Section 17.4.1 discusses issues related to removing user interaction time from a transaction. Sometimes, user interaction time should remain part of a transaction.

Other Transaction Examples Figures 17.2 and 17.3 depict transactions for an airline reservation and online shopping purchase. In both examples, the transaction consists of more than one database action (read or write). The airline reservation updates two flight rows along with a reservation row. In a more complex airline reservation involving multiple flights per part (departure or return), more flight rows must be updated. In the online shopping purchase, product rows are updated and order detail rows are inserted.

17.1.2 Transaction Properties

DBMSs ensure that transactions obey certain properties. The most important and widely known properties are the ACID properties (atomic, consistent, isolated, and durable) as presented in the following list.

- *Atomic* means that a transaction cannot be subdivided. Either all the work in the transaction is completed or nothing is done. For example, the ATM transaction will not debit an account without also crediting a corresponding account. The atomic property implies that partial changes made by a transaction must be undone if a transaction aborts.
- *Consistent* means that if applicable constraints are true before the transaction starts, the constraints will be true after the transaction terminates. For example, if a user's account is balanced before a transaction, then the account is balanced after the transaction. Otherwise, the transaction is rejected and no changes take effect.

FIGURE 17.2
Pseudo Code for an Airline Reservation Transaction

```
START TRANSACTION
    Get reservation preferences
    SELECT departure and return flight rows
    If reservation is acceptable Then
        Get payment method and details
        UPDATE seats remaining of departure flight row
        UPDATE seats remaining of return flight row
        INSERT reservation row
        Send receipt to customer
        Send payment data to payment processor
    End If
    On Error: ROLLBACK
COMMIT
```

FIGURE 17.3
Pseudo Code for an Online Shopping Transaction

```
START TRANSACTION
    Place product selections in shopping cart
    If checking out Then
        Get account and payment details
        SELECT account row
        For each product row
            UPDATE QOH of product row
            INSERT order detail row
            Send message to shipping department
        End For
    Send confirmation message to customer
    Send payment data to payment processor
    End If
    On Error: ROLLBACK
COMMIT
```

- *Isolated* means that transactions do not interfere with each other except in allowable ways. A transaction should never overwrite changes made by another transaction. In addition, a transaction may be restricted from interfering in other ways such as not viewing the temporary changes made by other transactions. For example, your significant other will not know that you are withdrawing money until your ATM transaction completes.
- *Durable* means that any changes resulting from a transaction are permanent. No failure will erase any changes after a transaction terminates. For example, if a bank's computer experiences a failure five minutes after a transaction completes, the results of the transaction are still recorded on the bank's database.

To ensure that transactions meet the ACID properties, DBMSs provide certain services that are transparent to database developers (programmers and analysts). In common usage, transparency means that you can see through an object, rendering its inner details invisible. For DBMSs, transparency means that inner details of transaction services are invisible. Transparency is very important because services that ensure ACID transactions are difficult to implement. By providing these services, DBMSs improve productivity of database programmers and analysts.

DBMSs provide two services, recovery transparency and concurrency transparency, to ensure that transactions obey the ACID properties. Recovery involves actions to deal with failures such as communication errors and software crashes. Concurrency involves control of interference among multiple, simultaneous users of a database. The following discussion provides details about transparency for recovery and concurrency.

- *Recovery transparency* means that the DBMS automatically restores a database to a consistent state after a failure. For example, if a communication failure occurs during an ATM transaction, the DBMS removes effects of the transaction from the database. On the other hand, if the DBMS crashes three seconds after an ATM transaction completes, changes made by the transaction remain permanent.
- *Concurrency transparency* means that users perceive a database as a single-user system even though there may be many simultaneous users. For example, even though many users may try to reserve a popular flight using a reservation transaction, the DBMS ensures that users do not overwrite each other's work.

Even though the inner details of concurrency and recovery are invisible to users, these services are not free. Recovery and concurrency control involve overhead that may require additional resources and careful monitoring to reach an acceptable level of performance. The DBA must be aware of resource implications of these services and understand tools to monitor performance. More computing resources such as memory, disk space, and parallel processing may be necessary to improve performance. Performance monitoring is required to ensure adequate performance. The DBA should monitor key indicators of performance and change parameter settings to alleviate performance problems.

In addition to resources and monitoring, the selection of a DBMS can be crucial to achieve acceptable transaction processing performance. The purchase price of a DBMS often depends on the level of transaction services provided. Most DBMSs have different editions to support workgroups to entire enterprises. DBMS editions can vary by the number of concurrent transactions supported, parallel processing services provided, and recovery services available. DBMSs that support large numbers of concurrent users with very high availability can be costly.

Transaction design is another reason for understanding details of concurrency control and recovery. Even with a high performance DBMS and adequate resources, poor transaction design can lead to performance problems. To achieve a satisfactory transaction design, you should have background about concurrency control and recovery as well as understand transaction design principles, as discussed in the following sections.

17.2 CONCURRENCY CONTROL

Most organizations cannot function without multiuser databases. For example, airline, retail, banking, and product support databases can have thousands of users simultaneously trying to conduct business. Multiple users can access these databases concurrently, that is, at the same time. If access was restricted to only one user at a time, little work would be accomplished and most users would take their business elsewhere. However, concurrent users cannot be permitted to interfere with each other. This section defines the objective, problems, and tools of concurrency control.

17.2.1 Objective of Concurrency Control

Transaction Throughput
the number of transactions processed per time interval. It is an important measure of transaction processing performance.

The objective of concurrency control is to maximize transaction throughput while preventing interference among multiple users. **Transaction throughput**, the number of transactions processed per time unit, measures the amount of work performed by a DBMS. Typically, transaction throughput is reported by transactions per minute. In a high-volume environment such as electronic commerce, DBMSs may need to process hundreds of thousands of transactions per minute. In 2013, the Transaction Processing Council (www.tpc.org) reported top results for the TPC-C benchmark (an order entry benchmark) ranging from 1.02 million to 8.55 million transactions per minute.

From a user's perspective, transaction throughput relates to response time. Higher transaction throughput means faster response times. Users are typically unwilling to wait more than a few seconds for completion of a transaction.

If there is no interference, the result of executing concurrent transactions is the same as executing the same transactions in some sequential order. Sequential execution means that one transaction completes before another one executes, thus ensuring no interference. Executing transactions sequentially would result in low throughput and high waiting times. Thus, DBMSs allow transactions to execute simultaneously while ensuring the results are the same as though executed sequentially.

Transactions executing concurrently cannot interfere unless they are manipulating common data. Essentially, the concurrency control component must maintain the same transaction order for each common database item. For example, if transaction *i* writes database item A before transaction *j* writes database item A, then transaction *j* cannot write item B before transaction *i* writes item B. If transaction *j* writes item B before transaction *i*, a concurrency control violation has occurred. Section 17.2.2 clarifies this informal definition of correct concurrent execution with presentation of specific concurrency control problems.

Most concurrent transactions manipulate only small amounts of common data. For example, in an airline reservation, two users can concurrently enter new reservation rows because the reservation rows are unique for each customer. However, interference can occur on the seats-remaining column of a flight table. For popular flights, many users may want to decrement the value of the seats-remaining column. It is critical that a DBMS control concurrent updating of this column in popular flight rows.

Hot Spot
common data that multiple users try to change. Without adequate concurrency control, users may interfere with each other on hot spots.

A **hot spot** is common data that multiple users try to change simultaneously. Essentially, a hot spot represents a scarce resource that users must queue to access. Typical hot spots are the seats-remaining for popular flights, the quantity-on-hand of popular inventory items, and the seats-taken in popular course offerings. In an ideal world, DBMSs would only track hot spots. Unfortunately, hot spots can be difficult to predict so DBMSs track access to all parts of a database.

Interference on hot spots can lead to lost data and poor decision-making. The following sections describe interference problems and tools to prevent them.

17.2.2 Interference Problems

Interference among concurrent users of a database can cause three problems: (1) lost update, (2) uncommitted dependency, and (3) inconsistent retrieval. This section defines each problem and presents examples of their occurrence.

Lost Update Lost update is the most serious interference problem because changes to a database are inadvertently lost. In a lost update, one user's update overwrites another user's update, as depicted by the timeline of Figure 17.4. The timeline shows two concurrent transactions trying to update the seats remaining (*SR*) column of the same flight row. Assume that the value of *SR* is 10 before the transactions begin. After time T_2, both transactions have stored the value of 10 for *SR* in local buffers as a result of the read operations. After time T_4, both transactions have made changes to their local copy of *SR*. However, each transaction changes the value to 9, unaware of the activity of the other transaction. After time T_6, the value of *SR* on the database is 9. However, the value after finishing both transactions should be 8, not 9! One of the changes has been lost.

Lost Update
a concurrency control problem in which one user's update overwrites another user's update.

Some students become confused about the lost update problem because of the actions performed on local copies of the data. The calculations at times T_3 and T_4 occur in memory buffers specific to each transaction. Even though transaction A has changed the value of *SR*, transaction B performs the calculation with its own local copy of *SR* having a value of 10. The write operation performed by transaction A is not known to transaction B unless transaction B reads the value again.

A lost update involves two or more transactions trying to change (write to) the same part of a database. As you will see in the next two problems, two transactions can also conflict if only one is changing the database.

Uncommitted Dependency An uncommitted dependency occurs when one transaction reads data written by another transaction before the other transaction commits. An uncommitted dependency is also known as a dirty read because it is caused by one transaction reading dirty (uncommitted) data. In Figure 17.5, transaction A reads the *SR* column, changes its local copy of the *SR* column, and writes the new value back to the database. Transaction B then reads the changed value. Before transaction A commits, however, an error is detected and transaction A issues a rollback. The rollback could occur because the user canceled the transaction or the transaction failed. The value used by transaction B is now a phantom value. The real *SR* value is now 10 because A's change was not permanent. Transaction B may use its value (9) to make an erroneous decision. For example, if *SR*'s value was 1 before transaction A began, transaction B might be denied a reservation.

Uncommitted Dependency
a concurrency control problem in which one transaction reads data written by another transaction before the other transaction commits. If the second transaction aborts, the first transaction may rely on data that will no longer exist.

Because data are not permanent until a transaction commits, a conflict can occur even though only one transaction writes to the database. An uncommitted dependency involves one transaction writing and another transaction reading the same part of the

Transaction A	Time	Transaction B
Read SR (10)	T_1	
	T_2	Read SR (10)
If SR > 0 then SR = SR − 1	T_3	
	T_4	If SR > 0 then SR = SR − 1
Write SR (9)	T_5	
	T_6	Write SR (9)

FIGURE 17.4
Example Lost Update Problem

Transaction A	Time	Transaction B
Read SR (10)	T_1	
SR = SR − 1	T_2	
Write SR (9)	T_3	
	T_4	Read SR (9)
Rollback	T_5	

FIGURE 17.5
Example Uncommitted Dependency Problem

database. However, an uncommitted dependency cannot cause a problem unless a rollback occurs. The third problem also involves conflicts when only one transaction writes to a database.

Problems Involving Inconsistent Retrievals The last problem involves situations in which interference causes inconsistency among multiple retrievals of a subset of data. All inconsistent retrieval problems involve one transaction reading and the second transaction changing the same part of the database. The **incorrect summary** problem is the most significant problem involving inconsistent retrievals. An incorrect summary[2] occurs when a transaction calculating a summary function, reads some values before another transaction changes the values but reads other values after another transaction changes the values. In Figure 17.6, transaction B reads SR_1 after transaction A changes the value but reads SR_2 before transaction A changes the value. For consistency, transaction B should read all values either before or after other transactions change the values.

Incorrect Summary
a concurrency control problem in which a transaction reads several values, but another transaction updates some of the values while the first transaction is still executing.

A second problem involving inconsistent retrievals, known as the phantom read problem, occurs when a transaction executes a query with row conditions. Then, another transaction inserts or modifies data that the query would retrieve. Finally, the original transaction executes the same query again. The second query execution retrieves different rows than the first execution. The new and changed rows seem phantom because they did not exist in the result of the first query execution.

A third problem involving inconsistent retrievals, known as the nonrepeatable read problem, occurs when a transaction reads the same value more than one time. In between reading the data item, another transaction modifies the data item. The second retrieval contains a different value than the first retrieval because of the change made by the other transaction.

The nonrepeatable read and phantom read problems are slightly different. A nonrepeatable read problem would occur if another user changed the value of a column of a query row so that the query returns a different value in the next execution. A phantom read problem would occur if a new inserted row matches a condition so that the query retrieves an additional row in the next execution. The key difference is the row condition requirement for the phantom read problem.

17.2.3 Concurrency Control Tools

This section describes two tools, locks and the two-phase locking protocol, used by most DBMSs to prevent the three interference problems discussed in the previous section. In addition to the two tools, the deadlock problem is presented because a

FIGURE 17.6
Example Incorrect Summary Problem

Transaction A	Time	Transaction B
Read SR_1 (10)	T_1	
$SR_1 = SR_1 - 1$	T_2	
Write SR_1 (9)	T_3	
	T_4	Read SR_1 (9)
	T_5	Sum = Sum + SR_1
	T_6	Read SR_2 (5)
	T_7	Sum = Sum + SR_2
Read SR_2 (5)	T_8	
$SR_2 = SR_2 - 1$	T_9	
Write SR_2 (4)	T_{10}	

[2] An incorrect summary is also known as an inconsistent analysis.

deadlock can be a negative byproduct resulting from lock usage. This section closes by briefly discussing optimistic concurrency control approaches that do not use locks.

Locks Locks provide a way to prevent other users from accessing a database item in use. A database item can be a row, block, a subset of rows, or even an entire table. Before accessing a database item, a lock must be obtained. Other users must wait if trying to obtain a conflicting lock on the same item. Table 17-3 shows conflicts for two types of locks. A shared (S) lock must be obtained before reading a database item, whereas an exclusive (X) lock must be obtained before writing. As shown in Table 17-3, any number of users can hold a shared lock on the same item. However, only one user can hold an exclusive lock.

Lock
a fundamental tool of concurrency control. A lock on a database item prevents other transactions from performing conflicting actions on the same item.

The concurrency control manager is the part of the DBMS responsible for managing locks. The concurrency control manager maintains a hidden[3] table to record locks held by various transactions. A lock record contains a transaction identifier, a row identifier, a lock type, and a count, as explained in Table 17-4. In the simplest scheme, the lock type is either shared or exclusive, as discussed previously. Most DBMSs have other types of locks to improve efficiency and allow for more concurrent access. The concurrency control manager performs two operations on lock records. The lock operator adds a row to the lock table whereas the unlock operator or release operator deletes a row from the lock table.

Locking Granularity Locking granularity is a complication about locks. Granularity refers to the size of the database item locked. Most DBMSs can hold locks for different granularities, as depicted in Figure 17.7. The entire database is the coarsest lock that can be held. If an exclusive lock is held on the entire database, no other users can access

Locking Granularity
the size of the database item locked. Locking granularity is a trade-off between waiting time (amount of concurrency permitted) and overhead (number of locks held).

TABLE 17-3
Basic Lock Compatibility Matrix

	User 2 Requests	
User 1 Holds	**S Lock**	**X Lock**
S Lock	Lock granted	User 2 waits
X Lock	User 2 waits	User 2 waits

TABLE 17-4
Fields in a Lock Record

Field Name	Description
Transaction identifier	Unique identifier for a transaction
Row identifier	Identifies the row to be locked
Lock type	Indicates the intended usage of the locked row
Count	Number of other users holding this kind of lock

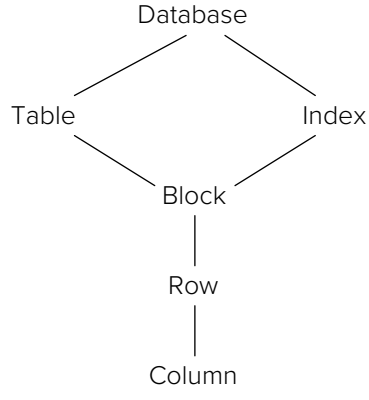


FIGURE 17.7
Typical Levels of Locking Granularity

[3] Most DBMSs provide tools for DBAs to view the lock table.

the database until the lock is released. On the other extreme, a column value is the finest lock that can be held. Locks also can be held on parts of the database not generally seen by users. For example, locks can be held on indexes and blocks (physical records).

Locking granularity is a trade-off between overhead and waiting. Holding locks at a fine level decreases waiting among users but increases system overhead because more locks must be obtained. Holding locks at a coarser level reduces the number of locks but increases the amount of waiting. In some DBMSs, the concurrency control manager tries to detect the pattern of usage and promotes locks if needed. For example, a concurrency control manager initially can grant row locks to a transaction in anticipation that only a few rows will be locked. If the transaction continues to request locks, the concurrency control component can promote the row locks to a lock on a subset of rows or the entire table.

Intent Lock
a lock on a large database item (such as a table) indicating intention to lock smaller items contained in the larger item. Intent locks alleviate blocking when locking coarse items and allow efficient detection of conflicts among locks on items of varying granularity.

To alleviate blocking caused by locking coarse items as shared or exclusive, **intent locks** are often used. Intent locks support more concurrency on coarse items than shared or exclusive locks. Intent locks also allow efficient detection of conflicts among locks on items of varying granularity. Intent locks improve performance because locking conflicts can be detected by examining items on the same level, not items on finer levels.

To support these goals, most DBMSs use three types of intent locks: (1) intent shared (IS) when intending to read lower-level items, (2) intent exclusive (IX) when intending to write lower-level items, and (3) shared with intent exclusive (SIX) when intending to read all lower level items and write some lower-level items. For example, a transaction should request an intent shared lock on a table for which it intends to read some blocks of the table. A transaction should request a shared with intent exclusive lock on a block when it will read all rows of a block and update some of the rows.

Intent locks require an expanded lock compatibility matrix as depicted in Table 17-5. To interpret this table, you should remember that conflicts apply to items on the same level of granularity but also prevent conflicts among lower level items. For example, SIX conflicts with IX because the shared locks on all lower level items (SIX) conflict with exclusive locks on some lower level items (IX). SIX conflicts with SIX because shared locks on all lower level items conflicts with exclusive locks on some lower level items.

Deadlock
a problem of mutual waiting that can occur when using locks. If a deadlock is not resolved, the involved transactions will wait forever. A DBMS can control deadlocks through detection or a timeout policy.

Deadlocks Using locks to prevent interference problems can lead to deadlocks. A deadlock is a problem of mutual waiting. One transaction has a resource that a second transaction needs, and the second transaction holds a resource that the first transaction needs. Figure 17.8 depicts a deadlock among two transactions trying to purchase two popular electronic goods (say a 2-1 laptop computer and a digital stylus). Transaction A obtains an exclusive lock on the product row containing electronic good 1. The second transaction obtains an exclusive lock on the product row containing electronic good 2. Transaction A then tries to obtain an exclusive lock on the row containing electronic good 2 but is blocked because transaction B holds an exclusive lock. Likewise, transaction B must wait to obtain an exclusive lock on the row containing electronic good 1. Deadlocks can involve more than two transactions, but the pattern is more complex.

TABLE 17-5
Expanded Lock Compatibility Matrix

	User 2 Requests				
User 1 Holds	**IS lock**	**IX lock**	**SIX lock**	**S lock**	**X lock**
IS	Grant	Grant	Grant	Grant	Wait
IX	Grant	Grant	Wait	Wait	Wait
SIX	Grant	Wait	Wait	Wait	Wait
S	Grant	Wait	Wait	Grant	Wait
X	Wait	Wait	Wait	Wait	Wait

Transaction A	Time	Transaction B
XLock Pr_1	T_1	
	T_2	XLock Pr_2
XLock Pr_2 (wait)	T_3	
	T_4	XLock Pr_1 (wait)

FIGURE 17.8
Example Deadlock Problem

To control deadlocks, most enterprise DBMSs perform deadlock detection. Deadlocks can be detected by looking for cyclic patterns of mutual waiting. In practice, most deadlocks involve two or three transactions. Because deadlock detection can involve significant computation time, deadlock detection is only performed at periodic intervals or triggered by waiting transactions. For example, deadlock detection for Figure 17.8 could be performed when transaction B is forced to wait. When a deadlock is detected, the transaction with the latest start time (transaction B in Figure 17.8) is usually forced to restart.

Some DBMSs use a simpler time-out policy to control deadlocks. In a time-out policy, the concurrency control manager aborts (with a ROLLBACK statement) any transaction waiting for more than a specified time. Note that a time-out policy may abort transactions that are not deadlocked. The time-out interval should be set large enough so that few non deadlocked transactions will wait that long.

Some DBMSs use update locks to reduce the number of deadlocks. An update lock addresses the typical situation of initially acquiring a shared lock before promoting the lock to exclusive before updating. This pattern can lead to deadlocks as transactions are initially granted shared locks but then become unable to promote the locks to exclusive because other transactions hold conflicting shared locks. Conflicts with update locks are subtle. An update lock can be acquired on a row when other users hold a read lock on the row. However, no other user can get another lock (shared, exclusive, or update) on the row after a user obtains an update lock. Because an update lock prevents subsequent read locks, it is easier to convert the update lock to an exclusive lock. If a transaction modifies an item, the update lock is converted to an exclusive lock. Otherwise, it is converted to a shared lock.

Two-Phase Locking Protocol To ensure that lost update problems do not occur, the concurrency control manager requires that all transactions follow the Two-Phase Locking (2PL) protocol. In computing, a *protocol* defines a rule about group behavior. A protocol binds all members of a group to behave in a specified manner. For human communication, Robert's Rules of Order require that all meeting participants follow certain rules. For data communication, protocols ensure that messages have a common format that both sender and receiver can recognize. For concurrency control, all transactions must follow the 2PL protocol to ensure that concurrency control problems do not occur. Two-phase locking has three conditions as listed in the following note.

Definition of 2PL

(1) Before reading or writing a data item, the transaction must acquire the applicable lock to the data item.
(2) Wait if a conflicting lock is held on a data item.
(3) After releasing a lock, the transaction does not acquire any new locks.

The first two conditions follow from the usage of locks as previously explained. The third condition is subtle. If new locks are acquired after releasing locks, two transactions can obtain and release locks in a pattern in which a concurrency control problem occurs.

In practice, the concurrency control component of a DBMS simplifies the third condition to hold exclusive locks until the end of the transaction. At commit time, the concurrency control component releases all locks of a transaction. Figure 17.9 graphically depicts the 2PL protocol with the simplified third condition. At the beginning of the transaction (BOT), a transaction has no locks. A growing phase ensues in which the transaction acquires locks but never releases any locks. At the end of the transaction (EOT), the shrinking phase occurs in which all locks are released together. Simplifying the definition of 2PL makes the protocol easier to enforce and obviates the difficult problem of predicting when a transaction may release locks.

For locking granularity with intent locks, the 2PL protocol is slightly extended. Locking begins with the root (coarsest item) and proceeds to finer level items. To obtain an S or IS lock on an item, a transaction must hold an IS or IX lock on the parent item. To obtain an X, IX, or SIX lock on an item, a transaction must obtain an IX or SIX lock on the parent item. Locks are released in the opposite order from finest to coarsest. This revised protocol is equivalent to directly locking items at the lowest level but much more efficient.

To provide flexibility between concurrency control problems permitted and potential waiting, most DBMSs relax the 2PL protocol to permit some locks to be released before the end of a transaction. Section 17.4.2 presents the concept of isolation levels to determine the level of interference tolerated.

Optimistic Approaches The use of locks and 2PL is a pessimistic approach to concurrency control. Locking assumes that every transaction conflicts. If contention only involves relatively few hot spots, then locking may require excessive overhead.

Optimistic concurrency control approaches assume that conflicts are rare. If conflicts are rare, it is more efficient to check for conflicts rather than manage a large number of locks. In optimistic approaches, transactions are permitted to access a database without acquiring locks. Instead, the concurrency control manager checks whether a conflict has occurred. The check can be performed either just before a transaction commits or after each read and write. By reviewing the relative time of reads and writes, the concurrency control manager can determine whether a conflict has occurred. If a conflict occurs, the concurrency control manager issues a rollback and restarts the offending transaction.

Despite the appeal of optimistic approaches, most organizations used 2PL even though some enterprise DBMSs supported optimistic approaches. The performance of optimistic approaches depends on the frequency of conflicts. If conflicts increase, the performance of optimistic approaches decreases. Even if conflicts are rare, optimistic approaches can have more variability because the penalty for conflicts is larger in optimistic approaches. Pessimistic approaches resolve conflicts by waiting. Optimistic approaches resolve conflicts by rolling back and restarting. Restarting a transaction may delay a transaction more than waiting for a resource to be released. Thus, optimistic approaches may have more variability in transaction service times than pessimistic approaches with locking.

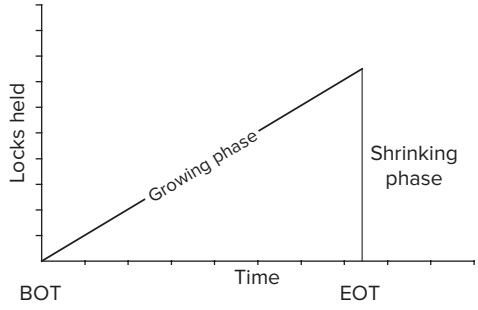


FIGURE 17.9
Growing and Shrinking Phases of 2PL

Recent advances in **in-memory transaction processing** have begun to change the marketplace and organizational practices for mission critical applications. Most enterprise DBMS vendors now offer high performance, in-memory transaction processing using optimistic concurrency control at least in the highest product editions. The optimistic concurrency control approach has been streamlined to reduce the probability of conflicts at commit time. Elimination or substantially reduced usage of locks (just at transaction commit time) minimizes concurrency overhead. Distributed processing of transactions on multiple CPU cores and main memory resident tables dramatically improves transaction throughput.

In-Memory Transaction Processing
a high performance technology utilizing distributed processing with large amounts of main memory and streamlined optimistic concurrency control for mission critical applications. Both enterprise relational DBMSs and emerging NoSQL DBMSs support in-memory transaction processing.

As a result of in-memory transaction processing technology, major organizations with mission critical applications are increasingly considering optimistic concurrency control. Enterprise DBMS vendors claim high potential (such as 10 times) performance improvement. However these benefits have costs of larger levels of main memory, effort to convert tables to utilize in-memory processing, redesign of transactions to utilize in-memory processing, and sometimes increased product licensing costs. Organizations should investigate trade-offs on individual tables to determine viability of in-memory transaction processing.

17.3 RECOVERY MANAGEMENT

Recovery management is a service to restore a database to a consistent state after a failure. This section describes the kinds of failures to prevent, the tools of recovery management, and the recovery processes that use recovery tools.

17.3.1 Data Storage Devices and Failure Types

From the perspective of database failures, volatility is an important characteristic of data storage devices. Main memory is volatile because it loses its state if power is lost. In contrast, a disk drive (magnetic or solid state) is nonvolatile because it retains its state if power is lost. This distinction is important because a DBMS cannot depend on volatile memory to recover data after failures. Even nonvolatile devices are not completely reliable. For example, certain failures make the contents of a disk drive unreadable. To achieve high reliability, DBMSs may replicate data on several kinds of nonvolatile storage media such as a hard disk, magnetic tape, and optical disk. Using a combination of nonvolatile devices improves reliability because different kinds of devices usually have independent failure rates.

Some failures affect main memory only, while others affect both volatile and nonvolatile memory. Table 17-6 shows four kinds of failures along with their effect and frequency. The first two kinds of failures affect the memory of one executing transaction. Transaction code should check for error conditions such as an invalid account number or cancellation of the transaction by the user. A program-detected failure usually leads to aborting the transaction with a specified message to the user. The SQL ROLLBACK statement can abort a transaction if an abnormal condition occurs. Recall that the ROLLBACK statement removes all changes made by the transaction. Program-detected failures are usually the most common and least harmful.

Abnormal termination has a similar effect as a program-detected failure but a different cause. The transaction aborts, but the error message is often unintelligible to

TABLE 17-6
Failure Types, Effects, and Frequency

Type	Effect	Frequency
Program-detected	Local (1 transaction)	Most frequent
Abnormal termination	Local (1 transaction)	Moderate frequency
System failure	Global (all active transactions)	Not frequent
Device failure	Global (all active and past transactions)	Least frequent

the user. Abnormal termination can be caused by events such as transaction time-out, communication line failure, or programming error (for example, dividing by zero). The ON ERROR statement in Figure 17.1 detects abnormal termination. A ROLLBACK statement removes any effects of the terminated transaction on the database.

The last two kinds of failures have more serious consequences but are usually far less common. A system failure is an abnormal termination of the operating system. An operating system failure affects all executing transactions. A device failure such as a disk crash affects all executing transactions and all committed transactions whose work is recorded on the disk. A device failure can take hours to recover while a system crash can take minutes.

17.3.2 Recovery Tools

The recovery manager uses redundancy and control of the timing of database writes to restore a database after a failure. Three tools discussed in this section, transaction log, checkpoint, and database backup, are forms of redundancy. The last tool (force writing) allows the recovery manager to control when database writes are recorded. This section explains the nature of these tools, while the next section explains how these tools are used in recovery processes.

Transaction Log
a table that contains a history of database changes. The recovery manager uses the log table to recover from failures.

Transaction Log A transaction log provides a history of database changes. Every change to a database is also recorded in the log. The log is a hidden table not available to normal users. A typical log (Table 17-7) contains a unique log sequence number (LSN), a transaction identifier, a database action, a time, a row identifier, a column name, and values (old and new). For insert operations, the column name * denotes all columns with the new value containing an entire row of values. The old and new values are sometimes called the before and after images, respectively. For insert actions, the log only contains the new values. Similarly, for delete actions, the log only contains the old values. Besides insert, update, and delete actions, log records are created for the beginning and ending of a transaction.

The recovery manager can perform two operations on the log. In an undo operation, the database reverts to a previous state by substituting the old value for whatever value is stored in the database. In a redo operation, the recovery component reestablishes a new state by substituting the new value for whatever value is stored in the database. To undo (redo) a transaction, the undo (redo) operation is applied to all log records of a specified transaction except for the start and commit records.

Checkpoint
the act of writing a checkpoint record to the log and writing log and some database buffers to disk. All transaction activity ceases while a checkpoint occurs. The checkpoint interval should be chosen to balance restart time with checkpoint overhead.

A log can add considerable storage overhead. In an environment of large transaction volumes, 100 gigabytes of log records can be generated each day. Because of this large size, many organizations have both an online log stored on disk and an archive log stored on tape or optical disk. The online log is usually divided into two parts (current and next) to manage online log space. Given the role of the log in the recovery process, the integrity of the log is crucial. Enterprise DBMSs can maintain redundant logs to provide nonstop processing in case of a log failure.

Checkpoint The purpose of a checkpoint is to reduce the time to recover from failures. At periodic times, a checkpoint record is written to the log to record all active transactions. In addition, all log buffers as well as some database buffers are written

TABLE 17-7
Example Transaction Log for an ATM Transaction

LSN	TransNo	Action	Time	Table	Row	Column	Old	New
1	101001	START	10:29					
2	101001	UPDATE	10:30	Acct	10001	AcctBal	100	200
3	101001	UPDATE	10:30	Acct	15147	AcctBal	500	400
4	101001	INSERT	10:32	Hist	25045	*		<1002, 500, ...>
5	101001	COMMIT	10:33					

to disk. At restart time, the recovery manager relies on the checkpoint log record and knowledge of log and database page writes to reduce the amount of restart work.

The checkpoint interval is defined as the period between checkpoints. The interval can be expressed as a time (such as five minutes) or as a size parameter such as the number of committed transactions, the number of log pages, or the number of database pages. The checkpoint interval is a design parameter. A small interval reduces restart work but causes more overhead to record checkpoints. A large interval reduces checkpoint overhead but increases restart work. A typical checkpoint interval might be 10 minutes for large transaction volumes.

Recording a checkpoint may involve considerable disruption in transaction processing as all transaction activity ceases while a checkpoint occurs. No new transactions can begin and existing transactions cannot initiate new operations during a checkpoint. The length of the disruption depends on the type of checkpoint used. In a cache-consistent checkpoint, buffer pages (log pages and dirty database pages) remaining in memory are written to disk and then the checkpoint record is written to the log. A page is dirty if it has been changed by a transaction.

To reduce the disruption caused by cache-consistent checkpoints, some DBMSs support either fuzzy checkpoints or incremental checkpoints. In a fuzzy checkpoint, the recovery manager only writes dirty database pages older than the previous checkpoint. Since most dirty database pages should have already been written to disk before the checkpoint, a fuzzy checkpoint should write fewer database pages than a cache consistent checkpoint. At restart time, the recovery manager uses the two most recent fuzzy checkpoint records in the log. Thus, fuzzy checkpoints involve less overhead than cache-consistent checkpoints but may require more restart work.

In an incremental checkpoint, no database pages are written to disk. Instead, dirty database pages are periodically written to disk in ascending age order. At checkpoint time, the log position of the oldest dirty data page is recorded to provide a starting point for recovery. The amount of restart work can be controlled by the frequency of writing dirty data pages.

Figure 17.10 provides a convenient summary of the three types of checkpoints. Enterprise DBMSs may provide a choice among more than one type of checkpoint. The trend is to use more resource efficient checkpoints (fuzzy and incremental) at the cost of somewhat longer restart times.

Force Writing The ability to control the timing of database page transfers to nonvolatile storage is known as force writing. Without the ability to control the timing of write operations to nonvolatile storage, reliable recovery is not possible. Force writing

Force Writing
the ability to control the timing of database page transfers to nonvolatile storage. This ability is fundamental to recovery management.

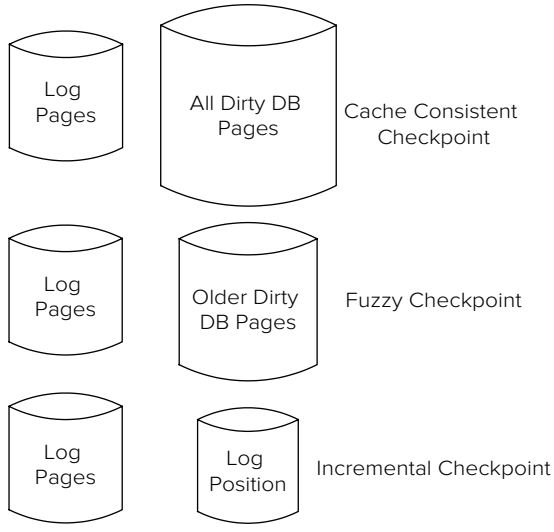


FIGURE 17.10
Summary of Checkpoint Types

means that the DBMS, not the operating system, controls when data are written to nonvolatile storage. Normally, when a program executes a write procedure, the operating system puts the data in a buffer. For efficiency, the data are not written to disk until the buffer is full. Typically, there is some small delay between arrival of data in a buffer and transferring the buffer to disk. With force writing, the operating system allows the DBMS to control when the buffer is written to disk.

The recovery manager uses force writing at checkpoint time and the end of a transaction. At checkpoint time, in addition to inserting a checkpoint record, all log and possibly some database buffers are force written to disk. This force writing can add considerable overhead to the checkpoint process. At the end of a transaction, the recovery manager force writes any log records of a transaction remaining in memory.

Database Backup A backup is a copy of all or part of a disk. The backup is used when the disk containing the database or log is damaged. For high reliability and faster restart processing, backups can be made on independent disk drives as well as sequential media such as magnetic tape. Periodically, a backup should be made for both the database and the log. To save time, most backup schedules include less frequent massive backups to copy the entire contents of a disk and more frequent incremental backups to copy only the changed part.

17.3.3 Recovery Processes

The recovery process depends on the kind of failure. Recovery from a device failure is simple but can be time-consuming, as listed below

- The database is restored from the most recent backup.
- Then, the recovery manager applies the redo operator to all committed transactions after the backup. Because the backup may be several hours to days old, the log must be consulted to restore transactions committed after the backup.
- The recovery process finishes by restarting incomplete transactions.

For local failures and system failures, the recovery process depends on when database changes are recorded on disk. Database changes can be recorded before the commit (immediate update) or after the commit (deferred update). The amount of work and the use of log operations (undo and redo) depend on the timing of database updates. The remainder of this section describes recovery processes for local and system failures under each scenario.

Immediate Update Approach
database updates are written to disk when they occur but after the corresponding log updates. To restore a database, both undo and redo operations may be needed.

Immediate Update Approach In the immediate update approach, database updates are written to disk when they occur. Database writes may also occur at checkpoint time depending on the checkpoint type. Database writes must occur after writes of the corresponding log records. This usage of the log is known as the write ahead log protocol. If log records were written after corresponding database records, recovery would not be possible if a failure occurred between the time of writing the database records and the log records. To support the write ahead log protocol, the recovery manager maintains a table of log sequence numbers for each database page in a buffer. A database page cannot be written to disk if its associated log sequence number is larger than the sequence number of the last log record written to disk.

Recovery from a local failure is easy because only a single transaction is affected. All log records of the transaction are found by searching the log backwards. The undo operation is then applied to each log record of the transaction. If a failure occurs during the recovery process, the undo operation is applied again. The effect of applying the undo operator multiple times is the same as applying undo one time. After completing the undo operations, the recovery manager may offer the user the opportunity to restart the aborted transaction.

Recovery from a system failure is more difficult because all active users are affected. To help you understand recovery from a system failure, Figure 17.11 shows

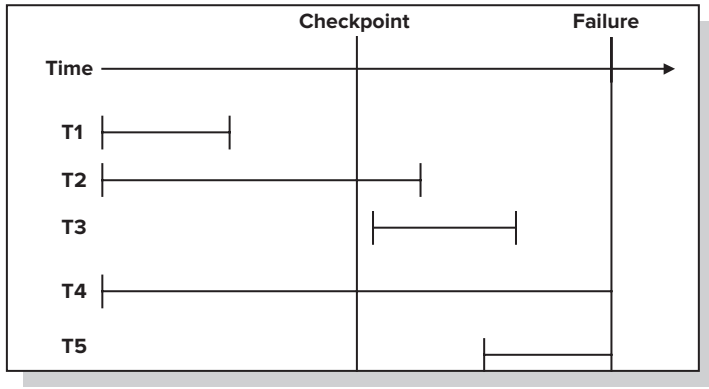


FIGURE 17.11
Transaction Timeline

the progress of transaction types with respect to the commit time, the most recent checkpoint, and the failure. For example, transaction class T1 represents transactions started and finished before the checkpoint (and the failure). No other transaction types are possible.

The immediate update approach may involve both undo and redo operations, as summarized in Table 17-8. To understand the amount of work necessary, remember that log records are stable at checkpoint time and end of transaction and database changes are stable at checkpoint time. Although other database writes occur when a buffer fills, the timing of other writes is unpredictable. T1 transactions require no work because both log and database changes are stable before the failure. T2 transactions must be redone from the checkpoint because only database changes prior to the checkpoint are stable. T3 transactions must be redone entirely because database changes are not guaranteed to be stable even though some changes may be recorded on disk. T4 and T5 transactions must be undone entirely because some database changes after the checkpoint may be recorded on disk.

Deferred Update Approach In the deferred update approach, database updates are written to disk only after a transaction commits. No database writes occur at checkpoint time except for already committed transactions. The advantage of the deferred update approach is that undo operations are not necessary. However, it may be necessary to perform more redo operations than in the immediate update approach.

Deferred Update Approach database updates are written only after a transaction commits. To restore a database, only redo operations are used.

Local failures are handled without any restart work in the deferred update approach. Because no database changes occur until after a transaction commits, the transaction is aborted without any undo work. The recovery manager typically would provide a user with the option of restarting an impacted transaction.

System failures also can be handled without undo operations as depicted in Table 17-9. T4 and T5 transactions (not yet committed) do not require undo operations because no database changes are written to disk until after a transaction commits. T2 and T3 transactions (committed after the checkpoint) require redo operations because it is not known whether all database changes are stable. T2 transactions (started before the checkpoint) must be redone from the first log record rather than just from the checkpoint as in the immediate update approach. Thus, the deferred

TABLE 17-8
Summary of Restart Work for the Immediate Update Approach

Class	Description	Restart Work
T1	Finished before CP	None
T2	Started before CP; finished before failure	Redo forward from the most recent checkpoint
T3	Started after CP; finished before failure	Redo forward from the most recent checkpoint
T4	Started before CP; not yet finished	Undo backwards from most recent log record
T5	Started after CP; not yet finished	Undo backwards from most recent log record

TABLE 17-9
Summary of Restart Work for the Deferred Update Approach

Class	Description	Restart Work
T1	Finished before CP	None
T2	Started before CP; finished before failure	Redo forward from the first log record
T3	Started after CP; finished before failure	Redo forward from the first log record
T4	Started before CP; not yet finished	None
T5	Started after CP; not yet finished	None

update approach requires more restart work for T2 transactions than does the immediate update approach. However, the deferred update approach requires no restart work for T4 and T5 transactions, while the immediate update approach must undo T4 and T5 transactions.

Recovery Example Tables 17-8 and 17-9, although depicting the kind of restart work necessary, do not depict the sequence of log operations. To help you understand the log operations generated at restart time, Table 17-10 shows an example log including checkpoint records. The checkpoint action includes a list of active transactions at the time of the checkpoint. To simplify the restart process, Table 17-10 uses cache consistent checkpoints.

Table 17-11 lists the log operations for the immediate update approach. The immediate update approach begins in the rollback phase. In step 1, the recovery manager adds transaction 4 to the uncommitted list and applies the undo operator to LSN 20. Likewise in step 4, the recovery manager adds transaction 3 to the uncommitted list and applies the undo operator to LSN 17. In step 5, the recovery manager adds transaction 2 to the committed list because the log record contains the COMMIT action. In step 6, the recovery manager takes no action because redo operations will be applied

TABLE 17-10
Example Transaction Log

LSN	TransNo	Action	Time	Table	Row	Column	Old	New
1	1	START	10:29					
2	1	UPDATE	10:30	Acct	10	Bal	100	200
3		CKPT(1)	10:31					
4	1	UPDATE	10:32	Acct	25	Bal	500	400
5	2	START	10:33					
6	2	UPDATE	10:34	Acct	11	Bal	105	205
7	1	INSERT	10:35	Hist	101	*		<1, 400,...>
8	2	UPDATE	10:36	Acct	26	Bal	100	200
9	2	INSERT	10:37	Hist	102	*		<2, 200,...>
10	3	START	10:38					
11	3	UPDATE	10:39	Acct	10	Bal	100	200
12		CKPT(1,2,3)	10:40					
13	3	UPDATE	10:41	Acct	25	Bal	500	400
14	1	COMMIT	10:42					
15	2	UPDATE	10:43	Acct	29	Bal	200	300
16	2	COMMIT	10:44					
17	3	INSERT	10:45	Hist	103	*		<3, 400,...>
18	4	START	10:46					
19	4	UPDATE	10:47	Acct	10	Bal	100	200
20	4	INSERT	10:48	Hist	104	*		<3, 200,...>

TABLE 17-11
Restart Work Using the Immediate Update Approach

Step Number	LSN	Actions
1	20	Undo; add transaction 4 to the uncommitted list
2	19	Undo
3	18	Remove transaction 4 from the uncommitted list
4	17	Undo, add transaction 3 to the uncommitted list
5	16	Add transaction 2 to the committed list
6	15	No action
7	14	Add transaction 1 to the committed list
8	13	Undo
9	12	Note that transaction 3 remains uncommitted
10	11	Undo
11	10	Remove transaction 3 from the uncommitted list
12		Roll forward phase: begin reading the log at the most recent checkpoint record (LSN = 12)
13	14	Remove transaction 1 from the committed list
14	15	Redo
15	16	Remove transaction 2 from the committed list; stop because the committed list is empty

in the roll forward phase. In step 11, the roll backward phase ends because the START record for the last transaction on the uncommitted list has been encountered. In the roll forward phase, the recovery manager uses redo operations for each log record of active transactions.

Table 17-12 lists the log operations for the deferred update approach. The recovery manager begins by reading the log backwards as in the immediate update approach. In step 1, the recovery manager ignores log records 20 through 17 because they involve transactions that did not commit before the failure. In steps 2 and 4, the recovery manager notes that transactions 1 and 2 have committed and will need to be redone during the roll forward phase. In step 3, the recovery manager takes no action because a redo operation will be taken later during the roll forward phase. The roll backward phase continues until START records are found for all committed transactions. In the roll forward phase, the recovery manager uses redo operations for each action of transactions on the committed list. The roll forward phase ends when the recovery manager finds the COMMIT record of the last transaction on the committed list.

Recovery Features in Oracle To depict the recovery features used in an enterprise DBMS, highlights of the recovery manager in Oracle are presented. Oracle uses the immediate update process with incremental checkpoints. No database writes occur at checkpoint time as database writes are periodically written to disk in ascending age order. The log sequence number corresponding to the oldest dirty database page is written to disk to identify the starting point for restart.

Incremental checkpoints involve a trade-off between the frequency of writing dirty database pages versus restart time. More frequent writes slow transaction throughput but decrease restart time. To control this trade-off, Oracle provides a parameter known as the Mean Time to Recover (MTTR) defined as the expected time (in seconds) to recover from a system failure. Decreasing this parameter causes more frequent writing of database pages. To help a DBA set the MTTR parameter, Oracle provides the MTTR Advisor to choose parameter values under various transaction workloads. The MTTR Advisor also determines the log file size that is considered optimal based on the current setting of the MTTR parameter. To monitor the recovery process, Oracle provides a dynamic dictionary view that contains details about the state of the recovery process.

TABLE 17-12
Restart Work Using the Deferred Update Approach

Step Number	LSN	Actions
1	20,19,18,17	No action because transactions 3 and 4 cannot be complete
2	16	Add transaction 2 to the committed and incomplete lists
3	15	No action; redo during the roll forward phase
4	14	Add transaction 1 to the committed and incomplete lists
5	13	No action because transaction 3 cannot be complete
6	12	Note that START records have not been found for transactions 1 and 2
7	11,10	No action because transaction 3 cannot be complete
8	9,8,7,6	No action; redo during the roll forward phase
9	5	Remove transaction 2 from the incomplete list (START record found)
10	4	No action; redo during the roll forward phase
11	3	Note that START record has not been found for transaction 1
12	2	No action. Redo during the roll forward phase
13	1	Remove transaction 1 from the incomplete list; begin the roll forward phase
14	2	Redo
15	4	Redo
16	6	Redo
17	7	Redo
18	8	Redo
19	9	Redo
20	14	Remove transaction 1 from the committed list
21	15	Redo
22	16	Remove transaction 2 from the committed list; end the roll forward phase

17.4 TRANSACTION DESIGN ISSUES

With recovery and concurrency services, it may be surprising that a transaction designer still has important design decisions. A transaction designer can be a database administrator, a programmer, or a programmer in consultation with a database administrator. Design decisions can have a significant impact on transaction processing performance. Knowledge of the details of concurrency control and recovery can make you a better transaction designer. This section describes design decisions available to transaction designers to improve performance.

17.4.1 Transaction Boundary and Hot Spots

A transaction designer develops an application involving some database processing. For example, a designer may develop an application to enable a user to withdraw cash from an ATM, order a product, or register for classes. To build an application, a designer uses the transaction defining statements of SQL and the concurrency control and recovery services of the DBMS. The designer has several alternatives for the **transaction boundary** involving placement of transaction defining statements of SQL in an application.

Transaction Boundary
an important decision in transaction design. A transaction designer places transaction defining SQL statements to divide an application consisting of a collection of SQL statements into one or more transactions.

A transaction designer typically has the option of making one large transaction containing all SQL statements or dividing SQL statements into multiple, smaller transactions. For example, the SQL statements in the ATM transaction can be considered one transaction, as shown in Figure 17.1. Another option is to make each SQL statement a separate transaction. When transaction boundary statements (START TRANSACTION and COMMIT) are not used, each SQL statement defaults to a separate transaction.

Trade-offs in Choosing Transaction Boundaries When choosing a transaction's boundary, the objective is to minimize the duration of the transaction while ensuring satisfaction of critical constraints. DBMSs are designed for transactions of short duration because locking can force other transactions to wait. The duration includes not only the number of reads and writes to the database but the time spent waiting for user responses. Generally, a transaction boundary should not involve user interaction. In the ATM, airline reservation, and online shopping transactions (Figure 17.1, 17.2, and 17.3, respectively), the START TRANSACTION and COMMIT statements could be moved to surround just the SQL part of the pseudo code.

Duration should not compromise constraint checking. Because constraint checking must occur by the end of a transaction, it may be difficult to check some constraints if a transaction is decomposed into smaller transactions. For example, an important constraint in accounting transactions is that debits equal credits. If the SQL statements to post a debit and a credit are placed in the same transaction, then the DBMS can enforce the accounting constraint at the end of the transaction. If they are placed in separate transactions, constraint checking cannot occur until after both transactions are committed.

Hot Spots To understand the effects of transaction boundary choices, hot spots should be identified. Recall that hot spots are common data that multiple users try to change simultaneously. If a selected transaction boundary eliminates (creates) a hot spot, it may be a good (poor) design.

Hot spots can be classified as either system independent or system dependent. System-independent hot spots are parts of a table that many users simultaneously may want to change. Rows, columns, and entire tables can be system-independent hot spots. For example, in the airline reservation transaction (Figure 17.2), the seats-remaining column of popular flight rows is a system-independent hot spot. The seats-remaining column is a hot spot for every DBMS.

System-dependent hot spots depend on the DBMS. Usually, system-dependent hot spots involve parts of the database hidden to normal users. Pages (physical records) containing database rows or index records can often be system-dependent hot spots. For example, some DBMSs lock the next available page when inserting a row into a table. When inserting a new history row in the ATM transaction (Figure 17.1), the next available page of the history table is a system-dependent hot spot. On DBMSs that lock at the row level, no hot spot exists. There are also typically hot spots with commonly accessed index pages.

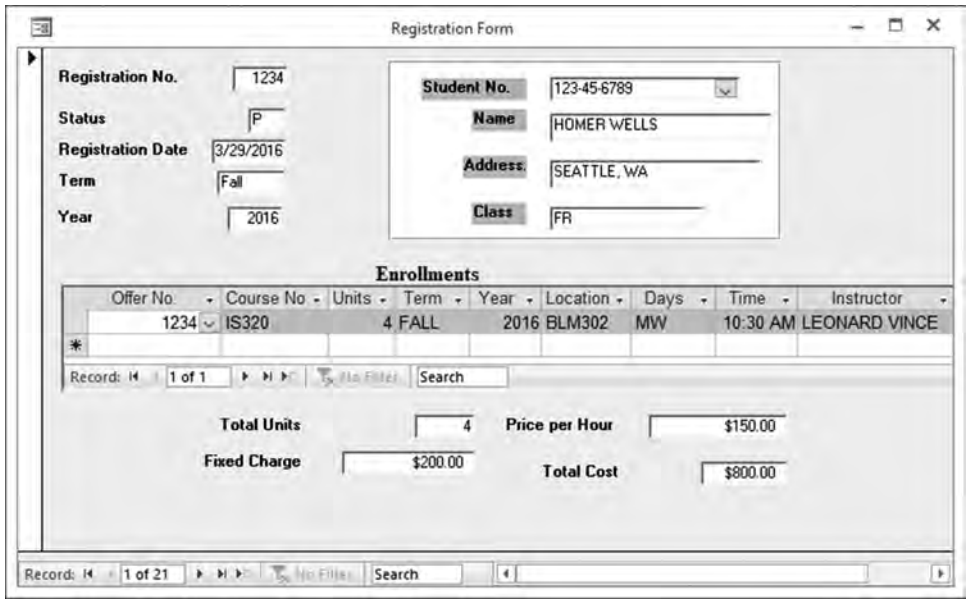
Example Transaction Boundary Design To depict a transaction boundary choice, hierarchical forms provide a convenient context. A hierarchical form represents an application that reads and writes to a database. For example, the registration form of the university database (Figure 17.12) manipulates the *Registration* table in the main form and the *Enrollment* and *Offering* tables in the subform. When using the registration form to enroll in courses, a record is inserted in the *Registration* table after completing the main form. After completing each line in the subform, a row is inserted into the *Enrollment* table and the *OffSeatsRemain* column of the associated *Offering* row is updated. Although the *OffSeatsRemain* column does not appear in the subform, it must be updated after inserting each subform line.

When designing a hierarchical form, the transaction designer has three reasonable choices for the transaction boundary:

(1) The entire form
(2) The main form as one transaction and all subform lines as a second transaction
(3) The main form as one transaction and each subform line as separate transactions.

The third choice is usually preferred because it provides transactions with the shortest duration. However, constraint checking may force the choice to (1) or (2). The registration form involves constraints on an entire registration such as a minimum number

FIGURE 17.12
Example Registration Form



of hours for financial aid and prerequisites for a course. However, these constraints are not critical to check at the end of a transaction. Most universities check these constraints at a later point before the next academic period begins. Thus, choice (3) is the best choice for the transaction boundary.

Processing of operations on the registration form involve several hot spots common to each transaction boundary choice. The *OffSeatsRemain* column in popular *Offering* rows is a system-independent hot spot in any transaction involving the subform lines. The *OffSeatsRemain* column must be updated after each enrollment. The next page in the *Enrollment* table is a system-dependent hot spot in some DBMSs. After each subform line, a row is inserted in the *Enrollment* table. If a DBMS locks the next available page rather than just the new row, all subform line transactions must obtain an exclusive lock on the next available physical record. However, if a DBMS locks at the row level, no hot spot exists because each transaction will insert a different row.

Choice (3) provides another advantage due to reduced deadlock possibilities. In choices (1) and (2), a deadlock may occur when a transaction involves multiple course enrollments. For example, one transaction could obtain a lock on a data communications offering (IS470) and another transaction on a database offering (IS480). The first transaction may then wait for a lock on the IS480 offering, and the second transaction may wait for a lock on the IS470 offering. Choice (3) will be deadlock free if the hot spots are always obtained in the same order by every transaction. For example, if every transaction first obtains a lock on the next available page of *Enrollment* and then obtains a lock on an *Offering* row, then a deadlock will not occur.

Avoiding User Interaction Time Another issue of transaction boundary is user interaction. Normally user interaction should be placed outside of a transaction. Figure 17.13 shows the airline reservation transaction redesigned to place user interaction outside of the transaction. The redesigned transaction will reduce waiting times of concurrent users because locks will be held for less time. However, side effects may occur as a result of removing user interaction. With user interaction outside of a transaction, a seat on the flight may not be available even though the user was just informed about the flight's availability. A database analyst should monitor occurrence of this side effect. If the side effect is rare, the interaction code should remain outside of the transaction. If the side effect occurs with a reasonable frequency, it may be preferable to retain the user interaction as part of the transaction.

```
Get reservation preferences
SELECT departure and return flight rows
If reservation is acceptable Then
   START TRANSACTION
   UPDATE seats remaining of departure flight row
   UPDATE seats remaining of return flight row
   INSERT reservation row
End If
On Error: ROLLBACK
COMMIT
Send receipt to customer
```

FIGURE 17.13
Pseudocode for Redesigned Airline Reservation Transaction

Tennis Court Reservation Case To demonstrate organizational impacts of transaction design choices, the last part of this section presents details about an actual tennis court reservation case. This case depicts elements of transaction design (hot spots, transaction boundary choices, and user waiting time) and impacts of transaction design on business practices.

The Southside Athletic Club[4] contracts for its online website from Tennis Systems Unlimited, an organization serving many tennis clubs with information services. An important part of the website is an online court reservation system. Although the tennis club is relatively small with about 500 members, members have been experiencing difficulty with making court reservations during peak reservation times. Reservations can be made six days in advance beginning at 7AM. During winter months, indoor tennis courts experience high demand especially during the period from 2:30PM until closing (normally 10PM).

Making a reservation involves choosing a date and time (Figure 17.14), selecting a duration (Figure 17.15), and choosing one to three partners (Figure 17.16). In Figure 17.14, a member can choose only an unreserved (white) time slot. After choosing an available time slot and duration, the reservation is subject to partner availability. Members can only appear on one reservation in each day. A small delay typically occurs in the choice of a duration. Longer delays typically occur when choosing partners as a partner must be selected from a list and the system must check partner availability.

The tennis reservation database has a predictable system independent hot spot. Popular reservation times create contention for locks on the corresponding rows in the reservation table. The reservation table contains columns for the unique reservation number, date, start time, duration, court, player1, player2, optional player3, and optional player4.

In the transaction boundary decision, two options were considered. The minimal duration choice involves no waiting time as it begins after selection of all reservation details (date, time, duration, and partners). However, the minimal duration choice has a side effect that a reservation may not be available after selecting all reservation parts. The alternative choice is to begin the transaction after selection of the date, time, and duration. The alternative choice involves wait time for partner selection, but the member will obtain the court after selection of the date, time, and duration. In actual system deployment, the alternative choice was selected due to the importance of stability of court selection. With high contention for court reservations at initial reservation times, member dissatisfaction became high when selected courts were not available after partner selection. To limit transaction duration, a five minute limit was imposed on partner selection time. The player selection window shows the remaining time (Figure 17.16) for player selection to avoid surprises during the reservation process.

Despite careful design, the online reservation system had substantial problems after deployment. Members had difficulty obtaining desired reservations during peak reservation times. The system was unstable and slow with web pages sometimes taking minutes to load. After consultations among club members, club management, and

[4] The names of the tennis club and external software provider have been changed.

FIGURE 17.14
Web Page to Select Court Date and Time

FIGURE 17.15
Web Page to Select Duration

vendor management, it was decided to divide the reservation period into two periods (6AM to 2:30PM starting at 7AM and 2:30PM to 10PM starting at 8AM). In addition, the current day's reservation calendar was preloaded prior to the beginning of a reservation period. System performance has improved as a result of these changes with stable operation and much faster page loading.

17.4.2 Isolation Levels

Two-phase locking prevents the three concurrency control problems described in Section 17.2.2 if all locks are held until the end of the transaction. However, some transactions may not need this level of concurrency control. The lost update problem is the most serious problem and should always be prevented. Some transactions may be able to tolerate conflicts caused by the uncommitted dependency and the inconsistent retrieval problems. Transactions that do not need protection from these problems can release locks sooner and achieve faster execution speed.

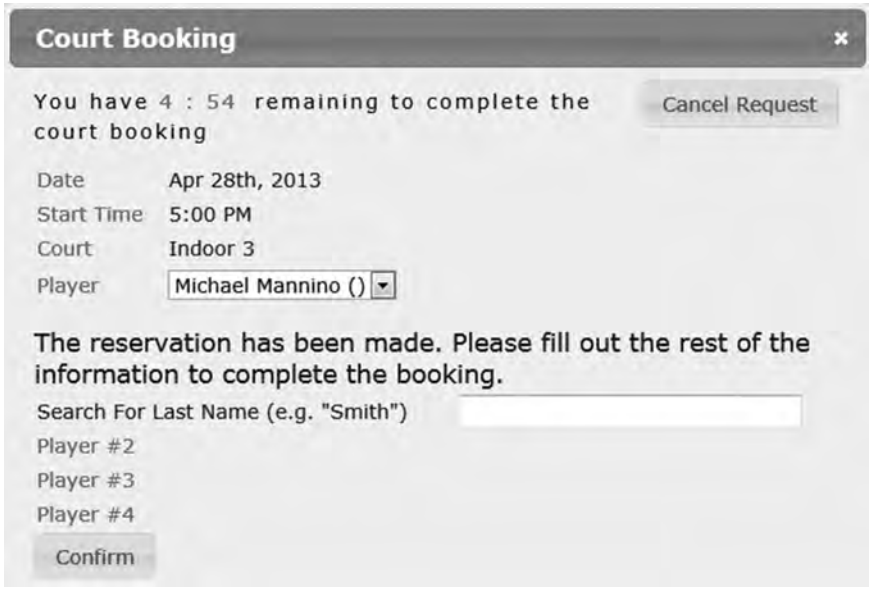


FIGURE 17.16
Web Page to Select Partners

The **isolation level** specifies the degree to which a transaction is separated from the actions of other transactions. A transaction designer can balance concurrency control overhead with potential interference problems by specifying the appropriate isolation level.

Isolation Level
defines the degree to which a transaction is separated from actions of other transactions. A transaction designer can balance concurrency control overhead with interference problems prevented by specifying the appropriate isolation level.

Table 17-13 summarizes the SQL:2016 isolation levels according to the duration and type of locks held. The serializable level prevents all concurrency control problems but involves the most overhead and waiting. To prevent concurrency control problems, predicate locks are used and all locks are long term (held until commit time). Predicate locks that reserve rows specified by conditions in the WHERE clause are essential to prevent phantom read problems. The repeatable read level uses short-term predicate locks to prevent the incorrect summary and the nonrepeatable read problems. The read committed level uses short-term shared locks to enable more concurrent access. However, it only prevents the uncommitted dependency problem and the traditional lost update problem. Because the read uncommitted level does not use locks, it is only appropriate for read-only access to a database.

A transaction designer can specify the isolation level using the SQL SET TRANSACTION statement, as shown in Example 17.1. The SET TRANSACTION statement is usually placed just before a START TRANSACTION statement to alter the default settings for transactions. In Example 17.1, the isolation level is set to READ COMMITTED. The other keywords are SERIALIZABLE, REPEATABLE READ, and READ UNCOMMITTED. Some DBMS vendors do not support all of these levels while other vendors support additional levels.

TABLE 17-13
Summary of Isolation Levels

Level	Exclusive Locks	Shared Locks	Predicate Locks	Problems Permitted
Read Uncommitted	None since read-only	None	None	Only uncommitted dependencies because transactions must be read-only
Read Committed	Long-term	Short-term	None	Scholar's lost updates, incorrect summary, non repeatable reads
Repeatable Read	Long-term	Long-term	Short-term read, Long-term write	Phantom reads
Serializable	Long-term	Long-term	Long-term	None

Example 17.1 (SQL:2016)

SET TRANSACTION Statement to Set the Isolation Level of a Transaction

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
START TRANSACTION
…
COMMIT
```

In SQL:2016, SERIALIZABLE is the default isolation level. For most transactions, this level is recommended because the REPEATABLE READ level provides only a small performance improvement. The READ UNCOMMITTED level is recommended for read-only transactions that can tolerate retrieval inconsistency.

Although SQL:2016 provides SERIALIZABLE as the default level, some DBMS vendors such as Oracle and Microsoft SQL Server use READ COMMITTED as the default level. READ COMMITTED can be a dangerous default level as it permits a variation of the lost update problem known as the scholar's lost update. The word *scholar* is ironic in that the scholar's lost update problem differs only slightly from the traditional lost update problem. Figure 17.17 depicts the scholar's lost update problem. The only essential difference between the scholar's lost update problem and the traditional lost update problem is that transaction A commits before transaction B changes the common data. Thus, the scholar's lost update is a serious potential problem that should not be permitted for most transactions.

17.4.3 Timing of Integrity Constraint Enforcement

Besides setting the isolation level, SQL allows control of the timing of integrity constraint enforcement. By default, constraints are enforced immediately after each INSERT, UPDATE, and DELETE statement. For most constraints such as primary and foreign keys, immediate enforcement is appropriate. If a constraint is violated, a DBMS issues a rollback operation on a transaction. The rollback restores the database to a consistent state as the ACID properties ensure consistency at the end of a transaction.

FIGURE 17.17
Example Scholar's Lost Update Problem

Transaction A	Time	Transaction B
Obtain S lock on SR	T_1	
Read SR (10)	T_2	
Release S lock on SR	T_3	
If SR > 0 then SR = SR − 1	T_4	
	T_5	Obtain S lock on SR
	T_6	Read SR (10)
	T_7	Release S lock on SR
	T_8	If SR > 0 then SR = SR − 1
Obtain X lock on SR	T_9	
Write SR (9)	T_{10}	
Commit	T_{11}	
	T_{12}	Obtain X lock on SR
	T_{13}	Write SR (9)

For complex constraints, immediate enforcement may not be appropriate. For example, a faculty workload constraint ensures that each faculty member teaches between three and nine units each semester. If a transaction assigns an entire workload, checking the constraint should be deferred until the end of the transaction. For these kinds of complex constraints, constraint timing should be specified.

Deferred Constraint Checking: enforcing integrity constraints at the end of a transaction rather than immediately after each manipulation statement. Complex constraints may benefit from deferred checking.

Constraint timing involves both constraint definition and transaction definition. SQL provides an optional constraint timing clause that applies to primary key constraints, foreign key constraints, uniqueness constraints, check constraints, and assertions. A database administrator typically uses the constraint timing clause for constraints that may need deferred checking. Constraints that never need deferred checking do not need the timing clause as the default is NOT DEFERRABLE. The timing clause defines the deferability of a constraint along with its default enforcement (deferred or immediate), as shown in Examples 17.2 and 17.3.

Example 17.2 (SQL:2016)

Timing Clause for the *FacultyWorkLoad* Assertion

The constraint is deferrable and the default enforcement is deferred.

```
CREATE ASSERTION FacultyWorkLoad
 CHECK (NOT EXISTS
  ( SELECT Faculty.FacNo, OffTerm, OffYear
      FROM Faculty, Offering, Course
      WHERE Faculty.FacNo = Offering.FacNo
        AND Offering.CourseNo = Course.CourseNo
      GROUP BY Faculty.FacNo, OffTerm, OffYear
      HAVING SUM(CrsUnits) < 3 OR SUM(CrsUnits) > 9 ) )
  DEFERRABLE INITIALLY DEFERRED
```

Example 17.3 (SQL:2016)

Timing Clause for the *OfferingConflict* Assertion

The constraint is deferrable and the default enforcement is immediate.

```
CREATE ASSERTION OfferingConflict
 CHECK (NOT EXISTS
  ( SELECT O1.OfferNo
      FROM Offering O1, Offering O2
      WHERE O1.OfferNo <> O2.OfferNo
        AND O1.OffTerm = O2.OffTerm
        AND O1.OffYear = O2.OffYear
        AND O1.OffDays = O2.OffDays
        AND O1.OffTime = O2.OffTime
        AND O1.OffLocation = O2.OffLocation  ) )
DEFERRABLE INITIALLY IMMEDIATE
```

For each transaction, the transaction designer may specify whether deferrable constraints are deferred or immediately enforced using the SET CONSTRAINTS statement. Normally the SET CONSTRAINTS statement is placed just after the START TRANSACTION statement as shown in Example 17.4. The SET CONSTRAINTS statement is not necessary for deferrable constraints with deferred default enforcement. For example, if the *FacultyWorkLoad* assertion is deferred, no SET CONSTRAINTS statement is necessary because its default enforcement is deferred.

Example 17.4 (SQL:2016)

SET CONSTRANTS Statements for Several Transactions

```
START TRANSACTION
SET CONSTRAINTS FacultyWorkLoad IMMEDIATE
…
COMMIT

START TRANSACTION
SET CONSTRAINTS OfferingConflict DEFERRED
…
COMMIT
```

Implementation of the constraint timing part of SQL is highly variable. Most DBMSs do not support the constraint timing part exactly as specified in the standard. Many DBMSs have different syntax and proprietary language extensions for constraint timing.

17.4.4 Save Points

Some transactions have tentative actions that can be cancelled by user actions or other events. For example, a user may cancel an item on an order after discovering that the item is out of stock. Because the ROLLBACK statement removes all transaction changes, it cannot be used to remove just a canceled item if a transaction involves an entire order. A transaction designer can code statements to explicitly delete the tentative parts of a transaction, but this coding can be tedious and involve excessive overhead.

SQL:2016 provides the SAVEPOINT statement to allow partial rollback of a transaction. A transaction designer uses the SAVEPOINT keyword followed by the save point name to establish an intermediate point in a transaction. To undo work since a particular save point, the ROLLBACK TO SAVEPOINT keywords can be used followed by the save point name. Figure 17.18 depicts the usage of a save point. Typically, partial rollback is used conditionally depending on a user action or an external event.

Save points are also used internally by some enterprise DBMSs to resolve deadlocks. Instead of rolling back an entire transaction, the DBMS rolls back a transaction to its last save point. Implicit save points can be used by a DBMS after each SQL statement to reduce the amount of lost work.

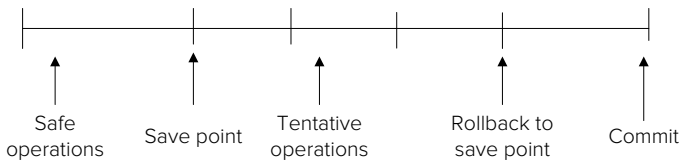


FIGURE 17.18
Transaction Flow with a Save Point

17.4.5 Relaxed Transaction Consistency Model

Some high performance applications in ecommerce can tolerate eventual consistency to increase availability. For example in ATM applications, the reconciliation of amount withdrawn with the account balance may occur hours after an ATM transaction completes. A limit such as $400 is imposed on ATM withdrawals to control the exposure of a financial institution to withdrawing more than an account balance. The application can control the timing of the eventual reconciliation of the withdrawal amount and account balance.

To relax the standard ACID consistency requirement and tradeoff consistency against availability, many NoSQL database products use the **BASE principle**. NoSQL database products use simplified database models[5], less stringent transaction processing models, and distributed processing to reduce bottlenecks for dealing with the demands of big data. The BASE principle has three components as shown in the following list.

BASE Principle
a principle of relaxed data consistency in distributed transaction processing used by NoSQL database products. The BASE (Basically Available, Soft state, and Eventual consistency) principle forces some responsibility for consistency on applications with the benefit of improved availability and performance.

- Basically Available indicates that the system emphasizes availability over consistency. Reduced consistency requirements combined with partitioning of data across storage systems supports higher levels of availability for at least part of the data.
- Soft state indicates that the state of the system may be partially inconsistent at times across storage systems. Each application must be developed without relying on standard ACID consistency levels for all replicated data that it may use. With the BASE principle, data consistency is the developer's problem, not the responsibility of the database software.
- Eventual consistency indicates that distributed and replicated data copies will become consistent over time typically without guarantees about the timing of consistency. If a time limit is required, applications must enforce the constraint using procedures outside the DBMS.

Message queuing can provide eventual consistency if messages are persistent or order independent. Persistent messages must be guaranteed delivery as part of the transaction performing an update. For the ATM transaction, a message to update the balance would be part of the transaction performing the ATM action (withdrawal or payment). If the overhead of persistent messages is unacceptable, messages can be made order independent by tracking message completions. Additional overhead will occur when processing messages but not during time-sensitive transactions.

Overall, the BASE transaction model puts the onus on applications to implement the appropriate level of consistency possibly aided by tools provided by a DBMS. This additional level of responsibility reduces productivity of application developers, but it provides a mechanism to increase performance and availability. For some high value applications, increased performance and availability may be more important than reduced development productivity.

17.5 WORKFLOW MANAGEMENT

Transaction management is part of a larger area known as workflow management. Workflow management supports business processes, both automated and human performed. In contrast, transaction management supports properties of automated database processing. This section presents workflow management to provide a broader perspective for transaction management. This section first describes workflows, a broader notion than database transactions. This section then discusses enabling technologies for workflow management showing how transaction management is an important component.

[5] Chapter 19 covers data representation and manipulation aspects of NoSQL DBMSs.

17.5.1 Characterizing Workflows

Workflow
A collection of related tasks structured to accomplish a business process.

Workflows support business processes such as installing pay TV service, obtaining a loan, and ordering custom products. **Workflows** consist of tasks that can be performed by computers (software and hardware), humans, or a combination. For example, in installing internet service, software determines the time of a service appointment and updates a scheduling database, while a technician inspects the box to determine whether a problem exists. A workflow defines the order of performing the tasks, the conditions for tasks to be performed, and the results of performing tasks. For example, providing internet service involves an initial customer contact, an optional service visit, billing, and payment collection. Each of these tasks can have conditions under which they are performed and may result in actions such as database updates and invocation of other tasks.

Many different kinds of workflows exist. Sheth, Georgakopoulos, and Hornrick (1995) classify workflows as human-oriented versus computer-oriented, as depicted in Figure 17.19. In human-oriented workflows, humans provide most of the judgment to accomplish work. The computer has a passive role to supply data to facilitate human judgment. For example, in processing a loan, loan officers often determine the status of loans when a customer does not meet standard criteria about income and debt. Consultation with underwriters and credit personnel may be necessary. To support human-oriented workflows, electronic communication software such as email, chat, and document annotation may be useful. In computer-oriented tasks, software determines the processing of work. For example, software for an ATM transaction determines whether a customer receives cash or is denied the request. To support computer-oriented workflows, transaction management is a key technology.

Another way to classify workflows is by task complexity versus task structure as depicted in Figure 17.20. Task complexity involves difficulty of performing individual tasks. For example, the decision to grant a loan may involve complex reasoning using many variables. In contrast, processing a product order may involve requesting product data from a customer. Task structure involves the relationships among tasks. Workflows with complex conditions have high structure. For example, processing an insurance claim may have conditions about denying a claim, litigating a claim, and investigating a claim.

FIGURE 17.19
Classification of Workflow by Task Performance (Adapted from Sheth, Georgakopoulos, and Hornrick (1995))

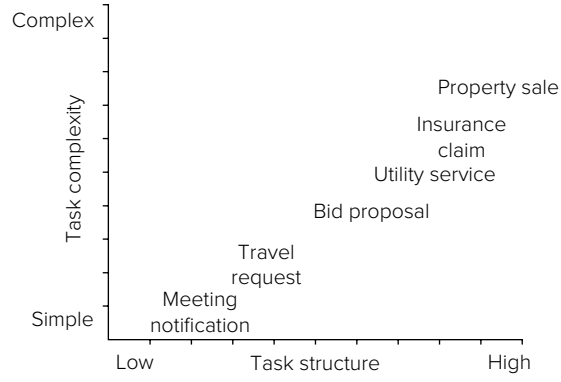


FIGURE 17.20
Classification of Workflow by Task Structure and Complexity (Adapted from Sheth, Georgakopoulos, and Hornrick (1995))

17.5.2 Enabling Technologies

To support the concept of a workflow discussed in the previous section, three enabling technologies are important: (1) distributed object management, (2) workflow specification, and (3) customized transaction management. Transaction management as described in previous sections fits as part of the third technology. The remainder of this section elaborates on each technology.

Distributed Object Management Workflows can involve many types of data in remote locations. For example, data can include photos of an insurance claim, x-rays supporting a diagnosis, and an appraisal documenting a property for a loan application. These types of data are not traditionally managed by DBMSs. A new class of DBMSs known as object DBMSs has been developed to manage diverse types of data. Chapter 19 describes object DBMSs that manage diverse data types.

In addition to new data types, data are typically not stored at one location and may be controlled by different DBMSs. For example, to support a loan application, a loan officer uses a credit report from a credit bureau, an appraisal from a certified appraiser, and loan processing guidelines from government agencies. Accessing and controlling distributed data can be difficult. Chapter 18 describes important principles of managing distributed data. Difficulties also can arise because the data may be controlled by different systems, some of which may not support SQL.

Workflow Specification and Implementation To support workflows, the structure of tasks must be properly represented and implemented. Representing a workflow involves identifying the tasks and specifying the relationships among tasks. A complex task can involve a hierarchy of subtasks. A complex workflow can involve many tasks with numerous relationships. Constraints, rules, and graphical notation can be used to depict the task order and completion. One use of task relationships is to define constraints among transactions. For example, workflow specification can indicate that a student should be denied financial aid unless enrolling for a minimum number of hours by a specified date.

After a workflow is specified, it must be efficiently implemented. The implementation may involve diverse hardware, software, and people. A major challenge is to make diverse components communicate efficiently. Optimizing workflows through reengineering has become an important concern in many organizations. Optimization may involve removing duplicate tasks and increasing the amount of parallel work.

A number of software systems have been developed to support workflow specification. The main function of software under the name workflow management is to support workflow specification and implementation.

Customized Transaction Management The earlier sections of this chapter described how DBMSs support ACID transactions. The ACID properties are indeed important and widely supported by DBMSs. However, to support workflows, the ACID properties may not be sufficient. The following list identifies shortcomings of traditional ACID transactions for workflow management:

- Some workflows involve tasks with a long duration because of user interaction. Traditional transaction management may not work well for these conversational transactions.
- Some tasks may involve subtasks, changing the notion of atomicity. The idea of nested transactions (transactions inside transactions) has been proposed for tasks with complex structures.
- Some tasks may be performed by legacy systems that do not support the ACID properties.
- Some workflows may require tasks to be undone after they are complete. In accounting systems, it is common for compensating transactions to correct

mistakes. For example, returning a defective product removes the effect of the original product order.

Some DBMSs support some of these extensions now. For example, Microsoft SQL Server provides nested transactions to support transactions that reside in procedures. Oracle provides autonomous transactions to allow interruption of a transaction by another transaction. In addition, SQL:2016 provides save points so that a ROLLBACK statement can undo only part of a transaction. This extension can reduce the amount of work lost when a long transaction fails.

To more fully support workflow management, transaction management should be customized according to workflow requirements. The transaction properties supported for a workflow should be part of the workflow specification, not hardwired into software supporting workflow management. DBMSs supporting workflow management may need to be more flexible. DBMSs might support different kinds of transactions or even allow transaction properties to be specified. Event processing in DBMSs can be used to support some features such as compensating transactions. Most DBMSs would need major extensions to support customized transaction management. So far, market demand has not supported major extensions for custom transaction management.

CLOSING THOUGHTS

This chapter has described concepts underlying database transactions, services provided by a DBMS to support transactions, and design skills for transaction designers. A transaction is a user-defined unit of work with any number of reads and writes to a database. To define a transaction, several new SQL statements were introduced including START TRANSACTION, COMMIT, and ROLLBACK. DBMSs ensure that transactions are atomic (all or nothing), consistent (satisfy integrity constraints after completion), isolated (no interference from concurrent users), and durable (survive failures). To ensure these properties of transactions, DBMSs provide services for concurrency control (making a database seem like a single-user system) and recovery management (automatically restoring a database after a failure). These powerful services are not free as they can consume large amounts of computing resources and add substantial cost for DBMS licensing.

Although the concurrency and recovery services provided by a DBMS are transparent, you should understand some details of these services. Knowledge of these services can help you allocate computing resources, select a DBMS that provides the appropriate level of transaction support, and design efficient transactions. For concurrency control, you should understand the objective, interference problems, and the two-phase locking protocol. For recovery management, you should understand the kinds of failures, the redundant storage needed for recovery, and the amount of work to restore a database after a failure.

To apply your knowledge of transaction management, this chapter demonstrated principles of transaction design. The most important choice in transaction design is the selection of a transaction boundary. The objective of choosing a transaction boundary is to minimize duration subject to the need for constraint checking. Critical constraints such as debit-credit in accounting systems may dictate that an application remains as one transaction rather than be split into smaller transactions. A transaction boundary can also be shortened by removing user interaction. Other important decisions involve the isolation level, the timing of constraint enforcement, and save points for partial rollback. SQL syntax for these elements was shown in the chapter. For even more control for critical applications, the BASE principle used in NoSQL products allows applications to balance availability, consistency, and performance at the cost of lower software productivity.

As a transaction designer, you should remember that transactions are only one kind of support for organizational work. Workflow management addresses issues beyond transaction management such as dependencies among transactions, different kinds of transactions, and annotation of work.

This chapter has described DBMS services to support transaction processing, the operating side of databases. As described in Chapters 12 and 15, DBMSs also provide extensive services for data warehouses that support for management decision making. You should contrast the requirements to support transaction processing for operational decision making with the requirements to support data warehousing for tactical and strategic decision making.

REVIEW CONCEPTS

- Transactions containing a user-specified collection of database reads and writes
- SQL statements to define transactions: START TRANSACTION, COMMIT, ROLLBACK, and SAVEPOINT
- ACID properties of transactions: atomic, consistent, isolated, durable
- Transparent services to hide inner details of transaction management
- Concurrency control to support simultaneous usage of a database
- Recovery management to restore a database to a consistent state after a failure
- Concurrency control objective: maximizing transaction throughput while preventing interference problems
- Interference on hot spots, common data manipulated by concurrent users
- Interference problems: lost update, uncommitted dependency, inconsistent retrieval
- Concurrency control manager to grant, release, and analyze locks
- Compatibility among basic locks (shared and exclusive)
- Intent locks supporting more concurrency on coarse items than shared or exclusive locks and efficient detection of conflicts among locks on items of varying granularity
- Compatibility among intent locks (intent shared, intent exclusive, and shared with intent exclusive)
- Growing and shrinking phases of two-phase locking (2PL)
- Resolution of deadlocks with either deadlock detection or timeout followed by transaction restart
- Update locks to reduce the frequency of deadlocks
- Optimistic concurrency control approaches when interference is rare
- In-memory transaction processing using optimistic concurrency control for mission critical applications
- Volatile versus nonvolatile storage
- Effect of local, system, and media failures
- Force writing to control timing of database writes
- Redundant storage for recovery: log, checkpoint, and backup
- Types of checkpoints: cache consistent, fuzzy, and incremental
- Tradeoff in frequency of checkpoints: transaction throughput versus restart time
- Amount of restart work in the immediate update and the deferred update recovery approaches
- Selecting a transaction boundary to minimize duration while enforcing critical integrity constraints

- Removing user interaction to reduce transaction duration except when resource stability is important
- Identifying system-independent and system-dependent hot spots in transactions
- Isolation levels for balancing potential interference against the corresponding overhead of concurrency control
- Potential for lost data when using the READ COMMITTED isolation level
- Constraint timing specification to defer enforcement of integrity constraints until end of transaction
- Save points for partial rollback of a transaction
- BASE (Basically Available, Soft state, and Eventual consistency) principle used by NoSQL database products forcing some responsibility for data consistency on applications with the benefit of increased performance and availability
- Workflow management to support collaborative work

QUESTIONS

1. What does it mean to say that a transaction is a user-defined concept? Why is it important that transactions are user-defined?
2. Identify transactions with which you have interacted in the last week.
3. Explain the purpose of the SQL statements START TRANSACTION, COMMIT, and ROLLBACK. How do these statements vary across DBMSs?
4. Briefly explain the meaning of the ACID properties. How do concurrency control and recovery management support the ACID properties?
5. Briefly explain the meaning of transparency as it relates to computer processing. Why is transparency important for concurrency control and recovery management?
6. What costs are associated with concurrency control and recovery management? In what role, database administrator or database programmer, would you assess these costs?
7. What is the objective of concurrency control? How is the measure used in the objective related to waiting time?
8. What is a hot spot? How are hot spots related to interference problems?
9. Discuss the consequences of each kind of interference problem. Which problem seems to be the most serious?
10. What is a lock? Briefly explain the differences between shared (S) and exclusive (X) locks.
11. What operations are performed by the lock manager?
12. What is a deadlock and how are deadlocks handled?
13. What is locking granularity? What are the trade-offs of holding locks at a finer level versus a coarser level of granularity?
14. What is an intent lock? Why are intent locks used on items of coarse granularity?
15. Why is the third condition of 2PL typically simplified so that locks are released at the end of a transaction?
16. What is the appeal of optimistic concurrency control approaches? Why might optimistic concurrency control approaches not be used even if they provide better expected performance?
17. Explain the difference between volatile and nonvolatile storage.
18. Explain the effects of local, system, and device failures on active and past transactions.

19. Why is force writing the most fundamental tool of recovery management?
20. What kind of redundant data is stored in a log? Why is management of the log critical to recovery?
21. What is the checkpoint interval? What is the trade-off in determining the checkpoint interval?
22. What processing occurs when a cache-consistent checkpoint occurs?
23. What is a fuzzy checkpoint? What are the advantages of a fuzzy checkpoint as compared to a cache-consistent checkpoint?
24. What is an incremental checkpoint? How can the amount of restart work be controlled with incremental checkpoints?
25. What restart work is necessary for a media failure?
26. What restart work is necessary for local and system failures under the immediate update approach?
27. What restart work is necessary for local and system failures under the deferred update approach?
28. What is a transaction boundary? Why can an inappropriate choice for transaction boundary lead to poor performance?
29. What criteria should be used in selecting a transaction boundary?
30. Why must constraints such as the debit-credit constraint be enforced as part of a transaction rather than between transactions?
31. Explain the difference between system-independent and system-dependent hot spots. Why is it useful to identify hot spots?
32. Explain the three choices for transaction boundary of a hierarchical form.
33. How can deadlock possibility be influenced by the choice of a transaction boundary?
34. What side effect can occur by moving user interaction outside a transaction boundary?
35. What is the purpose of the SQL isolation levels?
36. How do the isolation levels achieve more concurrent access?
37. What isolation level can be dangerous and why?
38. Provide an example of a constraint for which deferred enforcement may be appropriate.
39. What SQL statements and clauses involve constraint timing specification?
40. What is the role of the DBA in specifying constraint timing?
41. What is the role of the database programmer in specifying constraint timing?
42. What is the purpose of a save point?
43. How can a save point be used in deadlock resolution?
44. What is a workflow and how is it related to database transactions?
45. What are the differences between human-oriented and computer-oriented workflows?
46. Provide an example of a workflow with high task complexity and another example with high task structure.
47. Discuss the enabling technologies for workflow management. What role does transaction management play in workflow management?
48. What are limitations of transaction management to support workflows?
49. What is the relationship between incremental checkpoints and recovery processes?
50. What level of involvement is necessary to utilize recovery and concurrency control services provided by a DBMS?

51. Should a transaction design always eliminate user interaction? Please explain your answer.
52. Briefly define the three types of intent locks.
53. Why does an intent shared (IS) lock not conflict with a shared with intent exclusive (SIX) lock?
54. How does the 2PL protocol change for locking granularity with intent locks?
55. How do update locks reduce frequency of deadlock?
56. What are the components of the BASE principle?
57. Why do NoSQL database products support the BASE principle?
58. What is in-memory transaction processing?
59. What concurrency control approach is used by in-memory transaction processing?
60. What costs occur to an organization using in-memory transaction processing?

PROBLEMS

The problems provide practice using transaction-defining SQL statements, testing your knowledge of concurrency control and recovery management, and analyzing design decisions about transaction boundaries and hot spots.

1. Identify two transactions that you have encountered recently. Define pseudo code for the transactions in the style of Figures 17.1, 17.2, and 17.3.
2. Identify hot spots in your transactions from problem 1.
3. Using a timeline, depict a lost update problem using your transactions from problem 1 if no concurrency control is used.
4. Using a timeline, depict an uncommitted dependency problem using your transactions from problem 1 if no concurrency control is used.
5. Using a timeline, depict a nonrepeatable read problem using your transactions from problem 1 if no concurrency control is used.
6. Explain whether deadlock would be a problem using your transactions from problem 1 if locking is used. If a deadlock is possible, use a timeline to demonstrate a deadlock with your transactions.
7. Use the following Accounting Database tables and the Accounting Register to answer problems 7.1 to 7.7. Comments are listed after the tables and the form.

   Account (AcctNo, Name, Address, Balance, LastCheckNo, StartDate)
   Entry (EntryNo, *AcctNo*, Date, Amount, Desc)
   Category (CatNo, Name, Description)
   EntryLine (*EntryNo*, *CatNo*, Amount, Description)

Accounting Register for Wells Fargo Credit Line			
Entry No.	E101	**Date:**	3/11/2017
Description:	Purchases at OfficeMax	**Amount:**	$442.00
Invoice No.	I101		

Category	**Description**	**Amount**
Office supplies	Envelopes	$25.00
Equipment	Fax machine	$167.00
Computer software	MS Office upgrade	$250.00

- The primary keys in the tables are underlined. The foreign keys are italicized.
- The Accounting Register records activities on an account, such as a line of credit or accounts receivable. The Accounting Register is designed for use by the accounting department of moderate-size businesses. The sample form shows one recorded entry, but a register contains all recorded entries since the opening of the account.
- The main form is used to insert a row into the *Entry* table and update the *Balance* column of the *Account* table. Accounts have a unique name (Wells Fargo Line of Credit) that appears in the title of the register. Accounts have other attributes not shown on the form: a unique number (name is also unique), start date, address, type (Receivable, Investment, Credit, Checking, etc.) and current balance. The Amount field in the main form is not computed. It must be entered by the user because the user is not required to allocate the amount to categories in the subform.
- In the subform, the user optionally allocates the total amount of the entry to categories. The *Category* field is a combo box. When the user clicks on the category field, the category number and name are displayed. Entering a new subform line inserts a row into the *EntryLine* table. If one or more subform rows are entered, the sum of the amount on the subform rows must equal the amount entered in the main form.
- The *Description* field in the subform describes a row in the *EntryLine* table rather than the *Category* table.

7.1 What are the possible transaction boundaries for the Accounting Register form?

7.2 Select a transaction boundary from your choices in problem 7.1. Justify your choice using the criteria defined in Section 17.4.1.

7.3 Identify system-independent hot spots that result from concurrent usage (say, many clerks in the accounting department) of the Accounting Register. For each hot spot, explain why it is a hot spot.

7.4 Identify system-dependent hot spots that result from concurrent usage (say many clerks in the accounting department) of the Accounting Register. You may assume that the DBMS cannot lock finer than a database page.

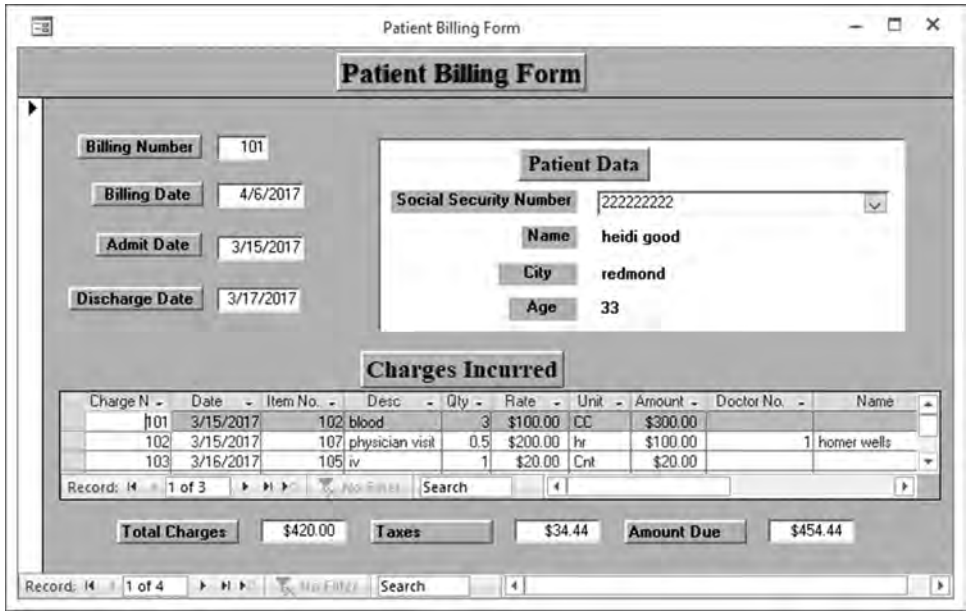
7.5 Describe a lost update problem involving one of your hot spots that could occur with concurrent usage of the Accounting Register. Use a timeline to depict your example.

7.6 Describe a dirty read situation involving one of your hot spots that could occur with concurrent usage of the Accounting Register. Use a timeline to depict your example.

7.7 Is deadlock likely to be a problem with concurrent usage of the Accounting Register? Consider the case where locks are held until all subform lines are complete. Why or why not? If deadlock is likely, provide an example as justification. Would there still be a deadlock problem if locks were held only until completion of each line on the subform? Why or why not?

8. Use the following Patient Database tables and the Patient Billing Form to answer problems 8.1 to 8.4. Comments are listed after the tables and the form.

Patient(PatSSN, PatName, PatCity, PatAge)
Doctor(DocNo, DocName, DocSpecialty)
Bill(BillNo, *PatSSN*, BillDate, AdmitDate, DischargeDate)
Charge(ChgNo, *BillNo*, *ItemNo*, ChgDate, ChgQty, *Doc*No)
Item(Itemno, ItemDesc, ItemUnit, ItemRate, ItemQOH)



- The main form is used to insert a record into the *Bill* table. Fields from the *Patient* table are read-only in the main form.
- The subform can be used to insert a new row into the *Charge* table. Fields from the *Doctor* and the *Item* tables are read-only.
- When a subform line is entered, the associated item row is updated. The form field *Qty* affects the current value in the field *ItemQOH* (item quantity on hand).

8.1 What are the possible transaction boundaries for the Patient Billing Form?

8.2 Select a transaction boundary from your choices in problem 8.1. Justify your choice using the criteria defined in Section 17.4.1.

8.3 Identify system-independent hot spots that result from concurrent usage (say many health providers) of the Patient Billing Form. For each hot spot, explain why it is a hot spot.

8.4 Identify system-dependent hot spots that result from concurrent usage of the Patient Billing Form. You may assume that the DBMS cannot lock finer than a database page.

9. Use the following Airline Reservation database tables and the Flight Reservation Form to answer problems 9.1 to 9.4. Comments are listed after the tables and the form.

Flight(FlightNo, *DepCityCode*, *ArrCityCode*, DepTime, ArrTime, FlgDays)
FlightDate(*FlightNo*, FlightDate, RemSeats)
Reservation(ResNo, *CustNo*, ResDate, Amount, CrCardNo)
ReserveFlight(*ResNo*, *FlightNo*, *FlightDate*)
Customer(CustNo, CustName, Coststreet, CustCity, CustState, CustZip)
City(CityCode, CityName, Altitude, AirportConditions)

- The primary keys in the tables are underlined. The foreign keys are italicized. Note that the combination of *ResNo*, *FlightNo*, and *FlightDate* is the primary key of the *ReserveFlight* table. The combination of *FlightNo* and *FlightDate* is a foreign key in the *ReserveFlight* table referring to the *FlightDate* table.
- The Flight Reservation Form is somewhat simplified as it accommodates only a single class of seating, no reserved seats, and no meals. However, commuter and low-cost airlines often have these restrictions.

Flight Reservation Form					
Reservation No.		R101		**Today's Date:**	8/26/2017
Credit Card No.		CC101		**Amount:**	$442.00
Customer No.		C101		**Customer Name**	Jill Horn
		Flight Schedule			
Flight No.	**Date**	**Dep City**	**Dep Time**	**Arr City**	**Arr Time**
F101	8/26/2017	DNV	10:30AM	CHG	11:45AM
F201	8/31/2017	CHG	10:00AM	DNV	1:20PM

- The main form is used to insert a record into the *Reservation* table. The fields from the *Customer* table are read-only.
- The subform is used to insert new rows in the *ReserveFlight* table and update the field *RemSeats* in the *FlightDate* table. The fields from the *Flight* table are read-only.

9.1 Select a transaction boundary for the Flight Reservation Form. Justify your choice using the criteria defined in Section 17.4.1.

9.2 Identify system-independent hot spots that result from concurrent usage (say many reservation agents) of the Flight Reservation Form. For each hot spot, explain why it is a hot spot.

9.3 Identify system-dependent hot spots that result from concurrent usage of the Flight Reservation Form. You may assume that the DBMS cannot lock finer than a database page.

9.4 Is deadlock likely to be a problem with concurrent usage of the Flight Reservation Form? If deadlock is likely, provide an example as justification.

10. The following timeline shows the state of transactions with respect to the most recent backup, checkpoint, and failure. Use the timeline when solving the problems in subparts of this problem (10.1 to 10.5).

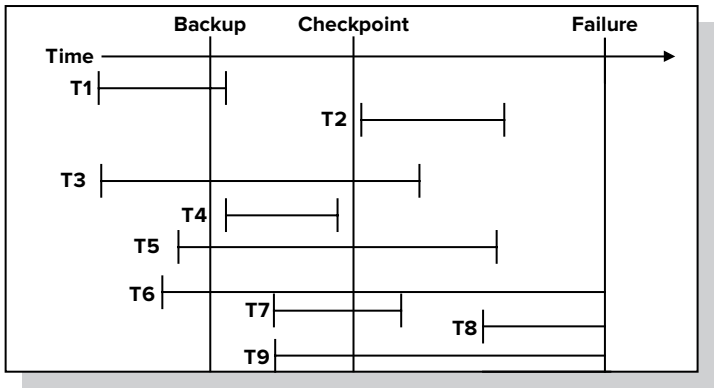
10.1 Describe the restart work if transaction T3 is aborted (with a ROLLBACK statement) after the checkpoint but prior to the failure. Assume that the recovery manager uses the deferred update approach.

10.2 Describe the restart work if transaction T3 is aborted (with a ROLLBACK statement) after the checkpoint but prior to the failure. Assume that the recovery manager uses the immediate update approach.

10.3 Describe the restart work if a system failure occurs. Assume that the recovery manager uses the deferred update approach.

10.4 Describe the restart work if a system failure occurs. Assume that the recovery manager uses the immediate update approach.

10.5 Describe the restart work if a device failure occurs.



11. Use the Transaction Process Council website to review transaction processing benchmarks. Why has the debit-credit benchmark been superseded by other benchmarks? How many transactions per minute are reported for various DBMSs? Inspect the code for one or more benchmark transactions. Can you identify hot spots in the transactions?
12. Redesign the ATM transaction (Figure 17.1) to remove user interaction. Please comment on any adverse side effects that may result from removing user interaction.
13. Redesign the online shopping transaction (Figure 17.3) to remove user interaction. Please comment on any adverse side effects that may result from removing user interaction.
14. Why do some enterprise DBMSs use READ COMMITTED as the default isolation level? Try to reason about the advantages and disadvantages about using this level as the default isolation level. In your analysis, you should think carefully about the significance of the scholar's lost update problem.
15. Using the following transaction log, create a table to list the log operations for the immediate update approach. You should use Table 17-10 as the format for your answer.
16. Using the transaction log from problem 15, create a table to list the log operations for the deferred update approach. You should use Table 17-11 as the format for your answer.

LSN	TransNo	Action	Time	Table	Row	Column	Old	New
1	1	START	2:09:20					
2	1	INSERT	2:09:21	Resv	1001	*		<101, 400,...>
3	2	START	2:09:22					
4	1	UPDATE	2:09:23	Flight	2521	SeatsRem	10	9
5	2	INSERT	2:09:24	Resv	1107	*		<101, 400,...>
6	2	UPDATE	2:09:25	Flight	3533	SeatsRem	3	2
7	3	START	2:09:26					
8	1	INSERT	2:09:27	Resv	1322	*		<102, 225,...>
9	1	UPDATE	2:09:28	Flight	4544	SeatsRem	15	14
10		CKPT(1,2,3)	2:09:29					
11	2	INSERT	2:09:30	Resv	1255	*		<111, 500,...>
12	2	UPDATE	2:09:31	Flight	3288	SeatsRem	2	1
13	1	COMMIT	2:09:32					
14	3	INSERT	2:09:33	Resv	1506	*		<151, 159,...>
15	3	UPDATE	2:09:34	Flight	3099	SeatsRem	50	49
16	4	START	2:09:36					
17	3	INSERT	2:09:37	Resv	1299	*		<222, 384,...>
18	3	UPDATE	2:09:38	Flight	4522	SeatsRem	25	24
19	4	INSERT	2:09:39	Resv	1022	*		<222, 384,...>
20		CKPT(2,3,4)	2:09:40					
21	2	COMMIT	2:09:41					
22	4	UPDATE	2:09:42	Flight	2785	SeatsRem	1	0
23	3	COMMIT	2:09:43					
24	4	INSERT	2:09:44	Resv	1098	*		<515,99,...>
25	4	UPDATE	2:09:45	Flight	3843	SeatsRem	15	14

17. Identify the concurrency control problem depicted in the following timeline. Identify the least restrictive isolation level that eliminates the problem. Note that SERIALIZABLE is the most restrictive isolation level. Redraw the timeline showing the locks imposed by the least restrictive isolation level that eliminates the problem.

Transaction A	Time	Transaction B
	T_1	UPDATE $QOH_2 = QOH_2 - 5$ (20)
Read QOH_2 (20)	T_2	
Sum = Sum + QOH_2	T_3	
Read QOH_1 (15)	T_4	
Sum = Sum + QOH_1	T_5	
	T_6	UPDATE $QOH_1 = QOH_1 - 5$ (13)
	T_7	Commit

18. Identify the concurrency control problem depicted in the following timeline. Identify the least restrictive isolation level that eliminates the problem. Note that SERIALIZABLE is the most restrictive isolation level. Redraw the timeline showing the locks imposed by the least restrictive isolation level that eliminates the problem.

Transaction A	Time	Transaction B
	T_1	Read QOH_1 (55)
	T_2	$QOH_1 = QOH_1 - 10$
	T_3	Write QOH_1 (45)
Read QOH_1 (45)	T_4	
Read QOH_2 (15)	T_5	
	T_6	Read QOH_2 (15)
	T_7	$QOH_2 = QOH_2 - 5$
	T_8	Write QOH_2 (10)
	T_9	Rollback

19. Identify the concurrency control problem depicted in the following timeline. Identify the least restrictive isolation level that eliminates the problem. Note that SERIALIZABLE is the most restrictive isolation level. Redraw the timeline showing the locks imposed by the least restrictive isolation level.

Transaction A	Time	Transaction B
	T_1	Read QOH_1 (10)
	T_2	If $QOH_1 > 10$ then $QOH_1 = QOH_1 + 30$
Read QOH_1 (10)	T_3	
$QOH_1 = QOH_1 - 3$	T_4	
	T_5	Write QOH_1 (40)
	T_6	Commit
Write QOH_1 (7)	T_7	

20. Identify the concurrency control problem depicted in the following timeline. Identify the least restrictive isolation level that eliminates the problem. Note that SERIALIZABLE is the most restrictive isolation level. Redraw the timeline showing the locks imposed by the least restrictive isolation level that eliminates the problem.

Transaction A	Time	Transaction B
Read QOH (10)	T_1	
QOH = QOH + 30	T_2	
	T_3	Read QOH (10)
	T_4	QOH = QOH – 10
Write SR (40)	T_5	
	T_6	Write SR (0)
Commit	T_7	
	T_8	Commit

21. Try to create a deadlock among more than two airline reservation transactions. You should assume that locks are granted in the flight time order for a reservation. Thus, a lock on a departing flight is granted before a lock on a return flight for a given transaction. You can add complications such as reservations with multiple legs per departure or return. With multiple legs, a lock on the first leg should be granted before a lock on the second leg, however.
22. Use the extended 2PL protocol (locking granularity with intent locks) to list the locks obtained by transaction Tr1 in the following scenario. Transaction Tr1 reads row R1 (located on block B150 in table T1), followed by writing row R2 (located on block B100 in table T2), followed by reading row R3 (located on block B210 in table T1), and then writes row R4 (located on block B190 in table T2). The tables are part of database DB1. You should also list the order of obtaining the locks.
23. Using the scenario of problem 22, list the order of releasing the locks at end of transaction following the extended 2PL protocol.
24. Use the extended 2PL protocol (locking granularity with intent locks) to list the locks obtained by transaction T1 in the following scenario. Transaction T1 first reads every row of block B95 of table T1, followed by writing every row of block B135 of table T2, followed by reading every row of block B201 of table T2, and finally writes rows R4 and R5 (located in block B201 of table T2). The tables are part of database DB1. You should also list the order of obtaining the locks.
25. Using the scenario of problem 24, list the order of releasing the locks at end of transaction following the extended 2PL protocol.
26. Identify the results (grant or wait) for the following lock requests given the sample lock table below (Table 17-P1). Lock requests: IX lock on database DB1, IS lock on database DB1, IX lock on table T1, IS lock on Table T1, SIX lock on block B100 of table T1, S lock on row R1 of B100, X lock on row R2 of B100.

TABLE 17-P1
Sample Lock Table

TransNo	ObjId	ObjectType	Parent	LockType	Count
101001	DB1	Database	-	IX	30
101001	DB1	Database	-	IS	15
101001	T1	Table	DB1	IX	10
101001	T2	Table	DB1	IS	8
101001	B100	Block	T1	IX	5
101001	B101	Block	T2	IS	5
101001	R1	Row	B100	S	3
101001	R2	Row	B100	S	5
101001	R3	Row	B101	X	1
101001	R4	Row	B101	X	1

27. Investigate features of in-memory transaction processing in an enterprise DBMS. What product version or edition supports in-memory transaction processing? Summarize the vendor's claims about benefits, reasons for improved performance, and trade-offs for increased resource levels.

REFERENCES FOR FURTHER STUDY

This chapter, although providing a broad coverage of transaction management, has only covered the basics. Transaction management is a detailed subject for which entire books have been written. Specialized books on transaction management include Bernstein and Newcomer (1997) and Gray and Reuter (1993). Harizopulous et al. (2008) analyze shortcomings of traditional concurrency control with two phase locking. Tu et al. (2013) describe in-memory transaction processing using optimistic concurrency control. Shasha and Bonnet (2003) provide more details about transaction design and recovery tuning. Peinl, Reuter, and Sammer (1988) provide a stock-trading case study on transaction design that elaborates on the ideas presented in Section 17.4. For details about transaction processing performance, consult the website of the Transaction Processing Council (www.tpc.org).

chapter

18

Client-Server Processing, Parallel Database Processing, and Distributed Databases

Learning Objectives

This chapter describes ways that database management systems utilize computer networks and distributed computing resources to support client-server processing, parallel database processing, and distributed databases. After this chapter, the student should have acquired the following knowledge and skills:

- List reasons for client-server processing, parallel database processing, and distributed data
- Describe basic tiered architectures for client-server database processing
- Describe specialized client-server architectures for web services, cloud computing, and extreme transaction processing
- Describe common architectures for parallel database processing and big data processing
- Describe differences between technology for tightly integrated and loosely integrated distributed databases
- Compare different kinds of distributed database transparency
- Understand the nature of query processing and transaction processing for distributed databases

OVERVIEW

Chapters 12 to 15 and 17 described database processing for business intelligence and transactions. As explained in these chapters, transaction and business intelligence processing are vital to modern organizations. In this chapter, you will learn about usage of computer networks and distributed computing resources (processors, memory, communication networks, and data storage) to improve reliability and performance for both kinds of processing.

This chapter explains utilization of computer networks and distributed computing resources by DBMSs. Before understanding the details, you should understand

the motivation for utilizing these resources. This chapter discusses business reasons for client-server processing, parallel database processing, and distributed data along with deployment of these approaches in cloud computing environments. After grasping the motivation, you are ready to learn basic and specialized architectures for client-server database processing and conceptual architectures for parallel database processing. Description of parallel database processing in Oracle and IBM DB2 along with big data approaches in open source projects complement the conceptual presentation. Distributing data in addition to distributing processing allows more flexibility but also involves more complexity. To depict the trade-off between flexibility and complexity, this chapter explains distributed database architectures, levels of transparency for distributed data, and distributed database processing for queries and transactions. Examples of transparency for Oracle distributed databases complement the conceptual material.

18.1 OVERVIEW OF DISTRIBUTED PROCESSING AND DISTRIBUTED DATA

To facilitate learning about conceptual issues, this section separates distributed processing from distributed data. Both areas have distinct architectures, design problems, and processing technologies. After learning them separately, you can understand ways to combine them. This section begins your study by discussing motivations behind two different kinds of distributed processing (client-server and parallel database processing) and distributed databases as well as deploying distributed processing and databases in cloud environments.

18.1.1 Motivation for Client-Server Processing

The client-server approach supports usage of remote computing resources for performing complex business processes consisting of a variety of subtasks. For example, electronic shopping is a complex process consisting of product selection, ordering, inventory management, payment processing, shipping, and product returns. A client is a program that makes requests to a server. A server performs requests and communicates results to clients. Computing architectures arrange clients and servers across networked computers to divide complex work into more manageable units. The simplest arrangement divides work between clients processing on personal computers and a server processing on a separate computer as shown in Figure 18.1. Section 18.2 presents more powerful architectures for client-server processing along with typical divisions of work among computers.

Distributed processing with the client-server approach offers a number of advantages related to flexibility, scalability, and interoperability. Flexibility refers to the ease of maintaining and adapting a system. Maintenance costs often dominate the cost of initially developing an information system because of long life and system revisions. The client-server approach promotes flexibility because volatile sections of code can be isolated from more stable sections. For example, a developer can separate user interface code from code for business rules and data access. If deploying a new interface, other parts of the code remain unchanged. In addition, the client-server approach is ideally suited for object-oriented programming to support reusability.

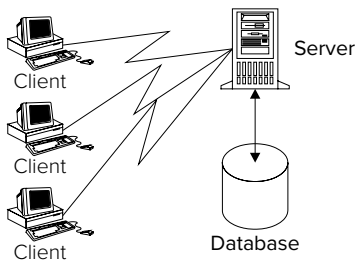


FIGURE 18.1
Simple Client-Server Architecture for Distributed Processing

The client-server approach supports scalable growth of hardware and software capacity. Scalability refers to the ability to add and remove capacity in small units. Vertical scalability refers to the ability to add capacity on the server side. For example, work from an overloaded server may be moved to a new server to alleviate a bottleneck or handle new demand from additional workstations. The new server can have just the level of additional computing capacity necessary. Horizontal scalability refers to the ability to add capacity on the client side through additional workstations and movement of work between clients and servers. For example, work can be moved from clients to a server to allow the use of inexpensive client hardware (thin clients). Work also can move in the opposite direction (from server to client) to alleviate server processing loads and take advantage of client computing capabilities.

Scalable growth also can lead to improved performance. For example, adding middleware can reduce contention problems caused by many users accessing a database. The next section describes middleware that can efficiently manage many simultaneous users accessing a database. In addition, specialized servers can be employed to handle work that would otherwise slow all users. For example, multimedia servers can handle requests for images, thus freeing other servers from this time-consuming task.

Client-server systems based on open standards support interoperability. Interoperability refers to the ability of two or more systems to exchange and use software and data. Open standards promote a marketplace of suppliers, leading to lower costs and higher quality. Software components in the marketplace are interoperable if they conform to the standards. The most heavily standardized areas are the Internet and Web, where client-server databases are becoming increasingly important.

Despite the advantages of client-server processing, some significant pitfalls may occur. Developing client-server software may be more complex because of architectural choices. A client-server architecture specifies an arrangement of components and a division of processing among the components. Section 18.2 presents several possible architectures for client-server database processing. The choice of an inappropriate architecture can lead to poor performance and maintenance problems. In addition to architectural issues, the designer may face a difficult decision about building a client-server database on proprietary methods versus open standards. Proprietary methods allow easier resolution of problems because one vendor is responsible for all problems. Proprietary methods may also have better performance because they are not as general as open standards. In the long term, proprietary methods can be expensive and inflexible, however. If a vendor does not grow with the industry, a client-server database may become outdated and expensive to upgrade.

18.1.2 Motivation for Parallel Database Processing

In contrast to the usage of client-server processing to distribute complex work among networked computers, parallel database processing divides large tasks into many smaller tasks and distributes the smaller tasks among interconnected computers. For example, parallel database processing may be used to perform a join operation on large tables. The usage of RAID architectures described in Chapter 8 is a simple form of parallel database processing. Section 18.3 presents more powerful architectures for parallel database processing.

Parallel database processing can improve performance through scaleup and speedup. Scaleup involves additional work accomplished by increasing computing capacity while holding completion time constant. Additional work means the increased size of a job such as additional transactions completed. For ideal linear scaleup, increasing computing capacity n times allows completion of n times the amount of work in the same time. Due to coordination overhead, linear scaleup is not possible in most situations. Scaleup is the ratio of the amount of work completed with a larger configuration to the amount of work completed with an original configuration. For example, scaleup increases to 1.75 for an original configuration processing 100 transactions per minute and to a revised configuration with doubled computing capacity processing 175 transactions per minute.

Speedup involves decrease in time to complete a task rather than amount of work performed. With added computing capacity, speedup measures time reduction while holding the amount of work constant. For example, organizations typically need to complete daily refresh processing in a timely manner to ensure availability for the next business day. Organizations need to determine the amount of additional computing capacity that ensures completion of the work within an allowable time. Speedup is the ratio of completion time with an original configuration to the completion time with additional capacity. For example, speedup increases to 1.5 if doubling capacity decreases refresh processing time from 6 hours to 4 hours.

Availability is the accessibility of a system often measured as the system's uptime. For highly available or fault-resilient computing, a system experiences little downtime and recovers quickly from failures. Fault-tolerant computing takes resiliency to the limit in that processing must be continuous without cessation. The cost of downtime determines the degree of availability desired. Downtime cost can include loss of sales, lost labor, and idle equipment. For a large organization, the cost of downtime can be hundreds of thousands of dollars per hour. Parallel database processing can increase availability because a DBMS can dynamically adjust to the level of available resources. Failure of an individual computer will not stop processing on other available computers.

The major drawback to parallel database processing is cost. Parallel database processing involves high costs for DBMS software and specialized coordination software. There are possible interoperability problems because coordination is required among DBMS software, operating system, and storage systems. DBMS vendors provide powerful tools to deploy and manage the high complexity of parallel database processing. Performance improvements if not predictable would be a significant drawback. Predictable performance improvements enable organizations to plan for additional capacity and dynamically adjust capacity depending on anticipated processing volumes and response time constraints.

18.1.3 Motivation for Distributed Data

Distributed data offer a number of advantages related to data control, communication costs, and performance. Distributing a database allows the location of data to match an organization's structure. For example, parts of a customer table can be located close to customer processing centers. Decisions about sharing and maintaining data can be set locally to provide control closer to the data usage. Often, local employees and management understand issues related to data better than management at remote locations.

Distributed data can lead to lower communication costs and improved performance. Data should be located so that 80 percent of the requests are local. Local requests incur little or no communication costs and delays compared to remote requests. Increased data availability also can lead to improved performance. Data are more available because there is no single computer responsible for controlling access. In addition, data can be replicated so that they are available at more than one site.

Despite the advantages of distributed data, some significant pitfalls may occur. Distributed database design issues are very difficult. A poor design can lead to higher communication costs and poor performance. Distributed database design is difficult because of the lack of tools, the number of choices, and the relationships among choices. Distributed transaction processing can add considerable overhead, especially for replicated data. Distributed data involve more security concerns because many sites can manage data. Each site must be properly protected from unauthorized access.

18.1.4 Motivation for Cloud Based Computing

The architectures presented in this section assume a traditional product licensing and hosting approach. Cloud computing provides a new approach without initial product

licensing costs and hosting requirements. Using web-based interfaces, organizations can design and deploy databases with dynamic resource allocation provided by the cloud. Cloud computing can lower costs through economies of scale and specialization achievable by deployments for large numbers of organizations. Some organizations may be reluctant to relinquish control over vital operations to a cloud vendor. A cloud may restrict flexibility forcing an organization to use a standard environment. Performance may suffer if a cloud cannot be tuned to match an organization's processing needs.

Internally, the cloud can use any distributed processing approach although the internal details of the cloud are invisible to the cloud user. Figure 18.2 depicts a cloud service using an internal client-server approach with replicated and distributed data. Cloud computing environments can provide varying levels of support for database development and operations including hosting the DBMS and computing infrastructure, providing platforms for development of database applications, and supporting databases for operations and business intelligence.

18.1.5 Summary of Advantages and Disadvantages

Before moving forward, you should review the relative merits of client-server processing, parallel database processing, distributed data, and cloud-based computing as listed in Table 18-1. To gain maximum leverage, the technologies can be combined.

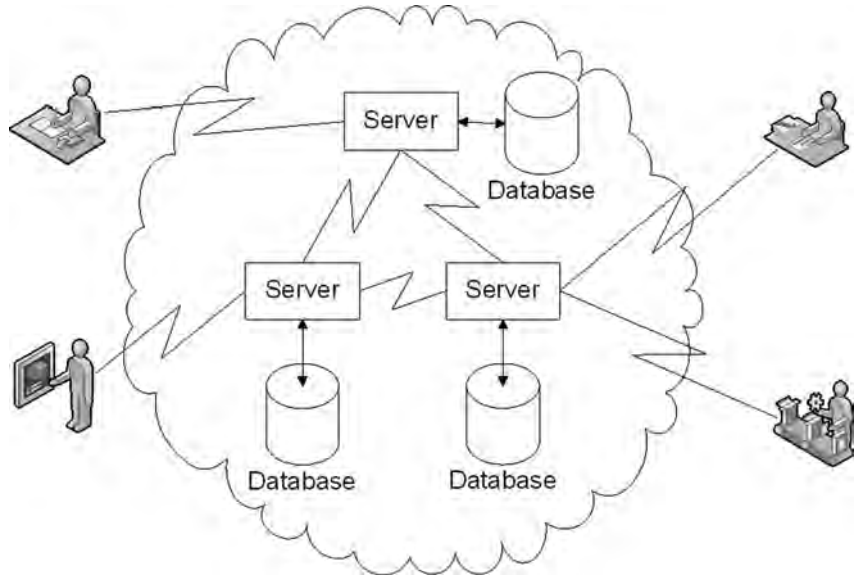


FIGURE 18.2
Cloud-Based Database Architecture

Technology	Advantages	Disadvantages
Client-server processing	Flexibility, interoperability, scalability	High complexity, high development cost, possible interoperability problems
Parallel database processing	Speedup, scaleup, availability, scalability for predictive performance improvements	Possible interoperability problems, high cost
Distributed databases	Local control of data, improved performance, reduced communication costs, increased reliability	High complexity, additional security concerns
Cloud computing	No initial licensing costs, no hosting, dynamic scalability, high availability, economies of scale, specialization	Possible performance reductions and loss of control and flexibility through reliance on external organizations

TABLE 18-1
Summary of Distributed Processing and Data

Distributed processing with the client-server approach is the more mature technology although parallel database processing has gained rapid acceptance and maturity. As distributed database technology matures and gains acceptance, organizations will deploy all three technologies. The impact of cloud computing on the DBMS market has become substantial with major enterprise DBMS vendors offering mature products. Cloud DBMS services provided by Amazon and Microsoft are market leaders but other vendors including Oracle, Google, and IBM are aggressively marketing cloud DBMS products.

18.2 CLIENT-SERVER DATABASE ARCHITECTURES

Client-Server Architecture an arrangement of components (clients and servers) among computers connected by a network. A client-server architecture supports efficient processing of messages (requests for service) between clients and servers.

The design of a client-server database affects the advantages and the disadvantages cited in the previous section. A good design tends to magnify advantages and reduce disadvantages relative to an organization's requirements. A poor design may exacerbate disadvantages and diminish advantages. Proper design of a client-server database may make the difference between success and failure of an information system project. To help you achieve good designs, this section discusses design issues of client-server databases and describes how these issues are addressed in various architectures.

18.2.1 Design Issues

Two design issues, division of processing and process management, affect the design of a client-server database. Division of processing refers to the allocation of tasks to clients and servers. Process management involves interoperability among clients and servers and efficient processing of messages between clients and servers. Software for process management is known as "middleware" because of its mediating role. This section describes these issues so that you will understand how various architectures address them in the next section.

Division of Processing In a typical client-server database, there are a number of tasks that can be performed locally on a client or remotely on a server. The following list briefly describes these tasks.

- *Presentation:* code to maintain the graphical user interface. The presentation code displays objects, monitors events, and responds to events. Events include user-initiated actions with the mouse and the keyboard as well as external events initiated by timers and other users.
- *Validation:* code to ensure the consistency of the database and user inputs. Validation logic often is expressed as integrity rules that are stored in a database.
- *Business logic:* code to perform business functions such as payroll calculations, eligibility requirements, and interest calculations. Business logic may change as the regulatory and the competitive environments change.
- *Workflow:* code to ensure completion of business processes. Workflow code may route forms, send messages about a deadline, and notify users when a business process is completed.
- *Data access:* code to extract data to answer queries and modify a database. Data access code consists of SQL statements and translation code that is usually part of the DBMS. If multiple databases are involved, some translation code may reside in software separate from a DBMS.

Parts of these tasks can be divided between clients and servers. For example, some validation can be performed on a PC client and some can be performed on a database server. Thus, there is a lot of flexibility about the division of processing tasks. Section 18.2.2 describes several popular ways to divide processing tasks.

Middleware Interoperability is an important function of middleware. Clients and servers can exist on platforms with different hardware, operating systems, DBMSs, and programming languages. Figure 18.3 depicts middleware allowing clients and servers to communicate without regard to the underlying platforms of the clients and the servers. The middleware enables a client and a server to communicate without knowledge of each other's platform.

Middleware
a software component in a client-server architecture that performs process management. Middleware allows servers to efficiently process messages from a large number of clients. In addition, middleware can allow clients and servers to communicate across heterogeneous platforms. To handle large processing loads, middleware often is located on a dedicated computer.

Efficient message control is another important function of middleware. In a typical client-server environment, there are many clients communicating with a few servers. A server can become overloaded just managing the messages that it receives rather than completing the requests. Middleware allows servers to concentrate on completing requests rather than managing requests. Middleware can perform queuing, scheduling, and routing of messages allowing clients and servers to perform work at different speeds and times.

Based on the functions of interoperability and message control, several kinds of middleware are commercially available, as described in the following list:

- *Transaction-processing monitors* are the oldest kind of middleware. Traditionally, transaction-processing monitors relieve the operating system of managing database processes. A transaction-processing monitor can switch control among processes much faster than an operating system. In this role, a transaction-processing monitor receives transactions, schedules them, and manages them to completion. In the last decade, transaction-processing monitors have taken additional tasks such as updating multiple databases in a single transaction.
- *Message-oriented middleware* maintains a queue of messages. A client process can place a message on a queue and a server process can remove a message from a queue. Message-oriented middleware differs from transaction processing monitors primarily in the intelligence of the messages. Transaction-processing monitors provide built-in intelligence but use simple messages. In contrast, message-oriented middleware provides much less built-in intelligence but supports more complex messages.
- *Object-request brokers* provide a high level of interoperability and message intelligence. To use an object-request broker, messages must be encoded in a standard interface description language. An object-request broker resolves platform differences between a client and a server. In addition, a client can communicate with a server without knowing the location of the server.
- *Data access middleware* provide a uniform interface to relational and nonrelational data using SQL. Requests to access data from a DBMS are sent to a data access driver rather than directly to the DBMS. The data access driver converts the SQL statement into the SQL supported by the DBMS and then routes the request to the DBMS. The data access driver adds another layer of overhead between an application and a DBMS. However, the data access driver supports independence between an application and the proprietary SQL supported by a DBMS vendor. The two leading data access middleware are the Open Database Connectivity (ODBC) supported by Microsoft and the Java Database Connectivity (JDBC) supported by Oracle.

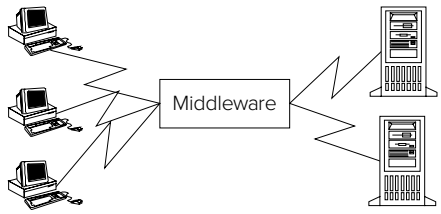


FIGURE 18.3
Client-Server Computing with Middleware

18.2.2 Basic Architectures

The basic client-server database architectures provide clear choices for the design issues. For each architecture, this section describes typical division of processing, message management approaches, and trade-offs among architectures. The basic architectures are the building blocks of specialized architectures described in the following section.

Two-Tier Architecture
a client-server architecture in which a PC client and a database server interact directly to request and transfer data. The PC client contains the user interface code, the server contains the data access logic, and the PC client and the server share the validation and business logic.

Two-Tier Architecture The two-tier architecture features a PC client and a database server as shown in Figure 18.4. The PC client contains the presentation code and SQL statements for data access. The database server processes the SQL statements and sends query results back to the PC client. In addition, the database server performs process management functions. The validation and business logic code can be split between the PC client and the database server. The PC client can invoke stored procedures on the database server for business logic and validation. Typically, much of the business logic code resides on the client. PC clients in a two-tier architecture are sometimes called "fat clients" because of the large amount of business logic in the client.

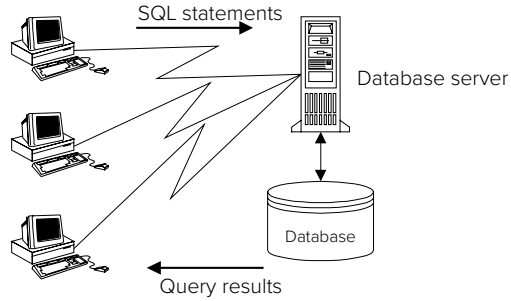
The two-tier architecture is a good approach for systems with stable requirements and a moderate number of clients. On the positive side, the two-tier architecture is the simplest to implement due to the number of good commercial development environments. On the negative side, software maintenance can be difficult because PC clients contain a mix of presentation, validation, and business logic code. To make a significant change in business logic, code must be modified on many PC clients. In addition, utilizing new technology may be difficult because two-tier architectures often rely on proprietary software rather than open standards. To lessen reliance on a particular database server, the PC client can connect to intermediate database drivers such as the Open Database Connectivity (ODBC) drivers instead of directly to a database server. The intermediate database drivers then communicate with the database server.

Performance can be poor when a large number of clients submit requests because the database server may be overwhelmed with managing messages. Several sources report that two-tier architectures are limited to about 100 simultaneous clients. With a larger number of simultaneous clients, a three-tier architecture may be necessary. In addition, connecting to intermediate drivers rather than directly to a database server can slow performance.

Three-Tier Architecture
A client-server architecture with three layers: a PC client, a backend database server, and either a middleware or an application server.

Three-Tier Architecture To improve performance, the three-tier architecture adds another server layer as depicted in Figure 18.5. One way to improve performance is to add a middleware server (Figure 18.5(a)) to handle process management. The middleware usually consists of a transaction-processing monitor or message-oriented middleware. A transaction-processing monitor may support more simultaneous connections than message-oriented middleware. However, message-oriented middleware provides more flexibility in the kinds of messages supported. A second way to improve performance is to add an application server for specific kinds of processing such as report writing. In either approach, the additional server software can reside on a separate computer, as depicted in Figure 18.5. Alternatively, the additional server software can be distributed between the database server and PC clients.

FIGURE 18.4
Two-Tier Architecture



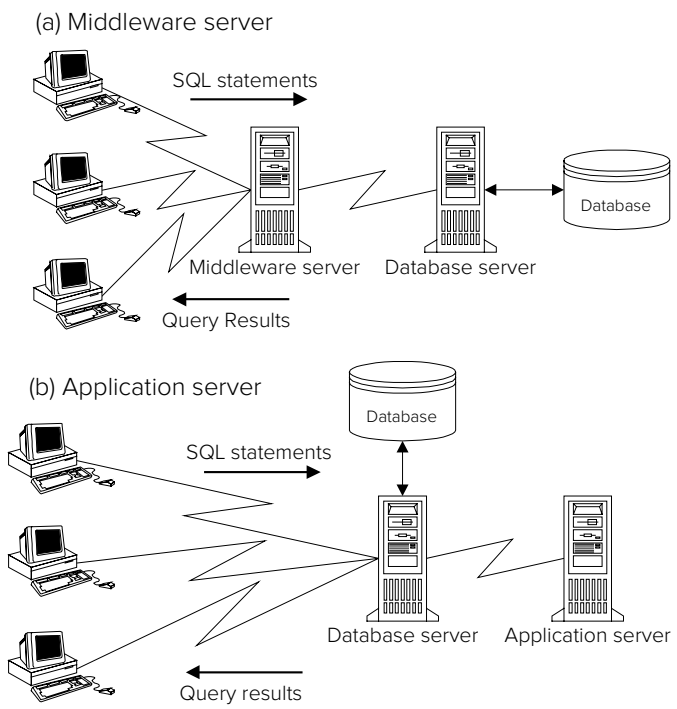


FIGURE 18.5
Three-Tier Architecture

Although the three-tier architecture addresses performance limitations of the two-tier architecture, it does not address division-of-processing concerns. The PC clients and the database server still contain the same division of code although the tasks of the database server are reduced. Multiple-tier architectures provide more flexibility on division of processing.

Multiple-Tier Architecture To improve performance and provide flexible division of processing, multiple-tier architectures support additional layers of servers, as depicted in Figure 18.6. The application servers can be invoked from PC clients, middleware, and database servers. The additional server layers provide a finer division of processing than a two- or a three-tier architecture. In addition, the additional server layers can also improve performance because both middleware and application servers can be deployed.

Multiple-Tier Architecture
a client-server architecture with more than three layers: a PC client, a backend database server, an intervening middleware server, and application servers. The application servers perform business logic and manage specialized kinds of data such as images.

Multiple-tier architectures for electronic commerce involve a Web server to process requests from Web browsers. The browser and the server work together to send and receive Web pages written in the Hypertext Markup Language (HTML). A browser displays pages by interpreting HTML code in the file sent by a server.

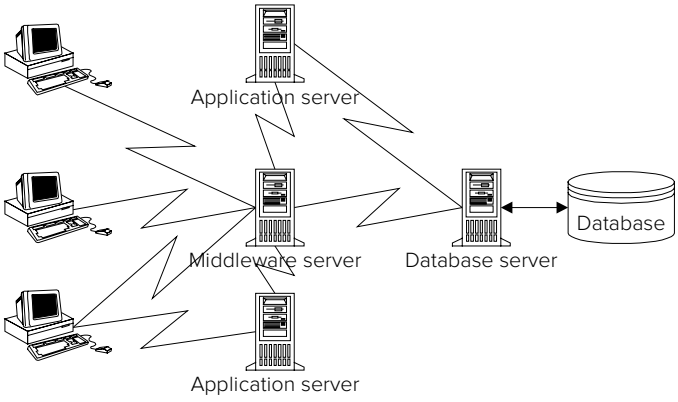


FIGURE 18.6
Multiple-Tier Architecture

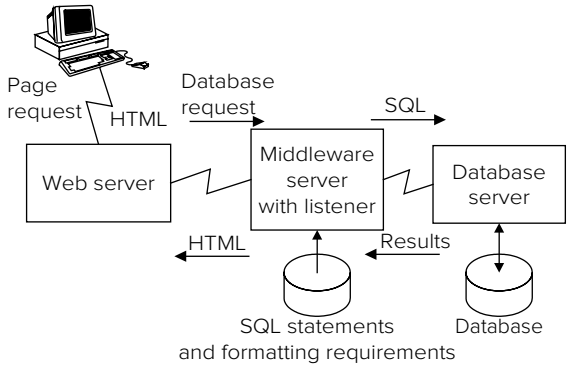


FIGURE 18.7
Web Server interacting with Middleware Server and Database Server

The Web server can interact with a middleware and database server as depicted in Figure 18.7. Requests for database access are sent to a middleware server and then routed to a database server. Application servers can be added to provide additional levels of client-server processing.

18.2.3 Specialized Architectures

The specialized client-server database architectures have been developed to meet specific market needs. The specialized architectures extend the basic architectures with new server layers, message interfaces, transparency, and processing features.

Web Services Architecture Web services extend multiple-tier architectures for electronic business commerce using Internet standards to achieve high interoperability. Electronic business commerce involves services provided by automated agents among organizations. Web services allow organizations to reduce the cost of electronic business by deploying services faster, communicating new services in standard formats, and finding existing services from other organizations. Web services operate in the Internet, a network of networks built with standard languages and protocols for high interoperability. Web services use general Internet standards and new standards for electronic business commerce.

Web Services Architecture
an architecture that supports electronic commerce among organizations. A set of related Internet standards supports high interoperability among service requestors, service providers, and service registries. The most important standard is the Web Service Description Language used by service requestors, service providers, and service registries.

The **Web Services Architecture** supports interaction between a service provider, service requestor, and service registry as depicted in Figure 18.8. The service provider owns the service and provides the computing platform offering the service. The service requestor application searches for a service and uses the service after it is discovered. The service registry is the repository where the service provider publishes its service description and the service requestor searches for available services. After a service requestor finds a service, the service requestor uses the service description to bind with the service provider and invoke the service implementation maintained by the service provider.

To support interoperability, the Web Services Architecture uses a collection of interrelated Internet standards. Table 18-2 summarizes the core web service standards. XML (eXtensible Markup Language) is the underlying foundation for most of the standards. XML is a meta language that supports the specification of other languages. In the Web Services Architecture, the WSFL, UDDI, WSDL, and SOAP standards are XML-compliant languages. For a service requestor and provider, the WSDL is the standard directly used to request and bind a Web service. A WSDL document provides an interface to a service enabling the service provider to hide the details of providing the service. The core standards are augmented by a wide array of additional standards for security, reliable messaging, transactions, meta data, and messaging.

To fulfill a Web service, a service provider may utilize client-server processing, parallel database processing, and distributed databases. The details of the processing are hidden from the service requestor. The Web Services Architecture provides

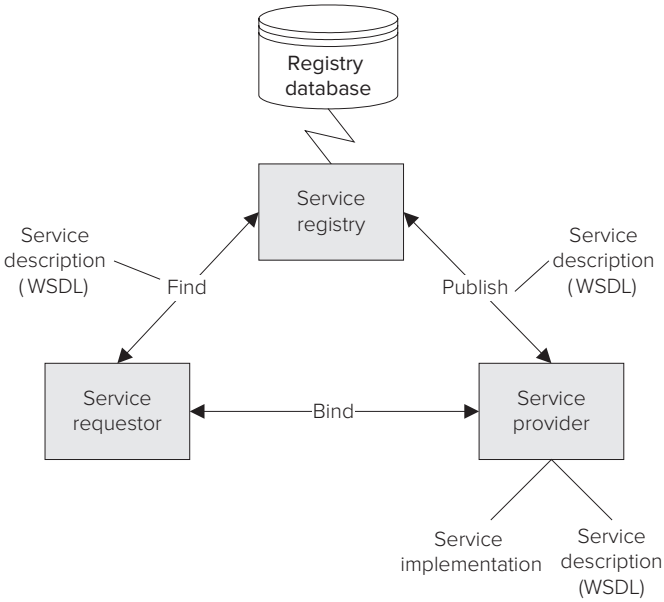


FIGURE 18.8
Web Services Architecture

Standard	Usage
Web Services Flow Language (WSFL)	Specification of workflow rules for services
Universal Description, Discovery Integration (UDDI)	Specification of a directory of Web services including terminology and usage restrictions
Web Services Description Language (WSDL)	Specification of Web services
Simple Object Access Protocol (SOAP)	Sending and receiving XML messages
HTTP, FTP, TCP-IP	Network and connections

TABLE 18-2
Summary of Core Standards for Web Services

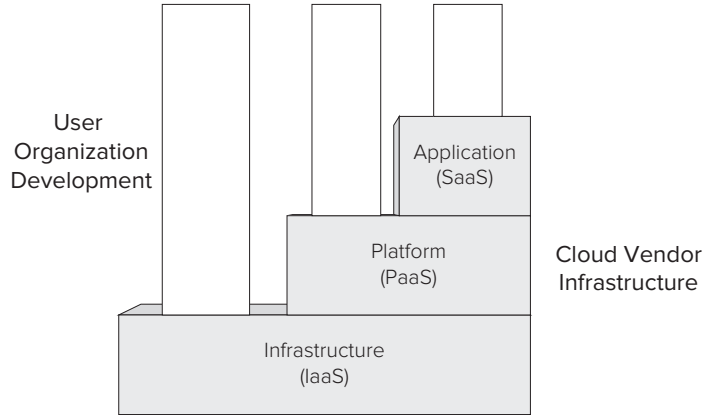
another layer of middleware to publish, find, and execute services among electronic businesses.

Cloud Computing Architectures Cloud computing extends other client-server architectures with an emphasis on transparency. According to the U.S. National Institute of Standards and Technology office, "Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." This definition supports a variety of service and deployment models that provide database services in a cloud.

Cloud service models provide varying levels of infrastructure, platform, and software support. As depicted in Figure 18.9, cloud service models vary by the services provided by the cloud vendor and organizations using the cloud. In the Infrastructure as a Service (IaaS) model, the cloud vendor provides basic hardware and software infrastructure possibly including servers, operating system, storage capacity, network capacity, website hosting, and database software. Organizations use their own development environments to build customized applications that operate in the infrastructure cloud. In the Platform as a Service (PaaS) model, the cloud vendor provides standard development environments deployed on the infrastructure cloud. Organizations use the standard development environments to build customized applications that operate in the infrastructure cloud. In the Software as a Service (SaaS) model, the cloud vendor provides standardized services that operate in the cloud. Organizations do not need to host or monitor software operations. In all three service models, the

Cloud Service Model provides varying levels of infrastructure, platform, and software support. The cloud service models vary by the services provided by the cloud vendor and organization using the cloud: Infrastructure as a Service (IaaS) with the vendor providing infrastructure support, Platform as a Service (PaaS) with the vendor providing infrastructure and development platforms, and Software as a Service (SaaS) with the vendor providing complete service solutions.

FIGURE 18.9
Cloud Service Models



cloud vendor provides logical control so that organizations can configure the cloud without the responsibility to host and physically manage the cloud.

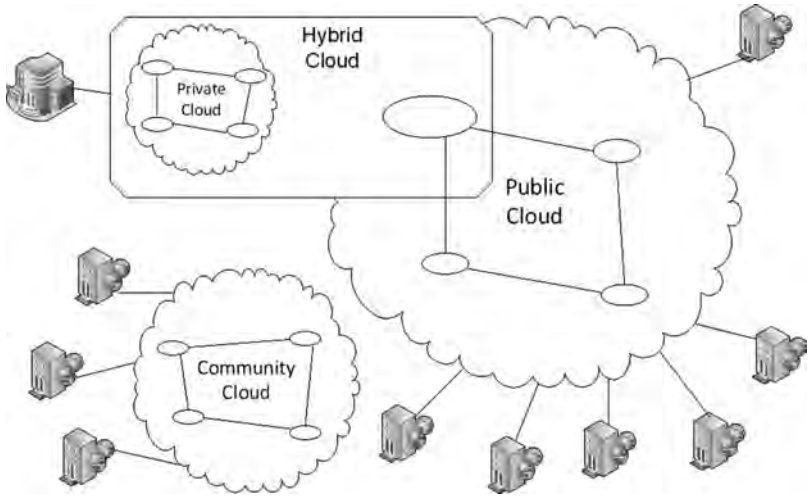
Cloud Deployment Model indicates variations of cloud availability and control. Cloud deployment can be public (open to any organization), community (open to cooperating organizations), private (open to a single organization), or hybrid (combination of a public or community cloud and private cloud). Typically, broader cloud availability increases economies of scale possibly at the cost of reduced performance and more restrictions on platforms and infrastructure.

A **cloud deployment model** indicates variations of cloud availability and control as demonstrated in Figure 18.10. A public cloud involves an external organization providing resource sharing to an open number of organizations. In contrast, a private cloud involves resource sharing controlled by a single organization often through a private network. Private clouds may provide improved performance at a higher cost than public clouds because private clouds may lack the economies of scale of public clouds. A community cloud involves resource sharing among cooperating organizations. A community cloud increases economies of scale of a private cloud while providing opportunities for improved performance as compared to a public cloud. A hybrid cloud combines a private cloud with aspects of a public or community cloud. Many organizations utilize hybrid clouds to balance economies of scale with performance.

Infrastructure cloud services include specialized database products and relational DBMSs under specialized licensing terms. Amazon Web Services (AWS) offers several DBMSs to support a variety of requirements. AWS provides two NoSQL DBMSs (DynamoDB and SimpleDBO), an open source relational DBMS (Amazon Aurora), a data warehouse DBMS (Amazon Redshift), and enterprise DBMSs (Amazon RDS hosting prominent enterprise DBMSs). To reduce product management complexity and decrease ownership costs, AWS automates common administrative tasks such as backup and software upgrades and provides flexible resource scalability.

Enterprise DBMS vendors also provide cloud DBMS services. For example, Microsoft Azure SQL Database is a cloud-based relational database platform built on SQL

FIGURE 18.10
Cloud Deployment Models



Server technologies. Azure SQL Database provides a subset of SQL Server features with limitations such as lack of distributed transactions, limited transaction isolation levels, and connection restrictions. For SQL Server users, SQL Azure Database provides an alternative to the Amazon RDS for organizations that prefer a Microsoft oriented cloud (Windows Azure).

Middleware for Extreme Transaction Processing Extreme transaction processing (XTP) is a client-server model to support applications with highly demanding requirements for performance, scalability, availability, security, manageability and dependability. The financial trading industry exemplifies XTP requirements with processing volumes of tens of thousands of transactions per second in which millisecond delays can be costly. Other industries embracing XTP include utilities for energy trading, digital media firms for order processing, and telecommunications firms for call connections and switching.

Extreme Transaction Processing (XTP)
a client-server model to support applications characterized by notably demanding requirements for performance, scalability, availability, security, manageability, and dependability.

To meet the demands of extreme transaction processing, XTP middleware has been developed. XTP middleware reduces the delay of disk-based updating using a memory cache distributed among a cluster of servers. XTP middleware extends the concept of in-memory transaction processing (see Chapter 17) with a grid computing architecture to improve scalability and reliability. XTP middleware and in-memory transaction processing are competing technologies with more recent product development on in-memory transaction processing.

The key feature of XTP middleware is the **write-behind cache** that batches updates to a database server within a specified time interval. The XTP middleware performs updates in the cache and tracks the list of dirty records, periodically performing database write operations on the set of dirty records. As an additional performance improvement, the XTP middleware performs conflation on the dirty records. Conflation means if the same record is updated multiple times within the buffering period then the cache only keeps the last update. Conflation can significantly improve performance in applications with rapidly changing values such as stock prices in financial trading applications.

Write-Behind Cache
a key feature of XTP middleware supporting batch updates to a database server within a specified time interval. The XTP middleware performs updates in the cache and tracks the list of dirty records, periodically performing database write operations on the set of dirty records.

XTP middleware incurs some overhead to provide reliable transaction processing. The cache at each server is replicated so that committed transactions survive failures. In addition to replication overhead, the write-behind cache has a time delay between database changes and the cache being updated (or invalidated) to reflect them. If all data access occurs through the cache, the cache will always have the correct latest value. Thus, a common constraint on XTP middleware is that all transaction activity must use the cache.

XTP middleware, using a write-behind cache, provides the promise of linear scalability. As transaction volumes increase, servers with cache are added to maintain near constant processing times. XTP middleware with promises of linear scalability are offered by both established DBMS vendors and emerging firms. Some of the major XTP middleware products are Oracle Coherence, IBM WebSphere eXtreme Scale, and Atomikos ExtremeTransactions.

18.3 PARALLEL DATABASE PROCESSING

In the last two decades, parallel database technology has gained commercial acceptance for large organizations. Most enterprise DBMS vendors and some open source DBMSs support parallel database technology to meet market demand. Organizations utilize these products to realize benefits of parallel database technology (scaleup, speedup, and availability) while managing possible interoperability problems. This section describes parallel database architectures to provide a framework to understand commercial offerings by enterprise DBMS vendors. The last part of this section presents parallel processing architectures for big data applications, developed in open source projects. Enterprise DBMSs vendors are now integrating parallel processing architectures for big data applications.

Parallel DBMS
a DBMS capable of utilizing tightly-coupled computing resources (processors, disks, and memory). Tight coupling is achieved by networks with data exchange time comparable to the time of the data exchange with a disk. Parallel database technology promises performance improvements and high availability although interoperability problems may occur if not properly managed.

Architectures for Parallel Database Processing
the clustered disk (CD) and clustered nothing architectures dominate in commercial DBMSs. A cluster is a tight coupling of two or more computers that behave as a single computer. In the CD architecture, the processors in each cluster share all disks, but nothing is shared across clusters. In the CN architecture, the processors in each cluster share no resources, but each cluster can be manipulated to work in parallel to perform a task.

18.3.1 Architectures and Design Issues

A **parallel DBMS** uses a collection of resources (processors, disks, and memory) to perform work in parallel. Given a fixed resource budget, work is divided among resources to achieve desired levels of performance (scaleup and speedup) and availability. A parallel DBMS uses a high-speed network, operating system, and storage system to coordinate division of work among resources. Thus, purchasing a parallel DBMS involves decisions about all of these components, not just a DBMS.

The degree of resource sharing determines **architectures for parallel database processing**. The standard architectures are known as shared everything (SE), shared disks (SD), and shared nothing (SN) as depicted in Figure 18.11. In the SE approach, memory and disks are shared among a collection of processors. The SE approach is usually regarded as a single multiprocessing computer rather than a parallel database architecture. In the SD architecture, each processor has its private memory, but disks are shared among all processors. In the SN architecture, each processor has its own memory and disks. Data must be partitioned among the processors in the SN architecture. Partitioning is not necessary in the SD and SE architectures because each processor has access to all data.

For additional flexibility, the basic architectures are extended with clustering. A cluster is a tight coupling of two or more computers so that they behave as a single computer. Figure 18.12 extends Figure 18.11 with clustering to depict the clustered disk (CD) and clustered nothing (CN) architectures. In the CD architecture, the processors in each cluster share all disks, but nothing is shared across clusters. In the CN architecture, the processors in each cluster share no resources, but each cluster can be manipulated to work in parallel to perform a task. Figure 18.12 shows only two clusters, but the clustering approach is not limited to two clusters. For additional flexibility, the number of clusters and cluster membership can be dynamically configured. Each processor node in a cluster can be a multiprocessing computer or an individual processor.

FIGURE 18.11
Basic Parallel Database Architectures

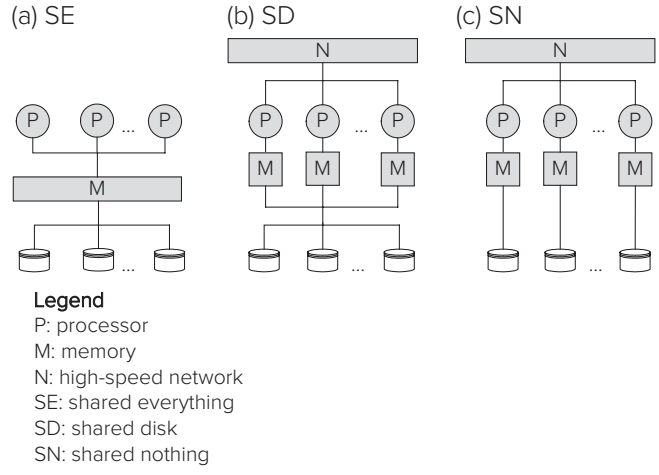
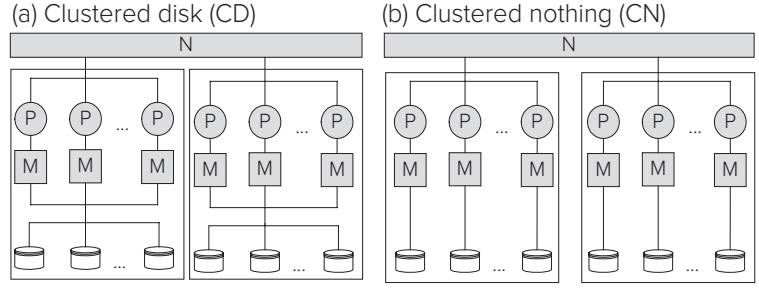


FIGURE 18.12
Parallel Database Architectures with Clustering



In all parallel database architectures, resource sharing is transparent to applications. Application code (SQL and programming language statements) does not need to be changed to take advantage of parallel database processing. In contrast, distributed database architectures presented in Section 18.4 usually do not provide transparent processing because of different goals for distributed databases.

The primary design issues that influence the performance of the parallel database architectures are load balancing, cache coherence, and interprocessor communication. Load balancing involves the amount of work allocated to different processors in a cluster. Ideally, each processor has the same amount of work to fully utilize the cluster. The CN architecture is most sensitive to load balancing because of the need for data partitioning. It can be difficult to partition a subset of a database to achieve equal division of work because data skew is common among database columns.

Cache coherence involves synchronization among local memories and common disk storage. After a processor addresses a disk page, the image of this page remains in the cache associated with the given processor. An inconsistency occurs if another processor has changed the page in its own buffer. To avoid inconsistencies when a disk page is accessed, a check of other local caches should be made to coordinate changes produced in the caches of these processors. By definition, the cache coherence problem is limited to shared disk architectures (SD and CD).

Interprocessor communication involves messages generated to synchronize actions of independent processors. Partitioned parallelism as used for the shared nothing architectures can create large amounts of interprocessor communication. In particular, partitioned join operations may generate a large amount of communication overhead to combine partial results executed on different processors.

Research and development on parallel database technology has found reasonable solutions to the problems of load balancing, cache coherence, and interprocessor communication. Commercial DBMS vendors have incorporated solutions into parallel DBMS offerings. Commercially, the CD and CN architectures are dominating. The next subsection depicts enterprise DBMSs using the CD and CN architectures.

18.3.2 Commercial Parallel Database Technology

This section presents details of enterprise DBMSs that offer parallel DBMSs. Since the classification discussed in section 18.3.1 is not sufficient to understand the differences among actual products, this section provides details about two prominent parallel DBMSs. Although both DBMSs promise high levels of performance, scalability, and availability, trade-offs among the levels are inevitable given the contrasting approaches of the two parallel DBMS approaches.

Oracle Real Application Clusters Oracle Real Application Clusters (RAC) requires an underlying hardware cluster, a group of independent servers that cooperate as a single system. The primary cluster components are processor nodes, a cluster interconnect, and a shared storage subsystem as depicted in Figure 18.13. Each server node has its own database instance with a log writing process, a database writing process, and a shared global area containing database blocks, redo log buffers, dictionary information, and shared SQL statements. All database writing processes in a cluster use the same shared storage system. Each database instance maintains its own recovery logs.

Cache Fusion technology in RAC enables synchronized access to the cache across all the nodes in a cluster, without incurring expensive disk I/O operations. The Global Cache Service (GCS) is a RAC component that implements the Cache Fusion technology. The GCS maintains a distributed directory of resources such as data blocks and queues access to the resources. To optimize performance, a resource is assigned to the node with the most frequent accesses to the resource. Access to a resource by other nodes is controlled by its master node.

Oracle RAC supports a number of features as listed below. All of these features are available without database or application partitioning.

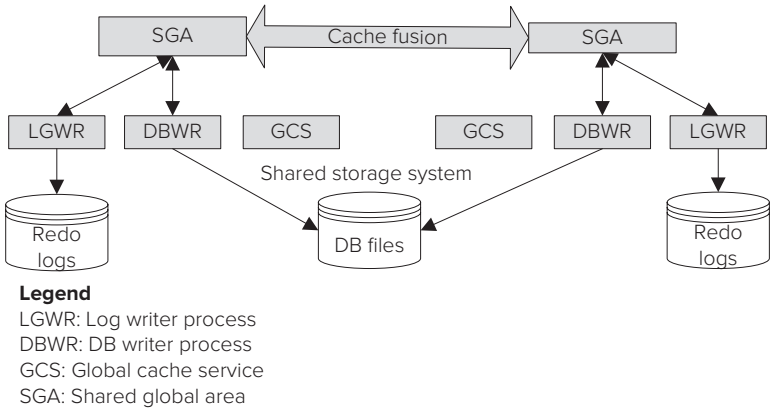


FIGURE 18.13
Example Two-Node Oracle Cluster

- The query optimizer considers the number of processors, degree of parallelism of the query, and CPU workload to distribute the workload across the nodes in the cluster and to optimally utilize the hardware resources.
- Load balancing automatically distributes connections among active nodes in a cluster based on node workload. Connection requests are sent to the least congested node in a cluster.
- Automatic failover enables fast switching from a failing node to a surviving node. Switching among nodes can be planned to allow for periodic maintenance.
- The high availability framework maintains components in a running state at all times. High availability involves monitoring critical components and restarting them if they stop. The framework enables clients to immediately react to changes, enabling application developers to hide outages and reconfigurations from end users.
- Oracle Clusterware provides a complete, integrated cluster management solution on all Oracle platforms. Oracle Clusterware features include node membership, group services, global resource management, session information for nodes in a cluster, SQL statement tracing, and high availability functions.
- Service management supports the enterprise grid vision. Services are entities that can be defined in Oracle RAC databases. Service management enables grouping of database workloads and routing the work to the most appropriate instance. In addition, resources can be assigned to services to process and monitor workloads. Applications assigned to services acquire the workload characteristics, including high availability and load balancing rules.
- In Oracle 12c, Oracle RAC supports three new features, multitenant databases, application continuity, and transaction guard. Multitenant databases consist of a container database holding pluggable databases. Oracle RAC provides local high availability to support dynamic provisioning of pluggable databases. Application continuity provides a replay capability to protect applications from instance and session failures. The replay capability avoids the need to reboot servers after a wave of logons, providing assurance that critical applications have completed. Transaction guard prevents committing duplicate transactions and committing transactions out of order after a database session failure.

IBM DB2 Enterprise Server Edition with the DPF Option The Database Partitioning Feature (DPF) option provides CN style parallel processing for DB2 databases. DB2 without the DPF option supports transparent parallel database processing for multiprocessor machines. DB2 access plans can exploit all CPUs and physical disks on a multiprocessor server. The DPF option adds the ability to partition a database across a group of machines. A single image of a database can span multiple machines and still appear to be a single database image to users and applications. Partitioned

parallelism available with the DPF option provides much higher scalability than multiprocessor parallelism alone.

Partitioning with the DPF option is transparent to users and applications. User interaction occurs through one database partition, known as the coordinator node for that user. Figure 18.14 depicts coordination in a partitioned database for multiprocessor server nodes. The database partition to which a client or application connects becomes the coordinator node. Users should be distributed across servers to distribute the coordinator function.

The DPF option supports automatic partitioning or DBA determined partitioning. With automatic partitioning, data is distributed among partitions using an updatable partitioning map and a hashing algorithm, which determine the placement and retrieval of each row of data. For DBA determined partitioning, a column can be selected as a partitioning key. With either approach, the physical placement of data is transparent to users and applications.

The DPF option of DB2 supports a number of features as listed below. All of these features require database partitioning.

- The query optimizer uses information about data partitioning to search for access plans. While comparing different access plans, the optimizer accounts for parallelism of different operations and costs associated with messaging among database partitions.
- DPF provides a high degree of scalability through its support of thousands of partitions. Near linear performance improvements have been reported for the DPF option.
- Partitioned parallelism provides improved logging performance because each partition maintains its own log files.

18.3.3 Big Data Parallel Processing Architectures

Scalable parallel processing for large data sets has been developed independent of parallel processing architectures for DBMSs. Batch processing of large text data sets in web search engines was the original impetus for big data parallel processing architectures. A typical big data task involves parsing a web log into delimited components such as request date, request time, web address, request action, and so on. A large retail organization may generate individual log files of 15GB needing to combine thousands of log files to analyze usage patterns of website visitors. These type of big data tasks involve batch processing of semi-structured data using commodity hardware and software. Scalability for big data processing remains an important requirement. In contrast, parallel processing architectures for enterprise DBMSs support a different set of requirements for optimized execution of SQL statements (mix of ad hoc and repetitive queries) with lower limits on scalability.

In response to big data processing requirements, an open source project[1] named Hadoop was developed. Hadoop provides scalable parallel processing using commodity

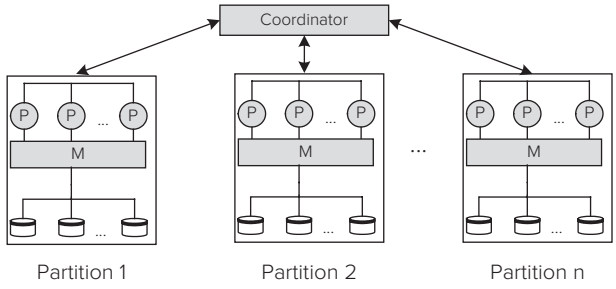
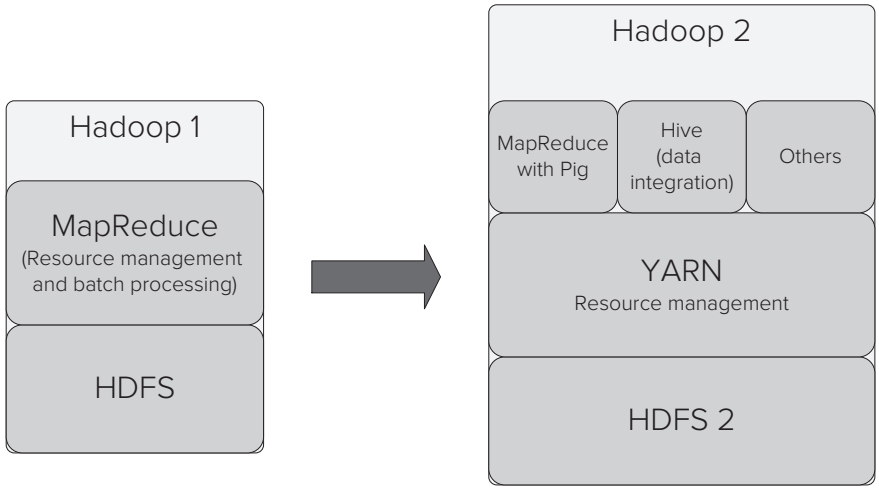


FIGURE 18.14
Coordination for Partitioned Parallelism in DPF

[1] Hadoop became an open source project of the Apache Software Foundation in 2005. Hadoop was the name of the stuffed elephant belonging to the son of one of the original developers. The Apache Software Foundation released Hadoop 2 in 2013.

FIGURE 18.15
Architectures of Hadoop 1 and Hadoop 2

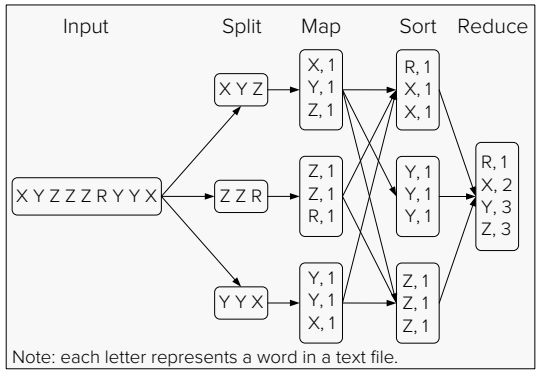


hardware and software for large data sets. Hadoop 1, the original open source project, contains two major components as shown in Figure 18.15. The Hierarchical Distributed File System (HDFS) provides reliable and scalable data storage using clusters of commodity servers. Hadoop partitions large files into equal size blocks stored in files. A large deployment in an HDFS cluster may involve 4,000 servers with 300 million files addressing 60 PB of storage. HDFS2 in Hadoop 2 improves reliability, eliminating the single point of failure of HDFS in Hadoop 1. MapReduce in Hadoop 1 combines a parallel processing model and cluster resource management. Hadoop 2 generalizes Hadoop 1 by separating parallel processing models and resource management.

MapReduce[2], the original and still widely used parallel processing model, divides a task into small parts for distributed execution. A mapping function converts input (typically unstructured such as text) into sorted <key,value> pairs. A reduce function aggregates sorted <key,value> pairs through filtering and combining operations such as counting occurrences of keys. An application developer provides customized mapping and reduce functions to apply MapReduce. The Hadoop resource management component (YARN) transparently allocates intermediate results to blocks in files and coordinates distributed processing among clusters. Figure 18.16 depicts a simple example of a MapReduce process. The MapReduce process splits text input, maps the split input into <key,value> pairs, sorts the pairs, and reduces the sorted pairs by counting the number of occurrences of each word. Twitter applies this type of MapReduce process to text files containing hundreds of millions of tweets per day.

The big data parallel processing approach initially developed in Hadoop has been extended for improved performance and new tasks. On the performance side, Apache

FIGURE 18.16
MapReduce Example to Count Words in Text



[2] Google engineers developed the MapReduce algorithm in 2004.

Spark, another open source project, provides distributed in-memory data sets (known as Resilient Distributed Datasets) to reduce overhead of intermediate file operations in Hadoop 2. Apache Hawq improves performance on analytic queries by reducing storage of intermediate results and degradation due to disk failures. On the task side, Hadoop 2, Spark, and Hawq provide programming interfaces to support SQL queries, streaming analytics, data mining, data integration, and graph computations. A new segment of the software industry has adopted and extended big data parallel architectures, funded by large levels of investment capital.

Big data parallel processing largely complements parallel processing in DBMSs. Big data parallel processing tools specialize in batch processing of large, unstructured data sets, tasks not well-suited to DBMS query components. Big data processing results can be stored in a data warehouse. Even SQL query execution in big data engines emphasizes queries on unstructured data, not well-suited to DBMS query optimization and query workloads. In recognition of the importance of big data, some enterprise DBMS vendors have integrated architectures for big data processing with a standard DBMS architecture for parallel processing.

18.4 ARCHITECTURES FOR DISTRIBUTED DATABASE MANAGEMENT SYSTEMS

Distributed DBMSs involve different technology than client-server processing and parallel database processing. Client-server processing emphasizes distribution of functions among networked computers using middleware for process management. A fundamental difference between parallel database processing and distributed database processing is autonomy. Distributed databases provide site autonomy while parallel databases do not. Thus, distributed databases require a different set of features and technology.

To support distributed database processing, fundamental extensions to a DBMS are necessary. Underlying the extensions are a different component architecture that manages distributed database requests and a different schema architecture that provides additional layers of data description. This section describes the component architecture and schema architecture to provide a foundation for more details about distributed database processing in following sections.

18.4.1 Component Architecture

Distributed DBMSs support global requests that use data stored at more than one autonomous site. A site is any locally controlled computer with a unique network address. Sites are often geographically distributed, although the definition supports sites located in close proximity. Global requests are queries that combine data from more than one site and transactions that update data at more than one site. A global request can involve a collection of statements accessing local data in some statements and remote data in other statements. Local data is controlled by the site in which a user normally connects. Remote data involves a different site in which a user may not even have an account to access. If all requests require only data from one site, distributed database processing capabilities are not required.

To depict global requests, you need to begin with a distributed database. Distributed databases are potentially useful for organizations operating in multiple locations with local control of computing resources. Figure 18.17 depicts a distributed database for an electronic retail business. The company performs customer processing at Boise and Tulsa and manages warehouses at Seattle and Denver. The distribution of the database follows the geographical locations of the business. The *Customer*, *OrderTbl*, and *OrderLine* tables (customer-order data) are split between Boise and Tulsa, while

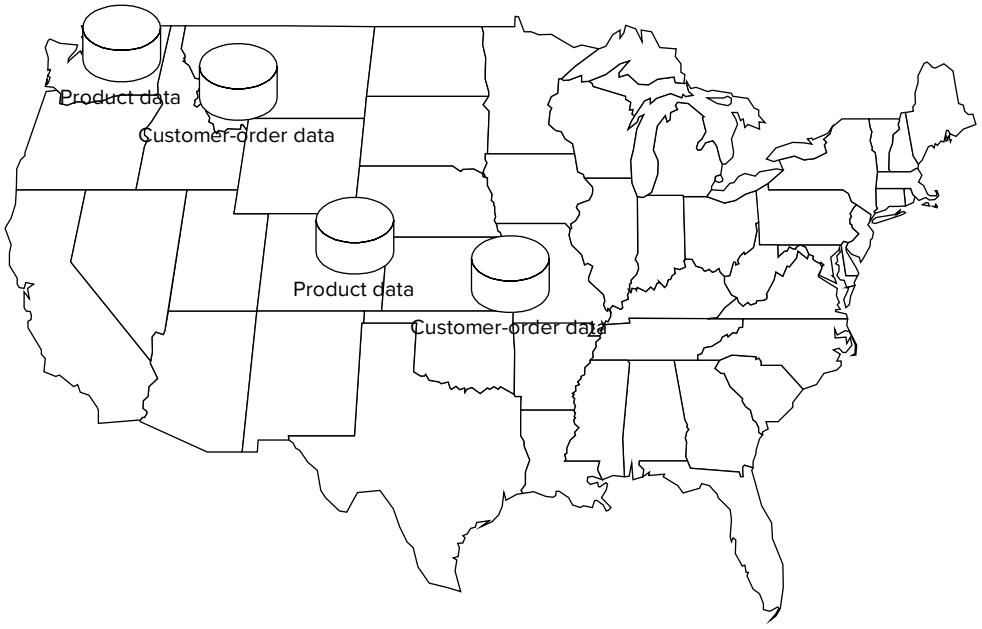


FIGURE 18.17
Distribution of Order-Entry Data

the *Product* and *Inventory* tables (product data) are split between Seattle and Denver. An example of a global query is to check both warehouse sites for sufficient quantity of a product to satisfy a shipment invoice. An example of a global transaction is an order-entry form that inserts rows into the *OrderTbl* and *OrderLine* tables at one location and updates the *Product* table at the closest warehouse site.

To support global queries and transactions, distributed DBMSs contain additional components as compared to traditional, nondistributed DBMSs. Figure 18.18 depicts a possible arrangement of the components of a distributed DBMS. Each server with access to the distributed database is known as a *site*. If a site contains a database, a local data manager (LDM) controls it. The local data managers provide complete features of a DBMS as described in other chapters. The distributed data manager (DDM) optimizes query execution across sites, coordinates concurrency control and recovery across sites, and controls access to remote data. In performing these tasks, the DDM uses the global dictionary (GD) to locate parts of the database. The GD can be distributed to various sites similar to the way that data are distributed. Because of the complexity of the distributed database manager, section 18.6 presents more details about distributed query processing and transaction processing.

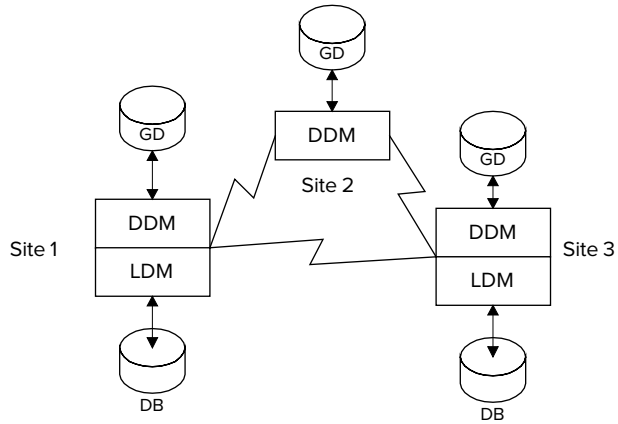


FIGURE 18.18
Component Architecture of a Distributed DBMS

In the component architecture, the local database managers can be homogeneous or heterogeneous. A distributed DBMS with homogeneous local DBMSs is *tightly integrated*. The distributed database manager can call internal components and access the internal state of local data managers. The tight integration allows the distributed DBMS to efficiently support both distributed queries and transactions. However, the homogeneity requirement precludes integration of existing databases.

A distributed DBMS with heterogeneous local data managers is *loosely integrated*. The distributed database manager acts as middleware to coordinate local data managers. SQL often provides the interface between the distributed data manager and the local data managers. The loose integration supports data sharing among legacy systems and independent organizations. However, the loosely integrated approach may not be able to support transaction processing in a reliable and efficient manner.

18.4.2 Schema Architectures

To accommodate distribution of data, additional layers of data description are necessary. However, there is no widely accepted schema architecture for distributed databases like the widely accepted Three Schema Architecture for traditional DBMSs. This section depicts possible schema architectures for tightly integrated distributed DBMSs and loosely integrated distributed DBMSs. The architectures provide a reference about the kinds of data description necessary and the compartmentalization of the data description.

The schema architecture for a tightly integrated distributed DBMS contains additional layers for fragmentation and allocation as depicted in Figure 18.19. The fragmentation schema contains the definition of each fragment while the allocation schema contains the location of each fragment. A fragment can be defined as a vertical subset (project operation), a horizontal subset (restrict operation), or a mixed fragment (combination of project and restrict operations). A fragment is typically allocated to one site. If the distributed DBMS supports replication, a fragment can be allocated to multiple sites. Typically, one copy of a fragment is considered the primary copy and the other copies are secondary. Only the primary copy is guaranteed to be current.

The schema architecture for a loosely integrated distributed DBMS supports more autonomy of local database sites in addition to data sharing. Each site contains the traditional three schema levels, as depicted in Figure 18.20. To support data sharing, the distributed DBMS provides a local mapping schema for each site. The local mapping schemas describe the exportable data at a site and provide conversion rules to translate data from a local format into a global format. The global conceptual schema depicts all

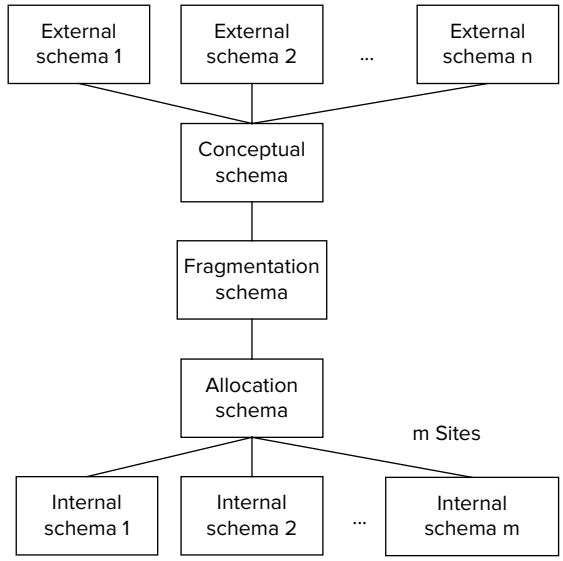


FIGURE 18.19
Schema Architecture for a Tightly Integrated Distributed DBMS

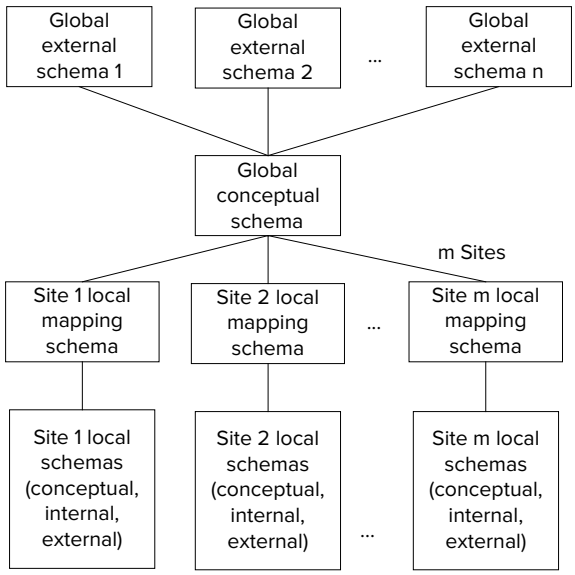


FIGURE 18.20
Schema Architecture for a Loosely Integrated Distributed DBMS

of the kinds of data and relationships that can be used in global requests. Some distributed DBMSs do not have a global conceptual schema. Instead, global external schemas provide views of shared data in a common format.

There can be many differences among the local data formats. Local sites may use different DBMSs, each with a different set of data types. The data models of the local DBMSs can be different, especially if legacy systems are being integrated. Legacy systems might use file interfaces and navigational data models (network and hierarchical) that do not support SQL. Even if local sites support a common SQL standard, there can be many differences such as different data types, scales, units of measure, and codes. The local mapping schemas resolve these differences by providing conversion rules that transform data from a local format into a global format.

The commercial realization of loosely integrated distributed databases has been partially eclipsed by XML document standards and information aggregators. XML schemas provide a set of rules governing document types to facilitate data interchange among independent organizations. An organization publishes XML documents conforming to a particular schema to provide access to a wide range of organizations. Information aggregators use web pages, XML documents and schemas, and other data gathering methods to provide access to a wide range of heterogeneous data including public records, home sales, credit scores, and comparison shopping. In a sense, XML schemas play the role of global conceptual schemas. Cooperating organizations provide the mapping to convert between local data stored in various formats and XML documents conforming to the schema rules.

18.5 TRANSPARENCY FOR DISTRIBUTED DATABASE PROCESSING

Recall from Chapter 17 that transparency refers to the visibility (visible or hidden) of internal details of a service. In transaction processing, concurrency and recovery services are transparent, or hidden from database users. Parallel database processing emphasizes transparency. In distributed database processing, transparency is related to data independence. If database distribution is transparent, users can write queries with no knowledge of the distribution. In addition, distribution changes will not cause changes to existing queries and transactions. If the database distribution is not transparent, users must reference some distribution details in queries and distribution changes can lead to changes in existing queries.

This section describes common levels of transparency and provides examples of query formulation with each level. Before discussing transparency levels, a motivating example is presented.

18.5.1 Motivating Example

To depict the levels of transparency, more details about the order-entry database are provided. The order-entry database contains five tables, as shown in the relationship diagram of Figure 18.21. You may assume that customers are located in two regions (East and West) and products are stored in two warehouses (1: Denver, 2: Seattle).

One collection of fragments can be defined using the customer region column as shown in Table 18-3. The *Western-Customers* fragment consists of customers with a region equal to "West." There are two related fragments: the *Western-Orders* fragment, consisting of orders for western customers and the *Western-OrderLines* fragment, consisting of order lines matching western orders. Similar fragments are defined for rows involving eastern customers.

The order and order line fragments are derived from a customer fragment using the semi-join operator. A **semi-join** is half of a join: the rows of one table that match the rows of another table. For example, a semi-join operation defines the *Western-Orders* fragment as the rows of the *OrderTbl* table matching customer rows with a region of "West." A fragment defined with a semi-join operation is sometimes called a derived horizontal fragment. Because some fragments should have rows related to other fragments, the semi-join operator is important for defining fragments.

Semi-Join
an operator of relational algebra that is especially useful for distributed database processing. A semi-join is half of a join: the rows of one table that match with at least one row of another table. Only the rows of the first table appear in the result.

Warehouse fragments are defined using the *WarehouseNo* column as shown in Table 18-4. In the fragment definitions, warehouse number 1 is assumed to be located in Denver and warehouse number 2 in Seattle. The *Product* table is not fragmented because the entire table is replicated at multiple sites.

Fragmentation can be more complex than described in the order-entry database. There can be many additional fragments to accommodate a business structure. For example, if there are additional customer processing centers and warehouses, additional fragments can be defined. In addition, vertical fragments can be defined as

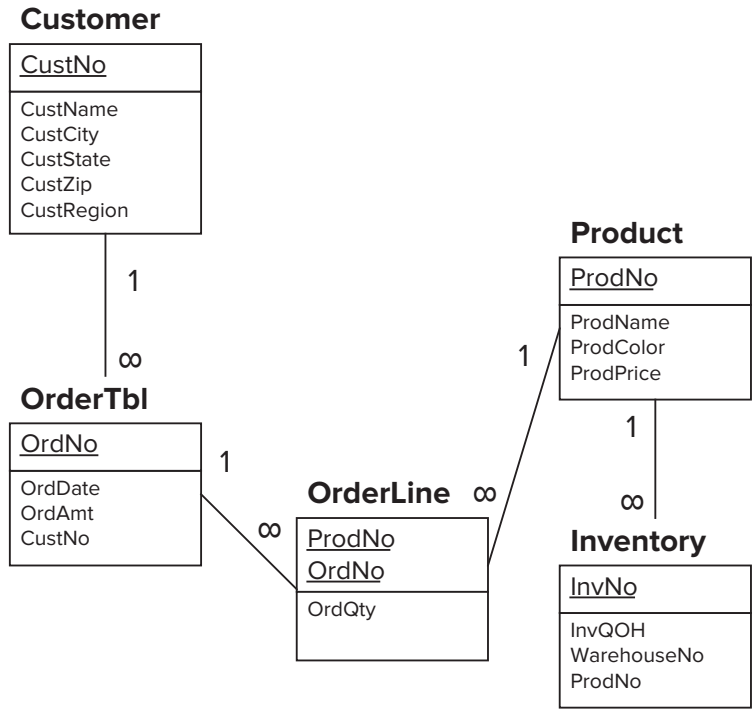


FIGURE 18.21
Relationship Diagram for the Order-Entry Database

TABLE 18-3
Fragments Based on the *CustRegion* Column[3]

```
CREATE FRAGMENT Western-Customers AS
  SELECT * FROM Customer WHERE CustRegion = 'West'
CREATE FRAGMENT Western-Orders AS
 SELECT OrderTbl.* FROM OrderTbl, Customer
  WHERE OrderTbl.CustNo = Customer.CustNo AND CustRegion = 'West'
CREATE FRAGMENT Western-OrderLines AS
 SELECT OrderLine.* FROM Customer, OrderLine, OrderTbl
  WHERE OrderLine.OrdNo = OrderTbl.OrdNo
    AND OrderTbl.CustNo = Customer.CustNo AND CustRegion = 'West'
CREATE FRAGMENT Eastern-Customers AS
 SELECT * FROM Customer WHERE CustRegion = 'East'
CREATE FRAGMENT Eastern-Orders AS
 SELECT OrderTbl.* FROM OrderTbl, Customer
  WHERE OrderTbl.CustNo = Customer.CustNo AND CustRegion = 'East'
CREATE FRAGMENT Eastern-OrderLines AS
 SELECT OrderLine.* FROM Customer, OrderLine, OrderTbl
  WHERE OrderLine.OrdNo = OrderTbl.OrdNo
    AND OrderTbl.CustNo = Customer.CustNo AND CustRegion = 'East'
```

TABLE 18-4
Fragments Based on the *WareHouseNo* Column

```
CREATE FRAGMENT Denver-Inventory AS
  SELECT * FROM Inventory WHERE WarehouseNo = 1
CREATE FRAGMENT Seattle-Inventory AS
  SELECT * FROM Inventory WHERE WarehouseNo = 2
```

projection operations in addition to the horizontal fragments using restriction and semi-join operations. A fragment can even be defined as a combination of projection, restriction, and semi-join operations. The only limitation is that the fragments must be disjoint. Disjointness means that horizontal fragments do not contain common rows and vertical fragments do not contain common columns except for the primary key.

After fragments are defined, they are allocated to sites. Fragments are sometimes defined based on where they should be allocated. The allocation of order-entry fragments follows this approach as shown in Table 18-5. The Boise site contains the western customer fragments, while the Tulsa site contains the eastern customer fragments. Similarly, the inventory fragments are split between the Denver and the Seattle sites. The *Product* table is replicated at the Denver and the Seattle sites because each warehouse stocks every product.

In practice, the design and the allocation of fragments is much more difficult than depicted here. Designing and allocating fragments is similar to index selection. Data about the distribution of queries, the distribution of parameter values in queries, and the behavior of the global query optimizer are needed. In addition, data about the frequency of originating sites for each query are needed. The originating site for a query is the site in which the query is stored or submitted from. Just as for index selection, optimization models and tools can aid decision making about fragment design and allocation. The details of the optimization models and the tools are beyond the scope

TABLE 18-5
Allocation of Fragments of the Order Entry Database

Fragments	Site
Western-Customers, Western-Orders, Western-OrderLines	Boise
Eastern-Customers, Eastern -Orders, Eastern-OrderLines	Tulsa
Denver-Inventory, Product	Denver
Seattle-Inventory, Product	Seattle

[3] The syntax in Table 18-3 is hypothetical as standard SQL does not support fragment creation.

of this book. The references at the end of the chapter provide details about fragment design and allocation.

18.5.2 Fragmentation Transparency

Fragmentation Transparency
a level of independence in distributed DBMSs in which queries can be formulated without knowledge of fragments.

Fragmentation transparency provides the highest level of data independence. Users formulate queries and transactions without knowledge of fragments, locations, or local formats. If fragments, locations, or local formats change, queries and transactions are not affected. In essence, users perceive the distributed database as a centralized database. Fragmentation transparency involves the least work for users but the most work for distributed DBMSs. Parallel database processing in shared nothing architectures involves fragmentation transparency.

To contrast the transparency levels, Table 18-6 lists some representative queries and transactions that use the order entry database. In these queries, the parameters $X and $Y are used rather than individual values. With fragmentation transparency, queries and transactions can be submitted without change regardless of the fragmentation of the database.

18.5.3 Location Transparency

Location Transparency
a level of independence in distributed DBMSs in which queries can be formulated without knowledge of locations. However, knowledge of fragments is necessary.

Location transparency provides a lesser degree of data independence than fragmentation transparency. Users need to reference fragments in formulating queries and transactions. However, knowledge of locations and local formats is not necessary. Even though site knowledge is not necessary, users are indirectly aware of a database's distribution because many fragments are allocated to a single site. Users may make an association between fragments and sites.

Location transparency involves more work in formulating requests as shown in Table 18-7. In the "find" queries, the union operator collects rows from all fragments. The Update Inventory query involves about the same amount of coding. The user substitutes a fragment name in place of the condition on *WarehouseNo* because this condition defines the fragment.

In the Customer Move request, much more coding is necessary. An update operation cannot be used because the column to update defines the fragment. Instead, rows must be inserted in the new fragments and deleted from the old fragments. For the customer fragment, the SELECT … INTO statement stores field values in variables that are used in the subsequent INSERT statement. The deletions are performed in the stated order if referenced rows must be deleted last. If deletions cascade, only one DELETE statement on the *Western-Customers* fragment is necessary.

The SQL statements for the first two requests do not reveal the number of union operations that may be required. With two fragments, only one union operation is necessary. With *n* fragments, *n-1* union operations are necessary, however.

TABLE 18-6
Representative Requests Using the Order-Entry Database

Find Order

```
SELECT * FROM OrderTbl, Customer
  WHERE OrderTbl.Custno = $X
    AND OrderTbl.CustNo = Customer.CustNo
```

Find Product Availability

```
SELECT * FROM Inventory
  WHERE ProdNo = $X
```

Update Inventory

```
UPDATE Inventory SET InvQOH = InvQOH - 1
  WHERE ProdNo = $X AND WarehouseNo = $Y
```

Customer Move

```
UPDATE Customer SET CustRegion = $X
   WHERE CustNo = $Y
```

TABLE 18-7
Requests Written with Location Transparency

Find Order

```
SELECT * FROM Western-Orders, Western-Customers
  WHERE Western-Orders.CustNo = $X
    AND Western-Orders.CustNo = Western-Customers.CustNo
UNION
SELECT * FROM Eastern-Orders, Eastern-Customers
  WHERE Eastern-Orders.CustNo = $X
    AND Eastern-Orders.Custno = Eastern-Customers.CustNo
```

Find Product Availability

```
SELECT * FROM Denver-Inventory
  WHERE ProdNo = $X
UNION
SELECT * FROM Seattle-Inventory
  WHERE ProdNo = $X
```

Update Inventory (Denver)

```
UPDATE Denver-Inventory SET InvQOH = InvQOH - 1
  WHERE ProdNo = $X
```

Customer Move (West to East)

```
SELECT CustName, CustCity, CustState, CustZip
  INTO $CustName, $CustCity, $CustState, $CustZip
  FROM Western-Customers WHERE CustNo = $Y

INSERT INTO Eastern-Customers
 (CustNo, CustName, CustCity, CustState, CustZip, CustRegion)
 VALUES ($Y, $CustName, $CustCity, $CustState, $CustZip, 'East')

INSERT INTO Eastern-Orders
 SELECT * FROM Western-Orders WHERE CustNo = $Y

INSERT INTO Eastern-OrderLines
 SELECT * FROM Western-OrderLines
  WHERE OrdNo IN
   (SELECT OrdNo FROM Western-Orders WHERE CustNo = $Y)

DELETE FROM Western-OrderLines
  WHERE OrdNo IN
   (SELECT OrdNo FROM Western-Orders WHERE CustNo = $Y)

DELETE Western-Orders WHERE CustNo = $Y

DELETE Western-Customers WHERE CustNo = $Y
```

To some extent, views can shield users from some of the fragment details. For example, using a view defined with union operations would obviate the need to write the union operations in the query. However, views may not simplify manipulation statements. If a DBMS does not support fragmentation transparency, it seems unlikely that updatable views could span sites. Thus, the user would still have to write the SQL statements for the Customer Move request.

18.5.4 Local Mapping Transparency

Local Mapping Transparency
a level of independence in distributed DBMSs in which queries can be formulated without knowledge of local formats. However, knowledge of fragments and fragment allocations (locations) is necessary.

Local mapping transparency provides a lesser degree of data independence than location transparency. Users need to reference fragments at sites in formulating queries and transactions. However, knowledge of local formats is not necessary. If sites differ in formats as in loosely integrated distributed databases, local mapping transparency still relieves the user of considerable work.

Location transparency may not involve much additional coding effort from that shown in Table 18-8. The only changes between Tables 18-7 and 18-8 are the addition

TABLE 18-8
Requests Written with Local Mapping Transparency

Find Order

```
SELECT *
 FROM Western-Orders@Boise, Western-Customers@Boise
  WHERE Western-Orders@Boise.CustNo = $X
    AND Western-Orders@Boise.CustNo =
        Western-Customers@Boise.CustNo
UNION
SELECT *
 FROM Eastern-Orders@Tulsa, Eastern-Customers@Tulsa
 WHERE Eastern-Orders@Tulsa.CustNo = $X
   AND Eastern-Orders@Tulsa.CustNo =
       Eastern-Customers@Tulsa.CustNo
```

Find Product Availability

```
SELECT * FROM Denver-Inventory@Denver
  WHERE ProdNo = $X
UNION
SELECT * FROM Seattle-Inventory@Seattle
  WHERE ProdNo = $X
```

Update Inventory (Denver)

```
UPDATE Denver-Inventory@Denver SET InvQOH = InvQOH - 1
  WHERE ProdNo = $X
```

Customer Move (West to East)

```
SELECT CustName, CustCity, CustState, CustZip
  INTO $CustName, $Custcity, $CustState, $CustZip
  FROM Western-Customers@Boise WHERE CustNo = $Y

INSERT INTO Eastern-Customers@Tulsa
 (CustNo, CustName, CustCity, CustState, CustZip, CustRegion)
 VALUES ($Y, $CustName, $CustCity, $CustState, $CustZip, 'East')

INSERT INTO Eastern-Orders@Tulsa
 SELECT * FROM Western-Orders@Boise WHERE CustNo = $Y

INSERT INTO Eastern-OrderLines@Tulsa
 SELECT * FROM Western-OrderLines@Boise
  WHERE OrdNo IN
   (SELECT OrdNo FROM Western-Orders@Boise
     WHERE CustNo = $Y)

DELETE FROM Western-OrderLines@Boise
  WHERE OrdNo IN
   (SELECT OrdNo FROM Western-Orders@Boise
     WHERE CustNo = $Y)

DELETE Western-Orders@Boise WHERE CustNo = $Y

DELETE Western-Customers@Boise WHERE CustNo = $Y
```

of the site names in Table 18-8. If fragments are replicated, additional coding is necessary in transactions. For example, if a new product is added, two INSERT statements (one for each site) are necessary with local mapping transparency. With location transparency, only one INSERT statement is necessary. The amount of additional coding depends on the amount of replication.

From the discussion in this section, you may falsely assume that fragmentation transparency is preferred to the other levels of transparency. Fragmentation transparency provides the highest level of data independence but is the most complex to implement. For parallel database processing in shared nothing architectures, fragmentation transparency is a key feature. For distributed databases, fragmentation transparency conflicts with the goal of site autonomy. Data ownership implies user awareness

when combining local and remote data. In addition, fragmentation transparency may encourage excessive resource consumption because users do not perceive the underlying distributed database processing. With location and local mapping transparency, users perceive the underlying distributed database processing at least to some extent. The amount and complexity of distributed database processing can be considerable, as described in section 18.6.

18.5.5 Transparency in Oracle Distributed Databases

Oracle provides two technologies for distributed databases. Oracle supports site autonomy for distributed databases using database links. Oracle supports fragmentation transparency through partitioning. This section provides an overview of both technologies.

Oracle Distributed Database Technology for Site Autonomy Oracle supports site autonomy for homogeneous and heterogeneous distributed databases. In the homogenous case, each site contains a separately managed Oracle database. The requirement for separate management provides autonomy for each participating site. Individual databases may utilize any supported Oracle version although functionality in global requests is limited to the lowest version database. Oracle supports replication in distributed databases through designated master sites and asynchronous processing at secondary sites. Non-Oracle databases can also participate in global requests using Heterogeneous Services and Gateway Agents. This section provides details about transparency in pure (nonreplicated) homogeneous distributed databases.

Database links are a key concept for Oracle distributed databases. A database link provides a one-way connection from a local database to a remote database. A local database is the database in which a user connects. A remote database is another database in which a user wants to access in a global request. Database links allow a user to access another user's objects in a remote database without having an account on the remote site. When using a database link, a remote user is limited by the privilege set of the object's owner.

Table 18-9 demonstrates Oracle statements for creating links and synonyms as well as using the links and synonyms in a global query. The first statement creates a fixed link to the database "boise.acme.com" through the remote user "clerk1." The next statement uses the link to access the remote *OrderTbl* table. It is assumed that the current user is connected to the Tulsa database and uses the link to access the Boise database. In the FROM clause, the unqualified table names on both sites are the same (*OrderTbl* and *Customer*). The CREATE SYNONYM statements create aliases for remote tables using the remote table names and link names. The SELECT statements can use the synonyms in place of table and link names.

As Table 18-9 demonstrates, database links provide local mapping transparency to remote data. To create a link, a user must know the global database name. A global database name usually contains information about an organization's structure and business locations. To use a link, a user must know the remote object names and details. Local mapping transparency is consistent with an emphasis on site autonomy. Oracle allows more transparency (location transparency) in remote data access through the usage of synonyms and views.

Oracle provides more features for links than depicted in Table 18-9. The details include link scopes (public, private, and global), link users, and administration of roles and privileges for remote database access. The link scopes differ according to which user groups are allowed to use the link to access to remote data. Table 18-10 provides a summary of the link scopes. The link owner should determine who can use the link to access remote data. Fixed users are the simplest to understand but connected users and current users provide more flexibility as summarized in Table 18-11. The Oracle Database Administrators Guide provides more details about managing database links including security administration.

Create Link

```
CREATE DATABASE LINK boise.acme.com CONNECT TO clerk1
   IDENTIFED BY clerk1
```

Find Order (using the link name)

```
SELECT *
 FROM OrderTbl@boise.acme.com WO, Customer@boise.acme.com WC
  WHERE WO.CustNo = 1111111
    AND WO.CustNo = WC.CustNo
          UNION
SELECT *
 FROM OrderTbl, Customer
 WHERE OrderTbl.CustNo = 1111111
   AND OrderTbl.CustNo = Customer.CustNo
```

Create Synonyms

```
CREATE PUBLIC SYNONYM BoiseOrder FOR OrderTbl@boise.acme.com;
CREATE PUBLIC SYNONYM BoiseCustomer FOR Customer@boise.acme.com;
```

Find Order (using the synonym names)

```
SELECT *
 FROM BoiseOrder WO, BoiseCustomer WC
  WHERE WO.CustNo = 1111111
    AND WO.CustNo = WC.CustNo
          UNION
SELECT *
 FROM OrderTbl, Customer
 WHERE OrderTbl.CustNo = 1111111
   AND OrderTbl.CustNo = Customer.CustNo
```

TABLE 18-9 Oracle Statements for a Global Request using a Link

Scope	Details
Private	More secure than a public or global link, because only the owner of the private link, or subprograms within the same schema, can use the link to access the remote database
Public	A database-wide link in which all users and PL/SQL subprograms in the database can use the link to access database objects in the corresponding remote database
Global	A network-wide link in which users and PL/SQL subprograms in any database can use the link to access objects in the corresponding remote database; Simplifies link management

TABLE 18-10 Summary of Database Link Scopes

User Type	Details
Connected User	A local user accessing a database link in which no fixed username and password have been specified; A user referencing the link connects to the remote database as the same user. Credentials do not have to be stored in the data dictionary.
Current User	Utilizes a global user who must be authenticated and be a user on both databases involved in the link. However, the user invoking the current user link does not have to be a global user if accessing a remote database through a stored procedure.
Fixed User	A user whose username/password is part of the link definition; Connects a user in a primary database to a remote database with the security context of the user specified in the connect string.

TABLE 18-11 Summary of Link User Types

Oracle Partitioning Technology for Fragmentation Transparency Oracle partitioning technology provides a hybrid approach combining parallel and distributed database features. The goals of partitioning in Oracle are more performance oriented rather than location oriented for distributed databases. Oracle partitioning

TABLE 18-12
Summary of Partitioning Options in Oracle 12c

Partition Option	Description	Typical Usage
Range	Consecutive ranges with catch all provision	Time based partitioning for orders, shipments, and reservations
List	Enumeration of unordered values	Country, region, or area code partitioning
Hash	Oracle hash function to spread values into groups	Hash on primary key such as order number
Interval	Extension to range partitioning with equal width ranges; On demand partitions after first partition	Fixed time period such as day, month , or year
REF	Partition a child table consistent with parent table partition	Partition details for orders, shipments, or reservations consistent with partitioning on parent such as by time period or region
Composite	Combine range, list, and hash options	Partition shipment table by region (list) and then sub partitioned by hash on customer number
Virtual column	Partition by calculated column	Partition shipment table on quarter calculated from shipment date

supports improved performance for large tables with high availability requirements. Partitioning can improve query performance especially for large data warehouse tables. Partitioning supports higher availability because individual partitions can experience downtime while other partitions remain available. Backup, restore, and load are faster to perform on partitions rather than on a single much larger table. Storage options for partitions can vary providing flexibility for physical database design.

Oracle provides fragmentation transparency for partitioned tables. A partitioned table is similar to a horizontal fragment with a variety of fragmentation options. Queries do not reference partitions. Oracle maps SQL statements on tables into SQL statements referencing associated partitions, executes queries against partitions, and collects results from partition operations.

Oracle provides a variety of partitioning options as summarized in Table 18-12. Range, list, and hash partitioning are the basic partitioning options. Interval, composite, REF, and virtual column are secondary partition options often combining with primary partition options.

Due to the complexity of selecting among varied partitioning options, Oracle provides a partition advisor integrated with the SQL Access Advisor. The Partition Advisor makes recommendations for partitions only or partitions as part of other performance recommendations.

18.6 DISTRIBUTED DATABASE PROCESSING

Just as distributed data adds complexity for query formulation, distributed data adds considerable complexity to query processing and transaction processing. Distributed database processing involves movement of data, remote processing, and site coordination that are absent from centralized database processing. Although the details of distributed database processing can be hidden from programmers and users, performance implications sometimes cannot be hidden. This section presents details about distributed query processing and distributed transaction processing to make you aware of complexities that can affect performance.

18.6.1 Distributed Query Processing

Distributed query processing is more complex than centralized query processing for several reasons. Distributed query processing involves both local (intrasite) and global

(intersite) optimization. Global optimization involves data movement and site selection decisions that are absent in centralized query processing. For example, to perform a join of distributed fragments, one fragment can be moved, both fragments can be moved to a third site, or just the join values of one fragment can be moved. If the fragments are replicated, then a site for each fragment must be chosen.

Many of the complexities of distributed query processing also exist for parallel databases with shared nothing architectures. The major difference is the much faster and more reliable communication networks used in parallel database processing.

Distributed query processing is also more complex because multiple optimization objectives exist. In a centralized environment, minimizing resource (input-output and processing) usage is consistent with minimizing response time. In a distributed environment, minimizing resources may conflict with minimizing response time because of parallel processing opportunities. Parallel processing can reduce response time but increase the overall amount of resources consumed (input-output, processing, and communication). In addition, weighting of communication costs versus local costs (input-output and processing) depends on network characteristics. For public networks such as the Internet, communication costs can dominate local costs. For local area networks and private networks, communication costs are more equally weighted with local costs.

Increased complexity makes optimization of distributed queries even more important than optimization of centralized queries. Because distributed query processing involves both local and global optimization, there are many more possible access plans for a distributed query than a corresponding centralized query. Variance in performance among distributed access plans can be quite large. The choice of a bad access plan can lead to extremely poor performance. In addition, distributed access plans sometimes need to adjust for site conditions. If a site is unavailable or overloaded, a distributed access plan should dynamically choose another site. Thus, some of the optimization process may need to be performed dynamically (during run-time) rather than statically (during compile time).

To depict the importance of distributed query optimization, access plans for a sample query are presented. To simplify the presentation, a public network with relatively slow communication times is used. Only communication times (*CT*) are shown for each access plan. Communication time consists of a fixed message delay (*MD*) and a variable transmission time (*TT*). Each record is transmitted as a separate message.

$CT = MD + TT$

MD = Number of messages * Delay per message
TT = Number of bits / Data rate

Global Query: List the order number, order date, product number, product name, product price, and order quantity for eastern orders with a specified customer number, date range, and product color. Table 18-13 lists statistics for the query and the network.

```
SELECT EO.OrdNo, OrdDate, P.ProdNo, OrdQty,
       ProdName, ProdPrice
  FROM Eastern-Orders EO, Eastern-Orderlines EOL, Product P
  WHERE EO.CustNo = $X AND EO.OrdNo = EOL.OrdNo
    AND ProdColor = 'Red' AND EOL.ProdNo = P.ProdNo
    AND OrdDate BETWEEN $Y AND $Z
```

1. Move the *Product* table to the Tulsa site where the query is processed.
   CT = 1,000 * 0.1 + (1,000 * 1,000) / 1,000,000 = 101 seconds
2. Restrict the *Product* table at the Denver site. Then move the result to Tulsa where the remainder of the query is processed.
   CT = 200 * 0.1 + (200 * 1,000) / 1,000,000 = 20.2 seconds

TABLE 18-13
Statistics for the Query and the Network

Record length is 1,000 bits for each table.
The customer has 5 orders in the specified date range.
Each order contains an average of 5 products.
The customer has 3 orders in the specified date range and color.
There are 200 red products.
There are 10,000 orders, 50,000 order lines, and 1,000 products in the fragments.
Fragment allocation is given in Table 18-5.
Delay per Message is 0.1 second.
Data Rate is 1,000,000 bits per second.

3. Perform join and restrictions of *Eastern-Orders* and *Eastern-OrderLines* fragments at the Tulsa site. Move the result to the Denver site to join with the *Product* table.
   CT = 25 * 0.1 + (25 * 2,000) / 1,000,000 = 2.55 seconds
4. Restrict the *Product* table at the Denver site. Move only the resulting product numbers (32 bits) to Tulsa. Perform joins and restrictions at Tulsa. Move the results back to Denver to combine with the *Product* table.
   CT (Denver to Tulsa) = 200 * 0.1 + (200 * 32) / 1,000,000 = 20.0064 seconds
   CT (Tulsa to Denver) = 15 * 0.1 + (15 * 2,000) / 1,000,000 = 1.53 seconds
   CT = *CT* (Denver to Tulsa) + CT (Tulsa to Denver) = 21.5364 seconds

These access plans show a wide variance in communication times. Even more variance would be shown if the order fragments were moved from Tulsa to Denver. The third access plan dominates the others because of its low message delay. Additional analysis of the local processing costs would be necessary to determine the best access plan.

18.6.2 Distributed Transaction Processing

Distributed transaction processing follows the principles described in Chapter 17. Transactions obey the ACID properties and the distributed DBMS provides concurrency and recovery transparency. However, a distributed environment makes the implementation of the principles more difficult. Independently operating sites must be coordinated. In addition, new kinds of failures exist because of a communication network. To deal with these complexities, new protocols are necessary. This section presents an introduction to the problems and solutions of distributed concurrency control and commit processing.

Distributed Concurrency Control Distributed concurrency control can involve more overhead than centralized concurrency control because local sites must be coordinated through messages over a communication network. The simplest scheme involves centralized coordination, as depicted in Figure 18.22. At the beginning of a transaction, the coordinating site is chosen and the transaction is divided into subtransactions performed at other sites. Each site hosting a subtransaction submits lock and release requests to the coordinating site using the normal two-phase locking rules.

Centralized coordination involves the fewest messages and the simplest deadlock detection. However, reliance on a centralized coordinator may make transaction processing less reliable. To alleviate reliance on a centralized site, lock management can be distributed among sites. The price for higher reliability is more message overhead and more complex deadlock detection. The number of messages can be twice as much in the distributed coordination scheme as compared to the centralized coordination scheme.

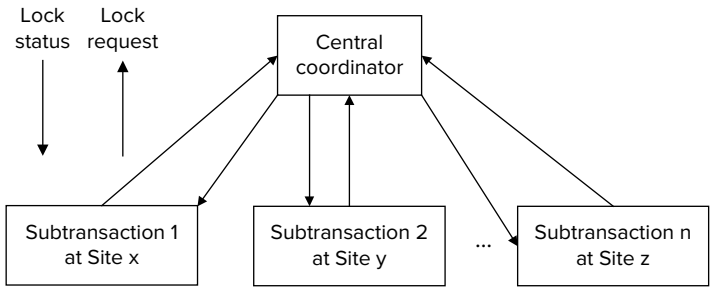


FIGURE 18.22
Centralized Concurrency Control

With both centralized and distributed coordination, replicated data are a problem. Updating replicated data involves extra overhead because a write lock must be obtained on all copies before any copy is updated. Obtaining write locks on multiple copies can cause delays and even rollbacks if a copy is not available.

To reduce overhead with locking multiple copies, the **primary copy protocol** can be used. In the primary copy protocol, one copy of each replicated fragment is designated as the primary copy, while the other copies are secondary. Write locks are necessary only for the primary copy. After a transaction updates the primary copy, updates are propagated to secondary copies. However, secondary copies may not be updated until after the transaction commits. The primary copy protocol provides improved performance but at the cost of noncurrent secondary copies. Because reduced overhead is often more important than current secondary copies, many distributed DBMSs use the primary copy protocol.

Primary Copy Protocol
a protocol for concurrency control of distributed transactions. Each replicated fragment is designated as either the primary copy or a secondary copy. During distributed transaction processing; only the primary copy is guaranteed to be current at the end of a transaction. Updates may be propagated to secondary copies after end of transaction.

Distributed Commit Processing Distributed DBMSs must contend with failures of communication links and sites, failures that do not affect centralized DBMSs. Detecting failures involves coordination among sites. If a link or site fails, any transaction involving the site must be aborted. In addition, the site should be avoided in future transactions until the failure is resolved.

Failures can be more complex than just a single site or communication link. A number of sites and links can fail simultaneously leaving a network partitioned. In a partitioned network, different partitions (collections of sites) cannot communicate although sites in the same partition can communicate. The transaction manager must ensure that different parts of a partitioned network act in unison. It should not be possible for sites in one partition to decide to commit a transaction but sites in another partition to decide not to commit a transaction. All sites must either commit or abort.

The most widely known protocol for distributed commit processing is the **two-phase commit protocol**[4]. For each transaction, one site is chosen as the coordinator and the transaction is divided into subtransactions performed at other participant sites. The coordinator and the participant sites interact in a voting phase and a decision phase. At the end of both phases, each participant site has acted in unison to either commit or abort its subtransaction.

Two-Phase Commit Protocol (2PC)
a rule to ensure that distributed transactions are atomic. 2PC uses a voting and a decision phase to coordinate commits of local transactions.

The voting and decision phases require actions on both the coordinator and the participant sites as depicted in Figure 18.23. In the decision phase, the coordinator sends a message to each participant asking if it is ready to commit. Before responding, each participant forces all updates to disk when the local transaction work finishes. If no failure occurs, a participant writes a ready-commit record and sends a ready vote to the coordinator. At this point, a participant has an uncertain status because the coordinator may later request the participant to abort.

The decision phase begins when the coordinator either receives votes from each participant or a timeout occurs. If a timeout occurs or at least one participant sends an

[4] Do not confuse two-phase commit with two-phase locking. The two-phase commit protocol is used only for distributed transaction processing. Two-phase locking can be used for centralized and distributed concurrency control.

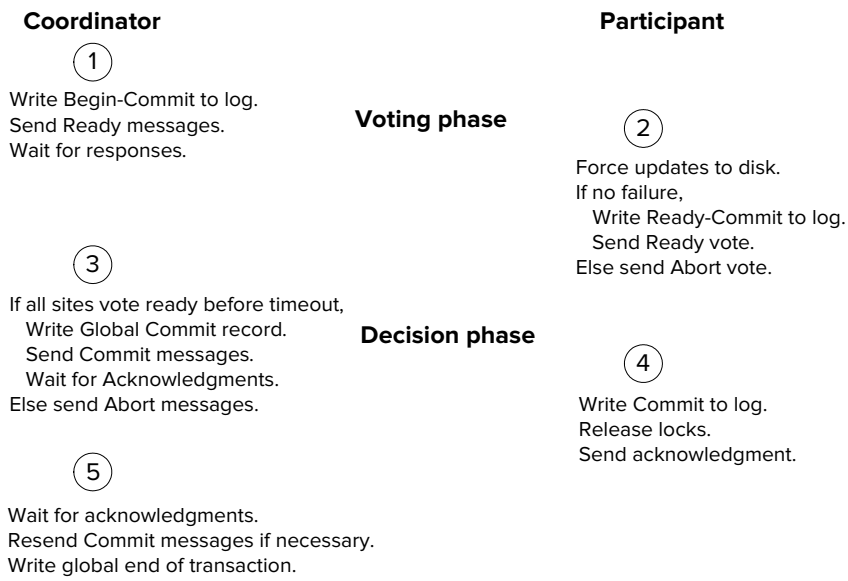


FIGURE 18.23
Two-Phase Commit Processing for Coordinator and Participants

abort vote, the coordinator aborts the entire transaction by sending abort messages to each participant. Each participant then performs a rollback of its changes.

If all participants vote ready, the coordinator writes the global commit record and asks each participant to commit its subtransaction. Each participant writes a commit record, releases locks, and sends an acknowledgment to the coordinator. When the coordinator receives acknowledgment from all participants, the coordinator writes the global end-of-transaction record. If a failure occurs in either the voting or the decision phase, the coordinator sends an abort message to all participating sites.

In practice, the two-phase commit protocol presented in Figure 18.23 is just the basic protocol. Other complications such as a failure during recovery and timeouts complicate the protocol. In addition, modifications to the basic protocol can reduce the number of messages needed to enforce the protocol. Because these extensions are beyond the scope of this book, you should consult the references at the end of the chapter for more details.

The two-phase commit protocol can use a centralized or a distributed coordinator. The trade-offs are similar to centralized versus distributed coordination for concurrency control. Centralized coordination is simpler than distributed coordination but can be less reliable.

The two-phase commit protocol does not handle any conceivable kind of failure. For example, the two-phase commit protocol may not work correctly if log records are lost. There is no known protocol guaranteeing that all sites act in unison to commit or abort in the presence of arbitrary failures. Because the two-phase commit protocol efficiently handles common kinds of failures, it is widely used in distributed transaction processing.

CAP Design Philosophy The two-phase commit protocol demonstrates limits of distributed transaction processing regarding failures. When failures partition a distributed system so that some parts of the system cannot communicate, the two-phase commit protocol cannot ensure consistency of data across sites. The two-phase commit protocol does not provide guidance about trade-offs among performance, consistency, and system availability when dealing with failures.

The CAP (Consistency, Availability, Partition tolerance) design philosophy has been developed to provide guidance about trade-offs among consistency, availability, and partition tolerance. Underlying the CAP design philosophy is the CAP theorem, devised by Professor Eric Brewer in 2000 as a conjecture and later proven by Professors Gilbert and Lynch of MIT in 2002. The CAP Theorem indicates that a distributed

database architecture for transaction processing can achieve any two of the following three objectives.

- Consistency: all copies of data items have the same value after completion of each transaction. This definition of consistency is similar to the notion of atomic for concurrency control.
- Availability: Every operation terminates with an intended response as the system is always available.
- Partition tolerance: Operations will complete even if individual components are unavailable. Only a total system failure will stop operations.

Some obvious ways to deal with the **CAP theorem** are not practical. Partition tolerance can be eliminated by placing all processing for a transaction on a single site, but scalability will be severely impacted. Availability can be eliminated so that any partition stops related processing until the partition is restored. In today's ecommerce environment, eliminating availability is not feasible.

CAP Theorem
a law about trade-offs among consistency, availability, and partition tolerance, first proposed by Eric Brewer in 2000. The law indicates that a distributed database architecture for transaction processing can achieve any two objectives but must compromise on the third objective. DBMS vendors now provide a range of alternatives to address the CAP theorem limitations with much innovation from NoSQL database products.

Typically, consistency is relaxed to preserve availability and partition tolerance. With eventual consistency, a popular type of relaxed consistency, the storage system ensures that accesses to all copies of an item will eventually return the last updated value provided that no new updates occur. The BASE concurrency approach as presented in Chapter 17, uses messages and redundant storage at each site to control the window of inconsistency.

Professor Brewer (2012) recommends a broader range of alternatives to the CAP theorem, arguing that the 2 of 3 choice is misleading. He recommends managing the partition detection interval with no trade-offs unless a partition is detected. When a partition occurs, he recommends several alternatives including recovering a partition and restoring consistency, blocking selected operations, performing compensating operations to fix inconsistencies, and using causal consistency, a variation of eventual consistency. In causal consistency, a write is guaranteed to supersede an earlier write among processes notified about writes to an item. Access by non-notified processes are subject to the normal eventual consistency rules.

CLOSING THOUGHTS

This chapter has described motivation, architectures, and services of DBMSs that support distributed processing and distributed data. Utilizing distributed processing and data can significantly improve DBMS services but at the cost of new design challenges. Distributed processing can improve scalability, interoperability, flexibility, availability, and performance while distributed data can improve data control, communication costs, and system performance. To realize these benefits, significant challenges caused by the complexity of distributed processing and data must be overcome.

Choosing an appropriate architecture is one way to manage additional complexity. This chapter described client-server architectures and parallel database architectures to utilize distributed processing. The basic tiered architectures provide alternatives among cost, complexity, and flexibility for dividing tasks among clients and servers. Specialized client-server architectures including web service architectures, cloud computing, and extreme transaction processing extend the basic tiered architectures to meet specific market demands. Parallel database processing distributes a large task among available resources. Parallel database technology of Oracle and IBM was described to indicate the implementation of the clustered disk and clustered nothing architectures. In addition, parallel processing architectures for big data were presented, complementing database parallel processing architectures.

The last part of the chapter described architectures and processing for distributed DBMSs. Architectures for distributed DBMSs differ in the integration of the local

databases. Tightly integrated systems provide both query and transaction services but require uniformity in the local DBMSs. Loosely integrated systems support data sharing among a mix of modern and legacy DBMSs. An important part of the data architecture is the level of data independence. This chapter described several levels of data independence that differ by the level of data distribution knowledge required to formulate global requests. Examples of transparency in Oracle distributed database and partitioning technologies complemented the conceptual presentation. To provide an introductory understanding to the complexity of distributed database processing, this chapter described distributed query processing and transaction processing. Both services involve complex issues not present in centralized DBMSs. The CAP design philosophy provides guidance about trade-offs among consistency, availability, and partition tolerance.

REVIEW CONCEPTS

- Motivations for client-server processing: scalability, interoperability, and flexibility
- Motivations for parallel database processing: scaleup, speedup, high availability, and predictable scalability
- Motivations for distributed data: increased local control, reduced communication costs, and improved performance
- Motivations for cloud computing: no initial licensing costs, no hosting requirements, dynamic scalability, high availability, economies of scale, and specialization
- Design issues in distributed processing: division of processing and process management
- Kinds of code to distribute between a client and a server
- Process management tasks performed by database middleware
- Differences between transaction-processing monitors, message-oriented middleware, data access middleware, and object request brokers
- Characteristics of basic client-server database architecture: two-tier, three-tier, and multiple-tier
- Web service architecture that extends multiple-tier architectures for electronic business commerce using Internet standards to achieve high interoperability
- Differences among cloud service models (Infrastructure as a Service, Platform as a Service, and Software as a Service) for cloud vendor and user organization
- Variations of cloud availability and control in cloud deployment models (public, community, private, and hybrid)
- Requirements of extreme transaction processing and typical industries with these processing requirements
- Middleware for extreme transaction processing using a write-behind cache to utilize memories from distributed servers to achieve near linear scalability
- Characteristics of parallel database architectures: clustered disk and clustered nothing
- Problems of parallel database processing: load balancing, cache coherence, and interprocessor communication
- Oracle Real Application Clusters as a commercial DBMS supporting the clustered disk architecture
- IBM DB2 with the Database Partitioning Feature as a commercial DBMS supporting the clustered nothing architecture

- Big data processing architectures supporting scalable parallel processing on large data sets with unstructured data, complementing DBMS parallel processing
- MapReduce, a parallel processing model, for batch processing of large data sets
- Apache open source projects for big data processing: Hadoop, Spark, and Hawq
- Global queries and transactions
- Role of the local database manager, the distributed database manager, and the global dictionary in the component architecture of a distributed DBMS
- Schema architectures for tightly integrated and loosely integrated distributed DBMSs
- Relationship of distributed database transparency levels and data independence
- Kinds of fragmentation: horizontal, vertical, and derived horizontal
- Complexity of fragment design and allocation
- Query formulation for fragmentation transparency, location transparency, and local mapping transparency
- Usage of database links in Oracle for providing remote data access with site autonomy
- Usage of synonyms in Oracle to hide location details
- Oracle partitioning options providing improved performance and availability with fragmentation transparency
- Performance measures and objectives for distributed query processing
- Use of two-phase locking for distributed concurrency control
- Use of the primary copy protocol to reduce update overhead with replicated data
- Additional kinds of failures in a distributed database environment
- Two-phase commit protocol for ensuring atomic commit of participant sites in a distributed transaction
- Trade-offs between centralized and distributed coordination for distributed concurrency control and recovery
- CAP design philosophy imposing trade-offs among consistency, availability, and partition tolerance

QUESTIONS

1. What is the role of clients and servers in distributed processing?
2. Briefly define the terms flexibility, interoperability, and scalability. How does client-server processing support interoperability, flexibility, and scalability?
3. Discuss some of the pitfalls of developing client-server systems.
4. Briefly define the terms scaleup and speedup and the measurement of these terms.
5. Define high availability and indicate how parallel database processing supports high availability.
6. How can a distributed database improve data control?
7. How can a distributed database reduce communication costs and improve performance?
8. Discuss some of the pitfalls when developing distributed databases.
9. Discuss why distributed processing is more mature and more widely implemented than distributed databases.
10. Why are division of processing and process management important in client-server architectures?

11. Explain how two-tier architectures address division of processing and process management.
12. Explain how three-tier architectures address division of processing and process management.
13. Explain how multiple-tier architectures address division of processing and process management.
14. What is a thin client? How does a thin client relate to division of processing in client-server architectures?
15. List some reasons for choosing a two-tier architecture.
16. List some reasons for choosing a three-tier architecture.
17. List some reasons for choosing a multiple-tier architecture.
18. What is the Web Services Architecture?
19. How does the Web Services Architecture support interoperability?
20. Briefly describe the basic architectures for parallel database processing.
21. Briefly describe the clustering extensions to the basic distributed database architectures.
22. What are the primary design issues for parallel database processing? Identify the architecture most affected by the design issues.
23. What is the cache coherence problem?
24. What is load balancing?
25. What parallel database architecture is supported by Oracle Real Application Clusters? What is a key technology in Oracle Real Application Clusters?
26. What parallel database architecture is supported by IBM DB2 DPF option? What is a key technology in the DPF?
27. What is a global request?
28. How does the integration level of the distributed DBMS affect the component architecture?
29. When is a tightly integrated distributed DBMS appropriate? When is a loosely integrated distributed DBMS appropriate?
30. Discuss the differences in the schema architecture for tightly and loosely integrated distributed DBMSs.
31. How is distributed database transparency related to data independence?
32. Is a higher level of distribution transparency always preferred? Briefly explain why or why not.
33. What is a derived horizontal fragment and why is it useful? What is the relationship of the semi-join operator and derived horizontal fragmentation?
34. What is the larger difference in query formulation: (1) fragmentation transparency to location transparency or (2) location transparency to local mapping transparency? Justify your answer.
35. Why is fragment design and allocation a complex task?
36. Why is global query optimization important?
37. What are differences between global and local optimization in distributed query processing?
38. Why are there multiple objectives in distributed query processing? Which objective seems to be more important?
39. What are the components of performance measures for distributed query processing? What factors influence how these components can be combined into a performance measure?

40. How does two phase-locking for distributed databases differ from two-phase locking for centralized databases?
41. Why is the primary copy protocol widely used?
42. What kinds of additional failures occur in a distributed database environment? How can these failures be detected?
43. What is the difference between the voting and the decision phases of the two-phase commit protocol?
44. Discuss the trade-offs between centralized and distributed coordination in distributed concurrency control and recovery.
45. What level of transparency is provided by Oracle distributed databases?
46. What are database links in Oracle distributed database processing?
47. What is the difference between a current user and connected user when using an Oracle link?
48. What is the motivation for cloud computing?
49. Briefly describe the cloud service models.
50. Briefly describe the cloud deployment models.
51. What types of DBMS products are available through cloud services?
52. What are the goals of Oracle partitioning technology?
53. Briefly describe the partitioning options available with Oracle 12c.
54. Briefly describe the requirements for extreme transaction processing and identify industries with these requirements.
55. How does the key feature of middleware for extreme transaction processing achieve near linear scalability?
56. Briefly explain the three components of the CAP Theorem.
57. What limitations does the CAP Theorem impose on transaction processing for distributed databases?
58. What are typical ways that DBMSs deal with the limitations imposed by the CAP Theorem?
59. What is the relationship of the BASE consistency principle as defined in Chapter 17 to the CAP Theorem?
60. What is Hadoop?
61. What is MapReduce? How is MapReduce related to Hadoop?
62. How do big data processing architectures complement parallel processing in DBMSs?
63. What extensions have been made to Hadoop for improved performance and tasks supported?

PROBLEMS

The problems provide practice with defining fragments, formulating queries at various transparency levels, and defining strategies for distributed query processing. The questions use the revised university database tables that follow. This database is similar to the database used in Chapter 4 except for the additional campus columns in the *Student, Offering*, and *Faculty* tables.

Student(StdNo, StdName, StdCampus, StdCity, StdState, StdZip, StdMajor, StdYear)

Course(CourseNo, CrsDesc, CrsCredits)

Offering(OfferNo, *CourseNo*, OffCampus, OffTerm, OffYear, OffLocation, OffTime, OffDays, *FacNo*)
FOREIGN KEY CourseNo REFERENCES Course
FOREIGN KEY FacNo REFERENCES Faculty

Enrollment(*OfferNo, StdNo*, EnrGrade)
FOREIGN KEY OfferNo REFERENCES Offering
FOREIGN KEY StdNo REFERENCES Student

Faculty(FacNo, FacName, FacCampus, FacDept, FacPhone, FacSalary, FacRank)

1. Write SQL SELECT statements to define two horizontal fragments as students who attend (1) the Boulder campus and (2) the Denver campus.
2. Write SQL SELECT statements to define two horizontal fragments as faculty who teach at (1) the Boulder campus and (2) the Denver campus.
3. Write SQL SELECT statements to define two horizontal fragments as offerings given at (1) the Boulder campus and (2) the Denver campus.
4. Write SQL SELECT statements to define two derived horizontal fragments as enrollments associated with offerings given at (1) the Boulder campus and (2) the Denver campus.
5. Write a SELECT statement to list the information systems courses offered in spring quarter 2017. Information systems courses contain the string "IS" in the course description. Include the course number, the description, the offer number, and the time in the result. Assume fragmentation transparency in your formulation.
6. Write a SELECT statement to list the information systems courses offered in spring quarter 2017. Information systems courses contain the string "IS" in the course description. Include the course number, the description, the offer number, and the time in the result. Assume location transparency in your formulation.
7. Write a SELECT statement to list the information systems courses offered in spring quarter 2017. Information systems courses contain the string "IS" in the course description. Include the course number, the description, the offer number, and the time in the result. Assume local mapping transparency in your formulation. The *Offering* fragments are allocated to the Boulder and the Denver sites. The *Course* table is replicated at both sites.
8. Move offering number O1 from the Boulder to the Denver campus. In addition to moving the offering between campuses, change its location to Plaza 112. Assume fragmentation transparency in your formulation.
9. Move offering number O1 from the Boulder to the Denver campus. In addition to moving the offering between campuses, change its location to Plaza 112. Assume location transparency in your formulation.
10. Move offering number O1 from the Boulder to the Denver campus. In addition to moving the offering between campuses, change its location to Plaza 112. Assume local mapping transparency in your formulation.
11. For the following query, compute communication time (*CT*) for the distributed access plans listed below. Use the formulas in section 18.6.1 and the query and network statistics (Table 18-A1) in your calculations.

```
SELECT Course.CourseNo, CrsDesc, OfferNo, OffTime, FacName
 FROM BoulderOfferings BOF, Course, DenverFaculty DF
 WHERE Course.CourseNo = BOF.Course AND OffTerm = 'Spring'
   AND OffYear = 2017 AND DF.FacNo = BOF.FacNo
   AND FacDept = 'Information Systems'
```

TABLE 18-A1
Statistics for Problem 11

Record length is 1,000 bits for each table.
32 bits for *FacNo*.
20 information systems faculty.
5,000 spring 2017 offerings
10 spring 2017 Boulder offerings taught by Denver faculty.
4,000 courses, 20,000 offerings, and 2,000 faculty in the fragments.
Course table is replicated at both the Denver and the Boulder sites.
BoulderOfferings fragment is located at the Boulder site.
DenverFaculty fragment is located at the Denver site.
Delay per message is 0.1 second.
Data rate is 100,000 bits per second.

11.1 Move the entire *DenverFaculty* fragment to the Boulder site and perform the query.

11.2 Move the entire *BoulderOfferings* fragment to the Denver site and perform the query.

11.3 Move the restricted *BoulderOfferings* fragment to the Denver site and perform the query.

11.4 Move the restricted *DenverFaculty* fragment to the Boulder site and perform the query.

11.5 Restrict the *DenverFaculty* fragment and move the join values (*FacNo*) to the Boulder site. Join the *FacNo* values with the *Course* table and the *BoulderOfferings* fragment at the Boulder site. Move the result back to the Denver site to join with the *DenverFaculty* fragment.

12. Investigate the client-server, parallel database, and distributed database features of a major DBMS. Identify the architectures used and important product features.

REFERENCES FOR FURTHER STUDY

This chapter, although providing a broad coverage of distributed processing and data, has only covered the basics. Specialized books on distributed database management include the classic by Ceri and Pelagatti (1984) and the more recent book by Ozsu and Valduriez (1991). The book by Ceri and Pelagatti has a well-written chapter on distributed database design. Date (1990) presents 12 objectives for distributed systems with the most emphasis on distributed DBMSs. Bernstein (1996) provides a detailed presentation of the role of middleware in client-server architectures. Brewer (2012) describes the evolution of the CAP design philosophy over a 12 year period. Dean and Ghemawat (2004) describe the MapReduce programming model, first deployed at Google. Chang et al. (2014) describe the SQL engine in Apache Hawq.

chapter

19

DBMS Extensions for Object and NoSQL Databases

Learning Objectives

This chapter describes extensions to DBMSs for alternative database representations, a richer object database representation and a simpler NoSQL database representation. After this chapter, the student should have acquired the following knowledge and skills:

- Explain business reasons for using object database technology
- Understand features in SQL:2016 for defining and manipulating user-defined types, typed tables, and subtable families
- Write and document Oracle SQL statements for user-defined types and typed tables
- Explain business reasons for using NoSQL technology
- Provide simple examples to depict data models used in NoSQL databases
- Write and validate documents conforming to the JavaScript Object Notation (JSON)
- Write and document Couchbase N1QL statements for manipulating JSON databases

OVERVIEW

Chapter 18 described ways to utilize remote processing capabilities and computer networks for client-server processing, parallel database processing, and distributed databases. An increasing amount of distributed database processing involves data representations that DBMSs have not traditionally supported. Two opposite approaches have emerged to extend database technology for non-traditional data and operations. Object database technology supports a richer data representation for new kinds of data and operations. NoSQL database technology supports a simplified, flexible representation for big data processing, both batch and online. In this chapter, you will learn about extensions to DBMSs for both approaches.

The first part of this chapter provides a broad introduction to object relational DBMSs. You will first learn about the business reasons to extend database technology for objects. This chapter discusses the increasing usage of both traditional and complex data in business applications and the mismatch between DBMSs and programming languages as the driving forces behind object database technology. After grasping the motivation, you are ready to learn about object technology and its impact on DBMSs. The second and third parts of this chapter present object support in SQL:2016, the standard for object-relational DBMSs and Oracle,

a significant implementation of the SQL:2016 standard. You will learn about data definition and data manipulation features for object-relational databases.

The remaining parts of this chapter cover NoSQL database technology, a more recent development than object database technology. You will first learn about the business reasons for using NoSQL databases. After understanding the business case for NoSQL databases, you will learn about alternative data models in NoSQL databases. To provide practice with a prominent NoSQL DBMS, the last part of this chapter covers the N1QL query language provided by Couchbase. You will learn about N1QL statements for data definition and manipulation of document databases conforming to the de facto standard, JavaScript Object Notation (JSON).

19.1 MOTIVATION FOR OBJECT DATABASE MANAGEMENT

This section discusses two forces driving the demand for object database management. After a discussion of these forces, this section shows example applications to depict business needs for object database management.

19.1.1 Complex Data

Most relational DBMSs support only a small number of traditional data types. Built-in data types supported in SQL include whole numbers, real numbers, fixed-precision numbers like currency representation, dates, times, logical (true/false) values, and text. These data types are sufficient for many business applications, or at least major parts of many applications. Many business databases contain columns for names, prices, addresses, and transaction dates that readily conform to standard data types.

Hardware and software advances support capture and manipulation of complex, unstructured data in a digital format. Almost any kind of complex data including images, audio, video, maps, and three-dimensional graphics can be digitally stored. For example, an image can be represented as a two-dimensional array of pixels (picture elements) in which each pixel contains a numeric property representing its color or shade of gray. Digital storage provides lower costs and higher reliability than traditional storage such as paper, film, or slides. In addition, digital storage allows easier retrieval and manipulation. For example, a medical professional can retrieve digital images by content and similarity to other images. An image editor can manipulate digital images with operations such as cropping, texturing, and color tuning.

The demand for object database technology does not come only from the ability to store and manipulate complex. Rather, the need to store large amounts of complex data and integrate complex data with simple data drives the demand for object database technology. Many business applications require large amounts of complex data. For example, insurance claims processing and medical records can involve large amounts of image data. Storing images in separate files becomes tedious for a large collection of images.

The ability to use standard and complex data together is increasingly important in many business applications. For example, to review a patient's condition, a physician may request X-rays along with vital statistics. Without integration, two separate applications are required to display the data: an image editor to display the X-rays and a DBMS to retrieve the vital statistics. The ability to retrieve both images and vital statistics in a single query is a large improvement. Besides retrieving complex data, new operations also may be necessary. For example, a physician may want to compare the similarity of a patient's X-rays with X-rays that show abnormalities.

19.1.2 Type System Mismatch

Increasingly, software written in a procedural language needs to access a database. Procedural languages support customized interfaces for data entry forms and reports,

operations beyond the capability of SQL, data intensive Web applications, and batch processing. For example, writing procedural code in a programming language is often necessary to develop consumer-oriented applications for electronic commerce. Embedded SQL is often used to access a database from within a web page or code page associated with a web page. After executing an embedded SQL statement, database columns are stored in program variables that can be manipulated further.

A mismatch between the data types in a relational DBMS and the data types of a programming language makes software more complex and difficult to develop. For example, payroll processing can involve many kinds of benefits, deductions, and compensation arrangements. A relational database may have one representation for benefits, deductions, and compensation arrangements while a programming language may have a rather different representation. Before coding complex calculations, data must be transformed from the relational database representation (tables) into the programming language representation (records, objects, and arrays). After the calculations, the data must be transformed back into the relational database representation.

Complex data exacerbates data type mismatch. Programming languages usually have richer data type systems than DBMSs. For example, relational databases provide a tedious representation of geometric shapes in a building's floor plan. Objects such as points, lines, and polygons may be represented as one text column or as several numeric columns such as X and Y coordinates for points. In contrast, a programming language may have custom data types for points, lines, and polygons. There may be considerable coding to transform between the relational database representation and the programming language representation.

In addition, a relational DBMS cannot perform elementary operations on complex data. Thus, a computer program must be written instead of using a query language. For example, a complex program must be written to compute the similarity between two images. The program will probably contain 10 to 100 times the amount of code found in a query. In addition, the program must transform between the database and the programming language representations.

19.1.3 Application Examples

This section depicts several applications that involve a mix of simple and complex data as well as ad hoc queries. These applications have features increasingly found in many business applications. As you will see, object DBMSs support requirements of these kinds of applications.

Mapping Websites and GPS Devices Mapping websites and Global Positioning System (GPS) devices are the most prominent applications involving a mix of simple and complex data. Millions of individuals use mapping websites and GPS devices every day. Mapping websites provide several different kinds of maps (street, aerial, and hybrid), complex data types involving coordinates, graphics, and text. The major services provided by a mapping website are maps and directions, queries combining complex data (maps) and simple data (addresses, locations, and directions). Directions involve the fundamental operation of shortest distance calculation between two points on a map. Mapping websites and GPS devices provide additional services including points of interest, gas price locators, and traffic conditions. These additional services involve queries combining maps along with traditional business data. GPS devices use voice, another complex data type, to guide individuals to specified locations.

Dental Office Support Dental offices use a mix of simple and complex data to make appointments, conduct examinations, and generate bills. Setting appointments requires a calendar with time blocks for service providers (dentists and hygienists). In conducting examinations, service providers use dental charts (graphic of mouth with each tooth identified), X-rays, patient facts, and dental histories. After an examination,

bill preparation uses the list of services in the examination and patient insurance data. Queries involving both simple and complex data are showing a dental chart with recent dental problems highlighted and comparing X-rays for symptoms of gum loss.

Real Estate Listing Service Real estate listing services increasingly use complex data to facilitate customer searches. A real estate listing includes a mix of simple and complex data. The simple data include numerous facts about homes such as the number of bedrooms, the square feet, and the listing price. Complex data include photographs of homes, floor plans, video tours, and area maps. Queries can involve a mix of simple and complex data. Customers want to see homes in a specified neighborhood with selected features. Some customers may want to see homes with the closest match to a set of ideal features in which a customer assigns a weight to each feature. After selecting a set of homes, a customer wants to explore the appearance, floor plan, and facts about the homes.

Auto Insurance Claims Auto insurance companies use complex data to settle claims. Analyzing claims involves complex data such as photographs, street maps, accident reports, and witness statements as well as simple data such as driver and vehicle descriptions. Settling claims involves a map showing service providers as well as service provider rates and service history. Queries for accident data and a list of service providers in close proximity to a customer involve both simple and complex data.

19.2 OBJECT DATABASE FEATURES IN SQL:2016

As a response to these needs, commercial firms and university research teams developed object database technology. These efforts led to a variety of commercial approaches to support object database technology and a major extension to the SQL standard. Over time, the SQL standard has eclipsed other approaches for object features in enterprise DBMSs. Enterprise DBMS vendors have implemented substantial parts of the object database features in the SQL standard. Collectively, the object database features have changed relational databases to object-relational databases.

User-defined data types are the most salient part of the object database features in the SQL standard. Almost any kind of complex data can be added as a user-defined type. Image data, spatial data, time series, and video are just a few of the possible data types. Major DBMS vendors provide a collection of prebuilt user-defined types and the ability to extend the prebuilt types as well as to create new user-defined types. Table 19-1 lists pre-built data types supported by major enterprise DBMSs. For each user-defined type, a collection of functions can be defined and used in SQL statements. For prebuilt types, specialized storage structures have been created to improve performance. For example, multidimensional Btrees can be provided for accessing spatial data.

Although user-defined types are the most salient feature of object-relational databases, the SQL standard contains other prominent features. Enterprise DBMS vendors have implemented some of these object features including subtable families (generalization hierarchies for tables), arrays, and the reference and row data types.

To clarify object database concepts, this section presents examples using the SQL:2016 syntax for object-relational databases. The examples demonstrate SQL:2016 features for user-defined data types, table definitions with typed tables, subtable families, and usage of typed tables. All examples were checked for correct syntax using the

TABLE 19-1
Pre-Built User-Defined Data Types in Object-Relational DBMSs

Product	User-Defined Types
IBM DB2 Extenders	Audio, Image, Video, XML, Spatial, Geodetic, Text, Net Search
Oracle Data Cartridges	Text, Video, Image, Spatial, XML, Medical Images, RFID
Informix Data Blades	Text, Spatial, Geodetic, Web, Time Series, Binary, Hierarchical Node

Mimer SQL-2003 validator (developer.mimer.com/validator/index.htm). Section 19.3 presents the object features in Oracle, a significant implementation of the SQL:2016 standard.

19.2.1 User-Defined Types

The user-defined type, the most fundamental object feature in SQL:2016, allows bundling of data and procedures. User-defined types support the definition of new structured data types as well as the refinement of the standard data types. A structured data type has a collection of attributes and methods. In SQL-92, the CREATE DOMAIN statement supports refinements to standard data types but not the definition of new structured types. A method is the object-oriented term for a procedure or function associated with an object.

Example 19.1 shows the *Point* type to depict the basic syntax of user-defined types. The first part of a user-defined type contains the attribute definitions. The double hyphen denotes a comment. For methods, the first parameter is implicit meaning that its specification is not needed. For example, the *Distance* method lists only one *Point* parameter (P2) because the other *Point* parameter is implicit. In SQL:2016, methods only use input parameters and should return values. The body of methods is not shown in the CREATE TYPE statement but rather in the CREATE METHOD statement.

Example 19.1

Point type in SQL:2016

```
CREATE TYPE Point AS
( X FLOAT,    -- X coordinate
  Y FLOAT )   -- Y coordinate
 METHOD Distance(P2 Point) RETURNS FLOAT,
  -- Computes the distance between 2 points
 METHOD Equals (P2 Point) RETURNS BOOLEAN
  -- Determines if 2 points are equivalent
;
```

SQL:2016 methods are somewhat limited in that they must return single values and only use input parameters. In addition, the first parameter of a method is implicitly an instance of the type in which it is associated. SQL:2016 provides functions and procedures that do not have the restrictions of methods. Because functions and procedures are not associated with a specific type, SQL:2016 provides separate definition statements (CREATE FUNCTION and CREATE PROCEDURE). Procedures can use input, output, and input-output parameters whereas functions only use input parameters.

Example 19.2 shows the *ColorPoint* type, a subtype of *Point*. The UNDER keyword indicates the parent type. Because SQL:2016 does not support multiple inheritance, only a single type name can follow the UNDER keyword. In the method definitions, the OVERRIDING keyword indicates that the method overrides the definition in a parent type.

Besides the explicit methods listed in the CREATE TYPE statement, user-defined types have implicit methods that can be used in SQL statements and stored procedures, as listed below:

- Constructor method: creates an empty instance of the type. The constructor method has the same name as the data type. For example, `Point()` is the constructor method for the *Point* type.
- Observer methods: retrieve values from attributes. Each observer method uses the same name as its associated attribute. For example, `X()` is the observer method for the *X* attribute of the *Point* type.

Example 19.2

ColorPoint type

```
CREATE TYPE ColorPoint UNDER Point AS
(Color INTEGER )
 METHOD Brighten (Intensity INTEGER) RETURNS INTEGER,
  -- Increases color intensity
 OVERRIDING METHOD Equals (CP2 ColorPoint)
    RETURNS BOOLEAN
  -- Determines equivalence of 2 ColorPoint objects
;
```

- Mutator methods: change values stored in attributes. Each mutator method uses the same name as its associated attribute with one parameter for the value. For example, `X(45.0)` changes the value of the *X* attribute.

SQL:2016 features the ARRAY and MULTISET collection types to support types with more than one value such as time-series and geometric shapes. Arrays support bounded ordered collections, while multisets support unbounded, unordered collections. Example 19.3a defines a triangle type with an array of three points. The number following the ARRAY keyword indicates the maximum size of the array. Example 19.3b defines a polygon with a multiset of points. You should observe that maximum length cannot be specified for MULTISET attributes.

Example 19.3a

Triangle Type Using an ARRAY type

```
CREATE TYPE Triangle AS
 (Corners Point ARRAY[3], -- Array of Corner Points
  Color INTEGER )
 METHOD Area() RETURNS FLOAT,
  -- Computes the area
 METHOD Scale (Factor FLOAT) RETURNS Triangle
  -- Computes a new triangle scaled by Factor
;
```

Example 19.3b

Polygon Type using a MULTISET type

```
CREATE TYPE Polygon AS
 (Corners Point MULTISET, Color INTEGER )
 METHOD Area() RETURNS FLOAT,
  -- Computes the area
 METHOD Scale (Factor FLOAT) RETURNS Polygon
  -- Computes a new polygon scaled by Factor
;
```

User-defined types are integrated into the heart of SQL:2016. User-defined types can be used as data types for columns in tables, passed as parameters, and returned as values. User-defined methods can be used in expressions in the SELECT, WHERE, and HAVING clauses.

19.2.2 Table Definitions

The examples in the remainder of Chapter 19 are based on a simple property database with properties and agents as depicted by the ERD in Figure 19.1. For the presentation here, the most important aspect of the ERD is the generalization hierarchy for properties. SQL:2016 provides direct support of generalization hierarchies rather than indirect support as indicated by the generalization hierarchy conversion rule presented in Chapter 6.

SQL:2016 supports two styles of table definitions. The traditional SQL-92 style uses foreign keys to link two tables. Example 19.4 depicts the *Property* table using a foreign key to reference the agent representing the property. The *PView* column uses the binary large object (BLOB) type. As an alternative to a BLOB data type for images, some DBMSs provide prebuilt user-defined types for prominent image formats.

Example 19.4

Agent and *Property* tables using the traditional SQL-92 style

```
CREATE TABLE Agent
(AgentNo INTEGER,
 AName    VARCHAR(30),
 Street   VARCHAR(50),
 City     VARCHAR(30),
 State    CHAR(2),
 Zip      CHAR(9),
 Phone    CHAR(13),
 Email    VARCHAR(50),
 CONSTRAINT AgentPK PRIMARY KEY(AgentNo) );

CREATE TABLE Property
(PropNo  INTEGER,
 Street  VARCHAR(50),
 City    VARCHAR(30),
 State   CHAR(2),
 Zip     CHAR(9),
 SqFt    INTEGER,
 PView   BLOB,
 AgentNo INTEGER,
 Location Point,
 CONSTRAINT PropertyPK PRIMARY KEY(PropNo),
 CONSTRAINT AgentFK FOREIGN KEY(AgentNo) REFERENCES Agent );
```

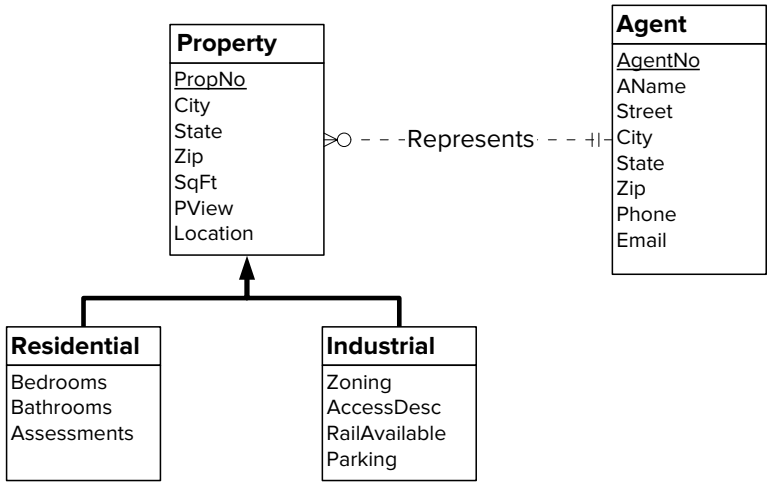


FIGURE 19.1
ERD for the Property Database

SQL:2016 supports the row type constructor to allow rows of a table to be stored as variables, used as parameters, and returned by functions. A row type is a sequence of name/value pairs. One use of a row type is to collect related columns together so that they can be stored as a variable or passed as a parameter. Example 19.5 depicts the *Property* table using a row type for the address columns (*Street, City, State*, and *Zip*).

Example 19.5

Revised *Property* table definition with a ROW type

```
CREATE TABLE Property
(PropNo  INTEGER,
 Address  ROW (Street  VARCHAR(50),
               City    VARCHAR(30),
               State   CHAR(2),
               Zip     CHAR(9) ),
 SqFt     INTEGER,
 PView    BLOB,
 AgentNo  INTEGER,
 Location Point,
 CONSTRAINT PropertyPK PRIMARY KEY(PropNo),
 CONSTRAINT AgentFK FOREIGN KEY(AgentNo) REFERENCES Agent );
```

SQL:2016 provides an alternative style of table definition known as typed tables to support object identifiers and object references. With typed tables, a table definition references a user-defined type rather than providing its own list of columns. Example 19.6 depicts the *AgentType* user-defined type and the *Agent* table referring to *AgentType*. In addition, the *AddressType* (a named structured type) is used in place of the unnamed ROW type in Example 19.5. The REF clause defines an object identifier for the table. The SYSTEM GENERATED keywords indicate that the object identifiers are generated by the DBMS, not the user application (USER GENERATED keywords).

Example 19.6

Definition of *AddressType* and *AgentType* followed by the typed *Agent* table based on *AgentType*

```
CREATE TYPE AddressType AS
 (Street  VARCHAR(50),
  City    VARCHAR(30),
  State   CHAR(2),
  Zip     CHAR(9) );
CREATE TYPE AgentType AS
(AgentNo INTEGER,
 AName   VARCHAR(30),
 Address AddressType,
 Phone   CHAR(13),
 Email   VARCHAR(50) );

CREATE TABLE Agent OF AgentType
(REF IS AgentOId SYSTEM GENERATED,
 CONSTRAINT AgentPK PRIMARY KEY(AgentNo) );
```

Other tables can reference tables based on user-defined types. Object references provide an alternative to value references of foreign keys. Example 19.7 depicts a definition of the *PropertyType* type with a reference to the *AgentType*. The SCOPE clause limits a reference to the rows of a table rather than objects of the type.

Example 19.7

Definition of *PropertyType* and the typed *Property* table

```
CREATE TYPE PropertyType AS
(PropNo   INTEGER,
 Address  AddressType,
 SqFt      INTEGER,
 PView     BLOB,
 Location Point,
 AgentRef REF(AgentType) SCOPE Agent );

CREATE TABLE Property OF PropertyType
(REF IS PropertyOId SYSTEM GENERATED,
 CONSTRAINT PropertyPK PRIMARY KEY(PropNo) );
```

As these examples demonstrate, SQL:2016 provides a variety of ways to define tables (typed versus untyped tables, references versus traditional foreign keys, ROW types versus columns versus named structured types). For productivity of application developers, consistent usage of table definition styles is important. A reasonable rule of thumb is to use either the traditional table definitions (untyped tables with unstructured columns and foreign keys) or typed tables with named structured types and reference types. Mixing table definition styles can burden application programmers because the definition style influences the coding used in retrieval and manipulation statements.

SQL:2016 supports nested tables using the MULTISET type with the ROW type for elements of a multiset. Nested tables are useful at the application level especially for complex business rules involving stored procedures. In addition, nested tables can be useful to reduce the type system mismatch between a DBMS and a programming language as discussed in section 19.1.2. At the table design level, the usage of nested tables is not clear for business databases. Although some design theory exists for nested tables, the theory is not widely known or practiced. Because of the immaturity of nested table practice, no examples of nested tables are presented for table design.

19.2.3 Subtable Families

Inheritance applies to tables in a similar way as it applies to user-defined types. A table can be declared as a subtable of another table. A subtable inherits the columns of its parent tables. SQL:2016 limits inheritance for tables to single inheritance. A potentially confusing part of table inheritance involves type inheritance. Tables involved in subtable relationships must be typed tables with the associated types also participating in subtype relationships as demonstrated in Example 19.8. Note that the REF clauses and primary key constraints are inherited from the *Property* table so they are not specified for the *Residential* and the *Industrial* tables.

Set inclusion determines the relationship of rows of a parent table to rows in its subtables. Every row in a subtable is also a row in each of its ancestor (direct parents and indirect parents) tables. Each row of a parent table corresponds to at most one row in direct subtables. This set inclusion relationship extends to an entire subtable family, including the root table and all subtables directly or indirectly under the root table. For

Example 19.8

Subtypes and subtables for *Residential* and *Industrial* properties

```
CREATE TYPE ResidentialType UNDER PropertyType
(BedRooms     INTEGER,
 BathRooms    INTEGER,
 Assessments DECIMAL(9,2) ARRAY[6] );

CREATE TABLE Residential OF ResidentialType UNDER Property;

CREATE TYPE IndustrialType UNDER PropertyType
(Zoning            VARCHAR(20),
 AccessDesc        VARCHAR(20),
 RailAvailable     BOOLEAN,
 Parking           VARCHAR(10) );

CREATE TABLE Industrial OF IndustrialType UNDER Property;
```

example, a subtable family includes security as the root, bond and stock under investment, and corporate, municipal, and federal under bond. The root of a subtable family is known as the maximal table. Security is the maximal table in this example.

Data manipulation operations on a row in a subtable family affect related rows in parent tables and subtables. The following is a brief description of side effects when manipulating rows in subtable families.

- If a row is inserted into a subtable, then a corresponding row (with the same values for inherited columns) is inserted into each ancestor table. The insert cascades upward in the subtable family until it reaches the maximal table.
- If a column is updated in a parent table, then the column is also updated in all direct and indirect subtables that inherit the column.
- If an inherited column is updated in a subtable, then the column is changed in the corresponding rows of direct and indirect parent tables. The update cascade stops in the parent table in which the column is defined, not inherited.
- If a row in a subtable family is deleted, then every corresponding row in both parent and subtables is also deleted.

19.4.4 Manipulating Complex Objects and Subtable Families

The richer data definition capabilities of SQL:2016 lead to new features when using row type columns and reference type columns in data manipulation and data retrieval statements. When inserting data into a table with a row type column, the keyword ROW must be used as demonstrated in Example 19.9. If a column uses a user-defined type instead of the ROW type, the type name must be used as depicted in Example 19.10.

When inserting data into a table with a reference type column, the object identifier can be obtained with a SELECT statement. If object identifiers for a referenced table are user-generated (such as primary key values), a SELECT statement may not be necessary. Even with user-generated object identifiers, a SELECT statement may be necessary if the object identifier is not known when the row is inserted. Example 19.11 demonstrates a SELECT statement to retrieve the object identifier (*AgentOID*) from the related row of the *Agent* table. In the SELECT statement, the other values to insert are constant values. For the *Assessments* column, the constant value is an array of values denoted by the ARRAY keyword along with the square brackets that surround the array element values.

Example 19.9

Using the ROW keyword when inserting a row in a table with a ROW type column. This example assumes that the *Address* column of the *AgentType* type was defined with the ROW type

```
INSERT INTO Agent
(AgentNo, AName, Address, Email, Phone)
VALUES (999999, 'Sue Smith',
         ROW('123 Any Street', 'Denver', 'CO', '80217'),
         'sue.smith@anyisp.com', '13031234567');
```

Example 19.10

Using the type name when inserting two rows in a table with a structured type column. This example corresponds to the *AgentType* type defined in Example 19.6

```
INSERT INTO Agent
(AgentNo, AName, Address, Email, Phone)
VALUES (999999, 'Sue Smith',
         AddressType('123 Any Street', 'Denver', 'CO', '80217'),
         'sue.smith@anyisp.com', '13031234567');

INSERT INTO Agent
(AgentNo, AName, Address, Email, Phone)
VALUES (999998, 'John Smith',
         AddressType('123 Big Street', 'Boulder', 'CO', '80217'),
         'john.smith@anyisp.com', '13034567123');
```

Example 19.11

Using a SELECT statement to retrieve the object identifier of the related *Agent* row

```
INSERT INTO Residential
(PropNo, Address, SqFt, AgentRef, BedRooms, BathRooms, Assessments)
SELECT 999999, AddressType('123 Any Street', 'Denver', 'CO', '80217'),
        2000, AgentOID, 3, 2, ARRAY[190000, 200000]
FROM Agent
WHERE AgentNo = 999999;
```

Example 19.11 also demonstrates several aspects about subtypes and subtables. First, the INSERT statement can reference the columns in both types because of the subtype relationship involving *ResidentialType* and *PropertyType*. Second, inserting a row into the *Residential* table automatically inserts a row into the *Property* table because of the subtable relationship between the *Residential* and *Property* tables.

Reference columns can be updated using a SELECT statement in a manner similar to that used in Example 19.11. Example 19.12 demonstrates an UPDATE statement using a SELECT statement to retrieve the object identifier of the related agent.

Example 19.12

Using a SELECT statement to retrieve the object identifier of the related *Agent* row

```
UPDATE Residential
  SET AgentRef =
   ( SELECT AgentOID FROM Agent WHERE AgentNo = 999998 )
  WHERE PropNo = 999999;
```

Path expressions using the dot operator and the dereference operator provide an alternative to the traditional value-based joins in SQL-92. Example 19.13 depicts the use of the dot and the dereference operators in path expressions. For columns with a row or user-defined type, you should use the dot operator in path expressions. The expression `Address.City` references the city component of the *Address* row column. For columns with a reference type, you should use the dereference operator (–>) in path expressions. The expression, `AgentRef->Name`, retrieves the *Name* column of the related *Agent* row. The dereference operator (–>) must be used instead of the dot operator because the column *AgentRef* has the type `REF(AgentType)`. The distinction between the dot and dereference operators is one of the more confusing aspects of SQL:2016. Other object-oriented languages such as Java do not have this distinction.

Example 19.13

SELECT statement with path expressions and the dereference operator

```
SELECT PropNo, P.Address.City, P.AgentRef->Address.City
 FROM Property P
 WHERE AgentRef->AName = 'John Smith'
```

Sometimes there is a need to test membership in a specific table without being a member of other subtables. Example 19.14 retrieves residential properties where the square feet column is greater than 3,000. The FROM clause restricts the scope to rows whose most specific type is the *ResidentialType*. Thus, Example 19.14 does not retrieve any rows of the *Industrial* table, a subtable of the *Property* table.

Example 19.14

Using ONLY to restrict the range of a table in a subtable family

```
SELECT PropNo, Address, Location
 FROM ONLY (Residential)
 WHERE Sqft > 1500
```

19.3 OBJECT DATABASE FEATURES IN ORACLE

The most widely implemented part of the SQL:2016 object packages is the user-defined type. The major relational DBMS vendors including IBM and Oracle have implemented user-defined types that provide similar features as the SQL:2016 standard. User-defined types are important for storage and manipulation of complex data in business databases.

Beyond user-defined types, the object features in the SQL:2016 standard have gained some commercial acceptance. As an example of commercial implementation of the object features in SQL:2016, this section presents the most important object features of Oracle 12c[1] using the examples from the previous section. Although Oracle 12c does not claim complete conformance with the object features, it supports most of the features of the SQL:2016 object packages as well as some additional object features. Even if you do not use Oracle 12c, you can gain insight about the complexity of the SQL:2016 object features and the difficulty of ensuring enhanced conformance with the SQL:2016 standard.

19.3.1 Defining User-Defined Types and Typed Tables in Oracle

Oracle supports user-defined types with a syntax close to the SQL:2016 syntax. As depicted in Example 19.15, most of the differences are cosmetic such as the different placement of the parentheses, the reserved word RETURN instead of RETURNS in SQL:2016, the reserved word OBJECT for root-level types, and the keywords FINAL and INSTANTIABLE. The keywords NOT FINAL mean that subtypes can be defined. The keyword INSTANTIABLE[2] means that instances of the type can be created. Differences in methods are more significant. Oracle supports functions and procedures as methods compared to only method functions in SQL:2016. Thus, the *Print* procedure in Example 19.15 is not used in Example 19.1 because SQL:2016 does not support method procedures. In addition, Oracle supports order methods for direct object to object comparisons, map methods for indirect object comparisons, and static methods for global operations that do not need to reference the data of an object instance.

Before methods can be used, a body for each method must be created in a CREATE TYPE BODY statement. Example 19.16 contains an implementation for each method in the *Point* type definition. Oracle uses the keyword SELF to refer to the implicit parameter for a method. However, the usage of SELF is optional as shown in each of the method bodies. Example 19.17 demonstrates usage of the methods of the *Point* type. Before demonstrating method usage, a table of points is created and points are inserted into the table.

[1] For the material presented in this section, Oracle 12c does not provide new object relational features beyond Oracle 11.2g.

[2] The keywords NOT INSTANTIABLE mean that the type is abstract without instances. Abstract types contain data and code but no instances. Abstract types have been found to enhance code sharing in object-oriented programming.

Example 19.15

Point type in Oracle (corresponds to Example 19.1 for SQL:2016)

```
CREATE TYPE Point AS OBJECT
( x FLOAT(15),
  y FLOAT(15),
 MEMBER FUNCTION Distance(P2 Point) RETURN NUMBER,
  -- Distance between implicit point and P2 point parameters
 MEMBER FUNCTION Equals (P2 Point) RETURN BOOLEAN,
  -- Determines if P2 and implicit parameter are equivalent
 MEMBER PROCEDURE Print )
NOT FINAL
INSTANTIABLE;
```

Example 19.16

CREATE TYPE BODY statement for *Point* type in Oracle

```
CREATE TYPE BODY Point AS
  MEMBER FUNCTION Distance(P2 Point) RETURN NUMBER IS
  BEGIN
    RETURN sqrt(power(x - P2.x,2) + power(y - P2.y,2));
 -- Equivalent to previous line using SELF
 -- RETURN sqrt(power(SELF.x - P2.x,2) + power(SELF.y - P2.y,2));
  END;
  MEMBER FUNCTION Equals(P2 Point) RETURN BOOLEAN IS
  BEGIN
    IF x = P2.x AND y = P2.y THEN
      RETURN TRUE;
    ELSE
      RETURN FALSE;
    END IF;
  END;
  MEMBER PROCEDURE Print IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('x: ' || to_char(x) || ' - '  || 'y: ' ||
      to_char(y));
  END;
 END;
```

Example 19.17

Script to create points and use point methods

```
CREATE TABLE PointTbl of Point;

INSERT INTO PointTbl VALUES(10, 10);
INSERT INTO PointTbl VALUES(3, 4);
SELECT * FROM PointTbl;

SET SERVER OUTPUT ON;
-- Anonymous block to use point methods
```

```
DECLARE
  P1 Point;
  P2 Point;
BEGIN
  SELECT VALUE(p) INTO P1 FROM PointTbl p WHERE p.x = 10;
  P1.Print();
  SELECT VALUE(p) INTO P2 FROM PointTbl p WHERE p.x = 3;
  P2.Print();
  DBMS_OUTPUT.PUT_LINE('Distance: ' || to_char(P1.Distance(P2)));
  IF P1.Equals(P2) THEN
     DBMS_OUTPUT.PUT_LINE('Same Point');
  ELSE
     DBMS_OUTPUT.PUT_LINE('Different Point');
  END IF;
END;
-- PointTbl is not used in the remainder of the examples.
DROP TABLE PointTbl;[3]
```

Oracle supports inheritance for user-defined types similar to SQL:2016. An important difference involves the overriding of methods. In Oracle, overriding methods have the same name and signature in the parent type and subtype. A signature consists of the method's name and the number, data types, and order of the parameters. If two methods have different signatures, there is no overriding as both methods exist in the subtype. As in SQL:2016, the OVERRIDING keyword should be used when overriding a method. In Example 19.18, there is no overriding as the *Equals* method in *ColorPoint* has a different signature than the *Equals* method in *Point*. The *Equals* method in *ColorPoint* uses a *ColorPoint* argument whereas the *Equals* method in *Point* uses a *Point* argument. However, the *Print* method in *ColorPoint* overrides the *Print* method in *Point* as both methods have the same signature.

Example 19.18

ColorPoint type in Oracle (corresponds to Example 19.2 for SQL:2016)

```
CREATE TYPE ColorPoint UNDER Point
(Color INTEGER,
 MEMBER FUNCTION Brighten (Intensity INTEGER) RETURN INTEGER,
  -- Increases color intensity
 MEMBER FUNCTION Equals (CP2 ColorPoint) RETURN BOOLEAN,
  -- Determines if 2 ColorPoints are equivalent
  -- No overriding: two Equals methods have different signatures.
OVERRIDING MEMBER PROCEDURE Print )
NOT FINAL
INSTANTIABLE;
```

Oracle supports row types and typed tables similarly to SQL:2016, as depicted in Example 19.19. Like SQL:2016, Oracle supports the ROW type and user-defined types for structuring subsets of columns. For example, in *AgentType*, the address attribute could use the ROW type instead of the user-defined *AddressType*. For the CREATE TABLE statement, Oracle specifies object identifiers differently than SQL:2016. In Oracle, the OBJECT IDENTIFIER clause defines an object identifier as system-generated

[3] The recyclebin must be purged (`purge recyclebin`) before creating the *ColorPoint* type in Example 19.18. The text file with the examples for Chapter 19 contains the purge statement.

or user-generated. System-generated object identifiers do not have a name as SQL:2016 requires. However, Oracle provides functions to manipulate system-generated object identifiers so a column name is not necessary.

Example 19.19

Oracle definition of *AddressType* and *AgentType* followed by the typed *Agent* table based on *AgentType* (corresponds to Example 19.6 for SQL:2016)

```
CREATE TYPE AddressType AS OBJECT
( Street  VARCHAR(50),
  City    VARCHAR(30),
  State   CHAR(2),
  Zip     CHAR(9) )
NOT FINAL;

CREATE TYPE AgentType AS OBJECT
(AgentNo INTEGER,
 AName   VARCHAR(30),
 Address AddressType,
 Phone   CHAR(13),
 Email   VARCHAR(50) )
 NOT FINAL;

CREATE TABLE Agent OF AgentType
( CONSTRAINT AgentPK PRIMARY KEY(AgentNo) )
  OBJECT IDENTIFIER IS SYSTEM GENERATED ;
```

Oracle supports reference types for columns similar to SQL:2016 as depicted in Example 19.20. The usage of the SCOPE clause is somewhat different in Oracle, however. In Oracle, the SCOPE clause cannot be used in a user-defined type as it can in SQL:2016[4]. To compensate, you can define a referential integrity constraint to limit the scope of a reference as shown for the *Property* table in Example 19.20.

Example 19.21 shows the user-defined types for residential and industrial properties along with the table definitions. The constraint and object identifier clauses are repeated in the *Residential* and *Industrial* tables because Oracle does not support subtables.

Example 19.21 also shows differences between the declaration of array columns in Oracle and SQL:2016. In Oracle, the VARRAY constructor cannot be used directly with columns of a table or attributes of a user-defined type. Instead, the VARRAY constructor must be used in a separate user-defined type as shown in the *AssessType* type in Example 19.21. In addition, Oracle uses parentheses for the array size instead of the square brackets used in SQL:2016.

19.3.2 Using Typed Tables in Oracle

We begin this section with manipulation statements to insert and modify objects in the typed tables. Example 19.22 demonstrates an INSERT statement using a type name

[4] The SCOPE clause can be used in a column definition of a CREATE TABLE statement. However, the table is no longer a typed table when using the SCOPE clause as part of a column definition.

Example 19.20

Oracle definition of *PropertyType* with a reference to *AgentType* and the typed *Property* table (corresponds to Example 19.7 for SQL:2016)

```
CREATE TYPE PropertyType AS OBJECT
(PropNo   INTEGER,
 Address  AddressType,
 SqFt     INTEGER,
 AgentRef REF AgentType,
 Location Point )
 NOT FINAL
 INSTANTIABLE;

CREATE TABLE Property OF PropertyType
 ( CONSTRAINT PropertyPK PRIMARY KEY(PropNo),
   CONSTRAINT AgentRefFK FOREIGN KEY(AgentRef) REFERENCES Agent )
  OBJECT IDENTIFIER IS SYSTEM GENERATED ;
```

Example 19.21

CREATE TYPE and CREATE TABLE statements for residential and industrial properties (corresponds to Example 19.8 for SQL:2016)

```
CREATE TYPE AssessType AS VARRAY(6) OF DECIMAL(9,2);

CREATE TYPE ResidentialType UNDER PropertyType
(BedRooms    INTEGER,
 BathRooms   INTEGER,
 Assessments AssessType )
 NOT FINAL
 INSTANTIABLE;

CREATE TABLE Residential OF ResidentialType
 (CONSTRAINT ResidentialPK PRIMARY KEY(PropNo),
  CONSTRAINT AgentRefFK1 FOREIGN KEY(AgentRef) REFERENCES Agent )
  OBJECT IDENTIFIER IS SYSTEM GENERATED ;

CREATE TYPE IndustrialType UNDER PropertyType
(Zoning          VARCHAR(20),
 AccessDesc      VARCHAR(20),
 RailAvailable   CHAR(1),
 Parking         VARCHAR(10) )
 NOT FINAL
 INSTANTIABLE;

CREATE TABLE Industrial OF IndustrialType
(CONSTRAINT IndustrialPK PRIMARY KEY(PropNo),
 CONSTRAINT AgentRefFK2 FOREIGN KEY(AgentRef) REFERENCES Agent )
 OBJECT IDENTIFIER IS SYSTEM GENERATED ;
```

for the structured *Address* column. If the *Address* column was defined with the ROW type constructor, the Oracle syntax would be identical to Example 19.9 with the ROW keyword replacing *AddressType*.

Example 19.22

Inserting two rows into the typed *Agent* table (corresponds to Example 19.10 for SQL:2016)

```
INSERT INTO Agent
(AgentNo, AName, Address, Email, Phone)
VALUES (999999, 'Sue Smith',
        AddressType('123 Any Street', 'Denver', 'CO', '80217'),
         'sue.smith@anyisp.com', '13031234567');

INSERT INTO Agent
(AgentNo, AName, Address, Email, Phone)
VALUES (999998, 'John Smith',
        AddressType('123 Big Street', 'Boulder', 'CO', '80217'),
         'john.smith@anyisp.com', '13034567123');
```

Because Oracle does not support subtables, additional manipulation statements are used to simulate set inclusion among subtables. In Example 19.23, INSERT statements are used for both the parent table and subtable. Ideally, triggers could be defined to hide the additional manipulation statements.

Example 19.23 also demonstrates the REF function to obtain a system-generated object identifier. When using the REF function, you must use a correlation variable (table alias) as the parameter. You cannot use the table name instead of the correlation variable. The REF statement can also be used in UPDATE statements, as demonstrated in Example 19.24.

Example 19.23

INSERT statements to add an object into the *Property* and *Residential* tables (corresponds to Example 19.11 for SQL:2016)

```
INSERT INTO Residential
(PropNo, Address, SqFt, AgentRef, BedRooms, BathRooms, Assessments)
SELECT 999999, AddressType('123 Any Street', 'Denver', 'CO', '80217'),
       2000, REF(A), 3, 2, AssessType(190000, 200000)
FROM Agent A
WHERE AgentNo = 999999;

-- This INSERT statement maintains set inclusion between the Property
-- and the Residential tables.
INSERT INTO Property
(PropNo, Address, SqFt, AgentRef)
SELECT 999999, AddressType('123 Any Street', 'Denver', 'CO', '80217'),
       2000, REF(A)
FROM Agent A
WHERE AgentNo = 999999;
```

Example 19.24

Using a SELECT statement with the REF Function to retrieve the object identifier of the related *Agent* row (corresponds to Example 19.12 for SQL:2016)

```
UPDATE Residential
  SET AgentRef =
   ( SELECT REF(A) FROM Agent A WHERE AgentNo = 999998 )
  WHERE PropNo = 999999;

-- Maintain consistency between the Property and Residential tables.
UPDATE Property
  SET AgentRef =
   ( SELECT REF(A) FROM Agent A WHERE AgentNo = 999998 )
  WHERE PropNo = 999999;
```

Oracle supports path expressions containing the dot operator and the DEREF function. The DEREF function can also be used in SQL:2016 in place of the -> operator. The DEREF function uses an object identifier as a parameter as shown in Example 19.25. When using columns that have an object type such as *Address*, a correlation variable must be used.

Example 19.25

Oracle SELECT statement with path expressions containing the dot operator and the DEREF function (corresponds to Example 19.13 for SQL:2016)

```
SELECT PropNo, P.Address.City, DEREF(AgentRef).Address.City
 FROM Property P
 WHERE DEREF(AgentRef).AName = 'John Smith';
```

Although Oracle supports the DEREF function, it does not seem necessary to use it. The dot operator can be used in path expressions even when a column has a reference type as shown in Example 19.26. Note that a correlation variable is necessary when using REF columns in a path expression with the dot operator.

Like SQL:2016, Oracle supports the ONLY keyword in the FROM clause. However, the ONLY keyword applies to views not tables in Oracle. Thus, Example 19.14 will not work in Oracle unless object views are used instead of tables.

In place of testing subtable membership, Oracle supports testing the associated type using the IS OF operator. In some situations, the IS OF operator can provide a capability similar to subtable membership testing in SQL:2016. Example 19.27 demonstrates the IS OF operator to test membership in a subtype. The REF operator converts the object type into a reference and the DEREF operator converts a reference into a value so that its type can be tested using the IS OF operator. Example 19.27 does not produce the same result as Example 19.14 because the *Property* rows were not inserted

Example 19.26

Oracle SELECT statement with path expressions containing the dot operator instead of the DEREF function (corresponds to Example 19.13 for SQL:2016)

```
SELECT PropNo, P.Address.City, P.AgentRef.Address.City
 FROM Property P
 WHERE P.AgentRef.AName = 'John Smith';
```

using the *ResidentialType* type. The Oracle syntax for using subtypes does not cover INSERT statements such as Example 19.23 involving reference types.

Example 19.27

Using IS OF to test type membership of a reference type. This example does not produce the same result as Example 19.14 for SQL:2016

```
SELECT PropNo, Address, Location
 FROM Property P
 WHERE Sqft > 1500 AND DEREF(REF(P)) IS OF (ResidentialType)
```

The VALUE function takes a correlation variable as a parameter and returns instances of the object table associated with the correlation variable. Thus, the VALUE function can be used to retrieve all columns from typed tables instead of using the * for untyped tables, as shown in Example 19.28.

Example 19.28

Using the VALUE function to retrieve all columns from a typed table

```
SELECT VALUE(A) FROM Agent A;
```

19.3.3 Dependencies among Types and Typed Tables

Object relational representations involve additional dependencies beyond traditional table designs. User defined types can be referenced in typed tables, columns, and other user defined types. A typed table references its associated type. An attribute or column definition references its associated user-defined type. A subtype references its associated parent type. Figure 19.2 depicts a dependency diagram with subtype references, column usage of types, typed table references, and foreign key references for the objects in the property database.

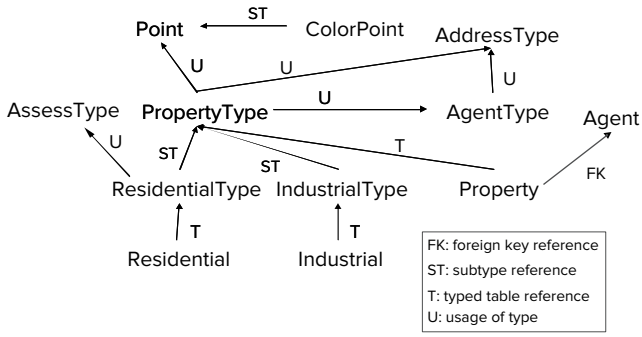


FIGURE 19.2
Dependency Diagram for the Property Database Objects

The usage of user defined types and typed tables introduces dependencies that must be respected when creating and dropping database objects. When creating objects, referenced objects should be created before referencing objects. Thus, a parent type should be created before its related subtypes, a type should be created before columns that reference it, and a type should be created before it is referenced in table definitions. When dropping objects, referencing objects should be dropped before the referenced object. Thus, a subtype should be dropped before its related parent type, a column should be dropped before its related type, and a typed table should be dropped before its related type.

Object orders (list of object names) that respect dependencies are known as topological orders. A topology is a structure with relationships such as a dependency diagram. There are usually many possible orderings that preserve object dependencies. Table 19-2 lists orders consistent with the dependency diagram in Figure 19.2 for CREATE and DROP statements. For CREATE statements, a topological order moves from the top of the diagram to the bottom. For DROP statements, a topological order moves from the bottom of the diagram to the top.

19.3.4 Other Object Features in Oracle

Oracle provides additional object features, some of which extend object features in SQL:2016. Type substitutability and object views provide limited alternatives for subtables. The TABLE collection type corresponding to the multiset type of SQL:2016 supports nested tables. Oracle XML DB provides efficient storage and manipulation of large repositories of XML documents. This section provides an overview of these object features. For more details about these object features, you should consult the Oracle online documentation.

Type Substitution and Object Views The Oracle documentation suggests the use of type substitutability to manage extents of parent tables and related subtables. Type substitutability means that a column or row defined to be of type X can contain instances of X and any of its subtypes. When using type substitutability to support subtables, user-defined types are defined with subtype relationships, but only one typed table (the root table) is defined. All manipulation operations are performed on the root table using type substitution for set inclusion relationships. However, the syntax for using reference types and subtype columns in manipulation statements is not clear. Certain INSERT and UPDATE statements do not work with substituted

TABLE 19-2
Topological Orders for the Property Database Objects

SQL Statement Type	Sample Object Ordering
CREATE	Point, ColorPoint, AddressType, AgentType, PropertyType, AssessType, IndustrialType, ResidentialType, Agent, Property, Residential, Industrial
DROP	Residential, Industrial, Property, Agent, ResidentialType, IndustrialType, PropertyType, AssessType, AgentType, AddressType, ColorPoint, Point

types. For managing set inclusion relationships, type substitution does not support overlapping subtypes. Thus, type substitutability does not provide a satisfactory solution for subtable families due to limited syntax and incomplete support for set inclusion relationships.

The Oracle documentation also suggests the use of object view hierarchies to manage extents of parent tables and related subtables. An object view is a virtual object table in which each row in the view is an object rather than a table row. An object view hierarchy is a set of object views each of which is based on a different type in a type hierarchy. Unlike subtable families, several storage models are possible to store object view hierarchies. The database administrator must choose the best storage model and ensure that users understand usage of object views in retrieval and manipulation statements. Although object view hierarchies may be useful, they do not provide a satisfactory substitute for subtable families because the DBMS does not manage extents of related tables.

Nested Tables Oracle provides extensive support for multiple levels of nested tables corresponding to the multiset feature of SQL:2016. As previously stated, the usage of nested tables is not clear for business databases. Until theory and practice provide more insight, nested table usage is appropriate for specialized situations. To indicate the features for nested tables, the following list summarizes Oracle support for nested tables.

- The TABLE and NESTED TABLE type constructors support CREATE TABLE statements with nested tables. A user-defined type can be specified with the TABLE constructor and then used in a CREATE TABLE statement using the NESTED TABLE constructor.
- Columns with nested table types can appear in query results. The TABLE operator flattens nested tables if a user wants to see flat rather than nested results.
- Comparison operators support equality, subset, and membership comparisons among nested tables.
- Set operators support union, intersection, and difference operations on nested tables as well as duplicate removal and nested table cardinality.
- Object views support multiple levels of nested tables.

The strongest case for nested tables is support for XML documents. XML documents have a hierarchical structure, fitting with the representation of nested tables. However, the SQL:2016 standard and most enterprise DBMS products provide a separate collection of tools for XML document support as presented in the last part of this section.

XML Document Support The eXtensible Markup Language (XML) has emerged as a foundation for electronic commerce for consumers and organizations. XML is a meta language that supports the specification of other languages. To restrict XML documents, XML schemas can be defined. An XML schema specifies the structure, content, and meaning of a set of XML documents. XML schemas support improved data interchange, Internet searching, and data quality. Many application domains have developed XML schemas as an essential element of electronic commerce.

As a result of the growing importance of XML, support for storage and manipulation of XML documents has become a priority for DBMSs. Part 14 of SQL:2016 is devoted to XML document storage and manipulation. Oracle and other commercial DBMS vendors have devoted large amounts of research and development to supporting the Part 14 specification and additional features. The most prominent feature is a new XML data type that most enterprise DBMS vendors support as a prebuilt data type. To provide insight about the extensive XML support available in commercial DBMSs, the following list summarizes features in Oracle XML DB.

- The XMLType data type allows XML documents to be stored as tables and columns of a table.
- Variables in PL/SQL procedures and functions can use the XMLType data type. An application programming interface for XMLType supports a full range of operations on XML documents.
- XML documents can be stored in a structured format using the XMLType data type or in an unstructured format using the CLOB type. Storage as XMLType data allows indexing and specialized query optimization.
- XML schema support applies to both XML documents and relational tables. Oracle can enforce constraints in an XML schema on both tables and XML documents stored in a database.
- XML/SQL duality allows the same data to be manipulated as tables and XML documents. Relational data can be converted into XML and displayed as HTML. XML documents can be converted into relational data.
- Oracle supports the majority of XML operators in the SQL:2016 standard. In particular, Oracle supports the XML traversal operators existsNode(), extract(), extractValue(), updateXML(), and XMLSequence() in the SQL/XML standard.
- Query rewrite transforms operations that traverse XML documents into standard SQL statements. The Oracle query optimizer processes the rewritten SQL statement in the same manner as other SQL statements.

The remainder of this section depicts the basic syntax of the XMLType data type to demonstrate integration of relational databases with hierarchical XML documents. Examples 19.29 and 19.30 demonstrate usage of XMLType columns as well as XMLType tables. Examples 19.31 and 19.32 demonstrate insertion of a row into the *AccountXML1* and *AccountXML2* tables. XMLType values are internally stored using the CLOB (Character Large Object) data type.

Example 19.29

Creating a table with an XMLType column

```
CREATE TABLE AccountXML1 (
 AcctId       INTEGER   PRIMARY KEY,
 AcctDetails  XMLType,
 AcctBal      NUMBER(9,2) );
```

Example 19.30

Creating a table of XMLType

```
CREATE TABLE AccountXML2 OF XMLType;
```

Oracle implements the standard XQuery language using XPath expressions to retrieve data from an XML document. XPath represents an XML document as a tree of nodes. An XPath expression is evaluated to yield an object, which is either a node-set (an unordered collection of nodes without duplicates) or a leaf node value (Boolean, number, or string). The existsNode(), extract(), and extractValue() functions use an XPath expression to retrieve data in an XML document. The existsNode() function is used in a WHERE clause to test the existence of a node in an XML document. The extract() function returns nodes that match its XPath expression. The extractValue() function takes an XPath expression and returns the corresponding leaf node value.

Example 19.31

Inserting a row into a table with an XMLType column

```
INSERT INTO AccountXML1 VALUES (1,
     '<Account>
        <AcctFName>John</AcctFName>
        <AcctLName>Smith</AcctLName>
        <AcctStreet>1234567 Quebec St.</AcctStreet>
        <AcctCity>Denver</AcctCity>
        <AcctState>CO</AcctState>
        <AcctZip>80237</AcctZip>
      </Account>',
     1000);
```

Example 19.32

Inserting a row into a table of XMLType

```
INSERT INTO AccountXML2 VALUES (
    '<Account>
       <AcctFName>John</AcctFName>
       <AcctLName>Smith</AcctLName>
       <AcctStreet>1234567 Quebec St.</AcctStreet>
       <AcctCity>Denver</AcctCity>
       <AcctState>CO</AcctState>
       <AcctZip>80237</AcctZip>
    </Account>' );
```

Examples 19.33 to 19.35 demonstrate usage of the existsNode(), extract(), and extractValue() functions as applied to XMLType columns. Example 19.36 depicts retrieval of the CLOB rows from an XMLType table.

Example 19.33

Selecting the number of rows with a WHERE condition using the existsNode() function. existsNode() returns 1 if the node is found in the XPath expression

```
SELECT  COUNT(*)
 FROM  AccountXML1
 WHERE existsNode(AcctDetails, '/Account/AcctFName') = 1;
```

Oracle also supports an alternative XQuery notation for retrieval known as FLWOR (pronounced flower; acronym incorporates the five main clauses of For, Let, Where, Order By, and Return). The FLWOR notation is somewhat similar to the SQL SELECT statement with the FLWOR FOR clause corresponding to the SQL

Example 19.34

Selecting a row using the extractValue() function

```
SELECT  extractValue(AcctDetails, '/Account/AcctStreet') "Street"
 FROM  AccountXML1
 WHERE extractValue(AcctDetails, '/Account/AcctStreet') LIKE '%St%';
```

Example 19.35

Selecting a row using the existsNode() and extractValue() functions; The existsNode() function uses a condition as part of the XPath expression

```
SELECT  extractValue(AcctDetails, '/Account/AcctCity')
 FROM  AccountXML1
 WHERE existsNode(AcctDetails, '/Account[AcctZip="80237"]') = 1;
```

Example 19.36

Retrieving rows from an XMLType table. The result is identical to retrieving all columns using the * in place of the getClobVal() function

```
SELECT a.getClobVal() FROM AccountXML2 a;
```

FROM clause, the FLWOR WHERE clause corresponding to the SQL WHERE clause, the FLWOR ORDER BY clause corresponding to the SQL ORDER BY clause, and the FLWOR RETURN clause corresponding to the SQL SELECT clause. Example 19.37 demonstrates a simple example to retrieve first names of Denver accounts.

Example 19.37

Retrieving rows using the FLWOR notation

```
SELECT AcctId, XMLQuery(
'for $i in /Account
 where $i /AcctCity = "Denver"
 order by $i/AcctFName
 return $i/AcctFName'
passing by value AcctDetails
RETURNING CONTENT) XMLData
FROM AccountXML1;
```

19.4 OVERVIEW OF NOSQL DATABASE MANAGEMENT

Object database features extend database definition and manipulation of SQL. The need for object features in database technology was a logical extension of object features for programming languages. Almost every modern programming language has object features in its core design. Object features for databases have not had this level of acceptance, however. Despite decades of research and development along with a careful standards effort, object feature usage in enterprises DBMSs seems moderate. Under the radar at first, the marketplace has demanded a new set of features leading to the development of a new class of DBMSs and extensions to existing enterprise DBMSs.

This section provides an overview of NoSQL database technology, a major recent extension of database technology. NoSQL, meaning Not only SQL, provides a broad umbrella of database products and technologies. In November 2017, the nosql-database.org website indicated that more than 225 products are labeled as NoSQL. The first part of this section covers motivation and features for NoSQL database technology, comparing NoSQL to relational DBMSs adhering to the SQL standard. The second part of this section presents data models used in NoSQL DBMSs. These data models provide simplicity and flexibility compared to the rich data representation and strictness of the relational data model.

19.4.1 Motivation and Features

Performance for big data processing is the major driver of NoSQL database technology. Big data applications make high demands on either volume in batch processing or velocity in online processing. Batch processing applications involve high volume as typified by major web vendors such as Google and Facebook with huge amounts of semi-structured data to analyze. Online processing applications involve high velocity as typified by weather stations and satellites, generating enormous levels of sensor data in frequent time intervals. Existing database technology did not meet these performance needs so innovative firms and open source projects created NoSQL database technology.

Use cases depict applications with performance requirements that were not met by enterprise relational DBMSs. Table 19-3 provides a convenient summary of several use cases. Financial trading, cell network monitoring, and environmental monitoring place enormous demands for high velocity on transaction throughput requirements. Enterprise DBMSs using ACID transaction principles could not handle these demands. Database technology using BASE principles (see Chapter 17) and the CAP design philosophy (see Chapter 18) support tradeoffs among consistency, performance, and availability to meet demands from these applications. Enterprise DBMSs with parallel processing architectures are not well-suited for batch applications with huge volumes of semi-structured data such as indexing websites, weather forecasting, and customer profile construction. Big data parallel processing architectures (see Chapter 18) were designed to handle batch processing of huge amounts of semi-structured data. These types of cases motivated organizations to develop specialized solutions that eventually were generalized into NoSQL database technology.

TABLE 19-3
Summary of Use Cases for NoSQL Database Technology

Application	Processing Type	Details
Financial trading	Online	Algorithmic trading based on price and volume movements along with other market activity
Environmental monitoring	Online and batch	Real-time capture of weather sensor data and analysis using complex forecasting models
Cell network monitoring	Online	Monitor cell traffic for bottlenecks and failures
Building customer profiles	Batch	Analyze web logs for customer preferences and recommendations

To meet these demands, NoSQL DBMSs have been developed over the last 15 years. The feature comparison in Table 19-4 indicates an emphasis on flexibility and simplicity for data modeling and high performance on specialized applications in NoSQL database technology. NoSQL DBMSs lack schemas (schema-less) for flexibility and simplicity. NoSQL DBMSs provide improved performance on specialized applications through in-memory transaction processing, lack of constraint checking, relaxed ACID properties with the BASE principle, big data parallel processing, and horizontal scaling across distributed servers. NoSQL DBMSs support automatic sharding, transparent partitioning of data across servers to scale efficiently for increased workloads.

Schema-less data model: does not provide data definition statements such as CREATE TABLE. Data is stored without conformance to a data definition statement.

These advantages involve sharp tradeoffs, however. Developers using a NoSQL DBMS have lower productivity because database manipulation is often performed using an application programming interface rather than a standard query language with a cost-based optimizing compiler. Complex software developed using a NoSQL DBMS may have more faults because of the lack of safety features such as static type checking, schema definition, and constraint checking. To overcome lack of safety features, applications developed with NoSQL technology may require more testing and validation using external schemas. Organizations using a relational DBMS face migration difficulties when moving to a NoSQL DBMS due to major differences in database representation and manipulation. Organizations switching between NoSQL DBMSs face switching difficulties because of lack of standards, especially a query language standard.

The distinction between SQL and NoSQL DBMSs has reduced over time as SQL DBMSs have implemented NoSQL technology. Major enterprise DBMSs now support in-memory transaction processing, columnstore indexes, and big data parallel processing. For flexibility, some major DBMSs now support simplified data models either as an alternative to tables or as an extension. For example, Oracle supports the XML data type allowing flexibility for data representation as part of a table. Oracle and

TABLE 19-4
Comparison of Features of NoSQL and SQL DBMSs

	DBMS Technology	
Feature	**SQL Database**	**NoSQL Database**
Data model	SQL CREATE TABLE statement	Schema-less data models emphasizing simplicity and flexibility; external schemas with validation for some data models
Transaction support	ACID transactions sometimes with in-memory processing	BASE approach with in-memory processing
Query Language	SQL SELECT, INSERT, UPDATE, and DELETE statements	Some proprietary query languages and open source projects
Data types	Standard and extended data types with static type checking	No data types and no static type checking
Constraint checking	Primary key, foreign key, CHECK, and trigger	None or very limited
Optimizing compiler	Cost based optimizing compiler	Lacks cost-based optimization; Load balancing with indexes
Performance	Dependent on optimizing compiler, table partitioning, and parallel processing	Dependent on hardware cluster size and network latency
Scalability	Designed to scale vertically with faster hardware and parallel processing	Designed to scale horizontally through automatic sharding

Microsoft also provide a NoSQL database as a separate product competing directly with NoSQL DBMSs. In the Gartner 2016 analysis of DBMSs[5], the authors indicate that "the NoSQL label will cease to distinguish DBMSs."

Consistent with the blurring between SQL and NoSQL database technology, this textbook provides coverage of NoSQL database technology in various chapters. Table 19-5 summarizes chapter coverage of NoSQL technology and application to relational DBMSs. Chapter 8 presents columnstore indexes, a storage technology first deployed in some NoSQL DBMSs. Chapter 17 presents in-memory transaction processing and the BASE principle, cornerstone technology of NoSQL DBMSs. Chapter 18 presents big data parallel processing architectures and the CAP design philosophy. Although big data parallel processing was not originally designed as part of a DBMS, this technology has acquired DBMS capabilities for analytical queries and data integration. Some enterprise DBMS vendors now integrate architectures for big data parallel processing with standard DBMS parallel processing architectures. The CAP design approach provides tradeoffs that dominate in distributed systems emphasized in NoSQL database technology.

The next subsection covers data models used in NoSQL DBMSs. These data models can be incorporated into relational DBMSs as user defined types, similar to the XMLType in SQL:2016 and Oracle.

19.4.2 Data Models in NoSQL DBMSs

Data models supported in NoSQL DBMSs emphasize simplicity and flexibility. The data models were designed for high performance using relatively simple database structures. For flexibility, the data models are schema-less although some NoSQL DBMSs provide optional schema definition and validation features. This subsection provides an overview of three prominent data models used in NoSQL DBMSs, key-value, document, and graph. This subsection also covers columnar DBMSs, a variation of relational DBMSs, using a different storage model.

Key-Value Data Model The key-value data model supports a set of key-value pairs in a structure known as an associative array, dictionary, or hash table. In a key-value pair, the key identifies a pair item and the value provides content about the item. The key must be unique in the structure so hash lookup can used to retrieve a key-value element using a key. The value can contain any content including text, numbers, lists, web addresses, and binary data such as images. Typically, the value part has a text data type so no type checking occurs on the content. This schema-less approach provides flexibility without restrictions on keys and values. However, this flexibility lacks a mechanism to define types of key-value elements or relationships among key-value elements. Thus, the key-value model is most appropriate for simple databases representing only a small number of entity types.

TABLE 19-5
Summary of Chapter Coverage of NoSQL Database Technology

Chapter	NoSQL Technology	Relational DBMS Usage
Chapter 8	Columnstore storage for some NoSQL DBMSs	Columnstore indexes for summary queries
Chapter 17	In-memory transaction processing, BASE principle	In-memory transaction processing with optimistic concurrency control
Chapter 18	Big data parallel processing architectures, CAP design philosophy	Some usage of big data parallel processing for data integration tasks
Chapter 19	Data models with simplicity and flexibility	Possible usage as a new data type

[5] Heudecker, N. et al. "Magic Quadrant for Operational Database Management Systems," Gartner Report, October 2016.

To depict basic ideas of key-value databases, examples show variations for keys and values. For the initial example, Table 19-6 contains a simple representation with a person's email address as the key and a person's age as the value.

To represent attributes or properties in a key-value database, a key can contain a property name along with a unique entity identifier. Table 19-7 contains a collection of computer models with CPU and memory. The key representation contains the identifier followed by a colon and property name. The key representation can be extended for additional properties such as screen, color, and storage.

A key-value database can also contain lists as values to represent 1-M relationships. Table 19-8 contains a list of products in a shopping cart. The value part contains a variable number of products with each product separated by a comma.

To demonstrate key-value databases for more complex data requirements, the next example compares a key-value database with a relational database. The table design contains two tables, *Agent* (Table 19-9) and *Home* (Table 19-10), with each agent listing one or more homes. The key-value database (Table 19-11) contains key-value pairs for agents and listings. Each key contains an entity, identifier, and property. For example, the key, Agent:A9999:AgFirstName, indicates the Agent entity with identifier A9999 and AgFirstName property. The relationship between agent and listing is encoded with the *AgentId* label in a key. Note that the key-value database does not support referential integrity and data type compliance. However, the key-value database ensures uniqueness of keys.

TABLE 19-6
Example Key-Value Pairs for Email Address Key and Age Value

Key (email address)	Value (age)
William.Smith@StateUniversity.edu	50
Monique5555@hotmail.com	42
AnimeeFan121@gmail.com	22
Lisa.Lee@AnyCompany.com	35

TABLE 19-7
Example Key-Value Pairs for Computer Models

Key (computer model and feature)	Value (feature value)
HP ProBook 455:CPU	AMD A10-9600P Quad Core 2.4 GHz
HP ProBook 455:Memory	16 GB DDR4 SDRAM
MSI GL62M 7RDX-NE 1050i7:CPU	Intel Core i7-7700HQ 2.80 GHz
MSI GL62M 7RDX-NE 1050i7:Memory	8 GB DDR4

TABLE 19-8
Example Key-Value Pairs for Shopping Cart

Key (shopping cart identifier)	Value (list of products)
S111111	PPP101, DFG256, GHI100
X991111	XYZ191, DFG256
S999222	ABC123
R225598	GH100, ABC999, XQZ234, GGG101

TABLE 19-9
Sample Agent Table

AgentId	AgFirstName	AgLastName	AgPhone
A871111	Willie	Jones	(720)555-1212
A991111	Jorge	Lopez	(303)435-9999
A999222	Aimee	Chan	(303)555-8888

TABLE 19-10
Sample Home Table

HomeId	HomeNoBdrms	HomeNoBathrms	HomeAge	AgentId
H111111	3	2	15	A871111
H222222	4	3	25	A871111
H333333	2	2	3	A991111
H444444	5	3	10	A999222

TABLE 19-11
Key-Value Pairs for Agents and Homes

Key (entity:id:property)	Value
Agent:A871111:AgFirstName	Willie
Agent:A871111:AgLasttName	Jones
Agent:A871111:AgPhone	(720)555-1212
Agent:A991111:AgFirstName	Jorge
Agent:A991111:AgLasttName	Lopez
Agent:A991111:AgPhone	(303)435-9999
Agent:A9991111:AgFirstName	Aimee
Agent:A991111:AgLasttName	Chan
Agent:A991111:AgPhone	(303)555-8888
Home:H111111:HomeNoBdrms	3
Home:H111111:HomeNoBathrms	2
Home:H111111:HomeAge	15
Home:H222222:AgentId	A871111
Home:H222222:HomeNoBdrms	4
Home:H222222:HomeNoBathrms	3
Home:H222222:HomeAge	25
Home:H222222:AgentId	A871111
Home:H333333:HomeNoBdrms	2
Home:H333333:HomeNoBathrms	2
Home:H333333:HomeAge	3
Home:H333333:AgentId	A991111
Home:H444444:HomeNoBdrms	5
Home:H444444:HomeNoBathrms	3
Home:H444444:HomeAge	10
Home:H444444:AgentId	A999222

DBMSs using the key-value data model emphasize memory caching of large amounts of simple data. Key-value databases support read-intense applications with simple data requirements such as social networking, gaming, and media sharing. In-memory storage improves retrieval performance of vital data especially in applications with heavy computational requirements. Many NoSQL DBMSs use the key-value data model including Redis, Memcached, and Amazon ElastiCache.

Document Data Model The document data model extends the key-value data model with structure. A document database contains collections of documents in which each document consists of a collection of key-value pairs. A value may be a scalar such as text, number, or date as well as a nested document. Thus, the document data model provides convenient representation of entity types (collections of documents) and relationships (nested documents). In contrast, the key-value data model lacks both features, collections of documents and nesting.

The most prominent specification of the document data model is the **JavaScript Object Notation (JSON)**. JSON is a similar to the XML data type specified in the SQL standard and implemented in Oracle. The next section on Couchbase N1QL provides precise details about JSON documents and their usage in Couchbase N1QL. This subsection informally depicts JSON documents through extensions to examples depicting the key-value data model.

JavaScript Object Notation (JSON)
the most prominent specification of the document data model. JSON, a schema-less notation, supports specification of document collections consisting of key-value pairs with optional nesting of documents.

The first example[6] replicates the key-value representation in Table 19-11. Figures 19.3 and 19.4 show collections of documents for agents and homes, respectively. In these examples, *Agent* and *Home* are collections of documents with the square brackets [] enclosing a set of documents. The curly brackets {} enclose an individual document in a set of documents. The only essential difference in data representation is the single set of key-value pairs in the key-value representation (Table 19-11) versus two sets of documents in Figures 19.3 and 19.4.

The document data model supports nesting of documents, another feature lacking in the key-value data model. Figure 19.5 shows an alternative representation for agents and homes with related homes nested inside an agent. Square brackets surround nested home documents inside an agent document. For example, the agent document with *AgentId* A871111 contains two nested home documents with *HomeId* H111111 and H222222.

Nesting is a specialized representation because nesting makes a set of documents exist only inside another document. The nested table feature in Oracle is specialized because retrieval of a nested table requires retrieval of the parent row.

The document data model has an optional schema feature, another difference with the key-value data model. The JSON schema feature provides documentation as well as structural validation for automated testing and constraint checking of input data. A schema specifies components of a document, data types, cardinalities for nested documents, and other constraints. Unlike SQL, JSON schemas are optional providing flexibility if a schema is not needed.

```
Agent: [
 {AgentId: A871111, AgFirstName: Willie, AgLastName: Jones, AgPhone: (720)555-1212},
  {AgentId: A991111, AgFirstName: Jorge, AgLastName: Lopez, AgPhone: (303)435-9999},
  {AgentId: A999222, AgFirstName: Aimee, AgLastName: Chan, AgPhone: (303)555-8888} ]
```

FIGURE 19.3
Agent Documents

```
Home: [
 {HomeId: H111111, HomeNoBdrms: 3, HomeNoBathrms: 2, HomeAge:15, AgentId: A871111},
 {HomeId: H222222, HomeNoBdrms: 4, HomeNoBathrms: 3, HomeAge:25, AgentId: A871111},
 {HomeId: H333333, HomeNoBdrms: 2, HomeNoBathrms: 2, HomeAge:3, AgentId: A991111},
 {HomeId: H444444, HomeNoBdrms: 5, HomeNoBathrms: 3, HomeAge:10, AgentId: A999222} ]
```

FIGURE 19.4
Home Documents

```
Agent: [
 {AgentId: A871111, AgFirstName: Willie, AgLastName: Jones, AgPhone: (720)555-1212}, Home:[
  {HomeId: H111111, HomeNoBdrms: 3, HomeNoBathrms: 2, HomeAge:15},
  {HomeId: H222222, HomeNoBdrms: 4, HomeNoBathrms: 3, HomeAge:25} ],
 {AgentId: A991111, AgFirstName: Jorge, AgLastName: Lopez, AgPhone: (303)435-9999},Home:[
    {HomeId: H333333, HomeNoBdrms: 2, HomeNoBathrms: 2, HomeAge:3} ],
 {AgentId: A999222, AgFirstName: Aimee, AgLastName: Chan, AgPhone: (303)555-8888}, Home: [
    {HomeId: H444444, HomeNoBdrms: 5, HomeNoBathrms: 3, HomeAge:10} ] ]
```

FIGURE 19.5
Home Documents Nested Inside Agent Documents

[6] The document examples in this section use notation similar to JSON. Section 19.5.1 presents precise JSON notation used in Couchbase Server and many other products.

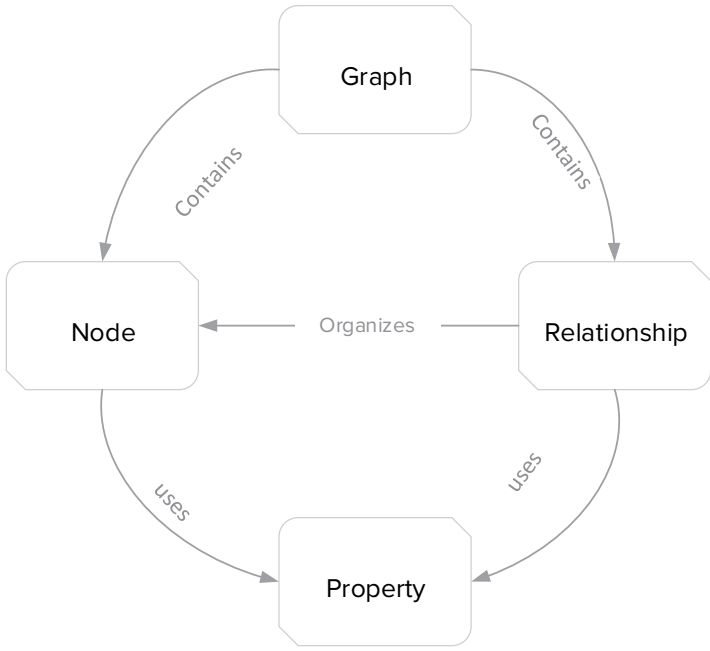
Similar to DBMSs supporting the key-value data model, DBMSs using the document data model emphasize memory caching of large amounts of simple data. The document data model extends the range of applications through document collections and nested documents. Prominent NoSQL DBMSs supporting the document data model are MongoDB, Couchbase, and Amazon DynamoDB.

Graph Data Model The graph data model extends the document data model with relationships. A graph database contains a collection of graphs. Each graph contains nodes and relationships as depicted in Figure 19.6. Nodes and relationships can both contain properties involving a list of key-value pairs. Like the document data model, values can be nested in nodes and relationships. Relationships organize nodes so that nesting values is not necessary. Relationships allow all nodes to participate in relationships, providing the ability to retrieve nodes without reference to parent nodes. The graph data model has an important integrity rule requiring no broken links in a graph. All relationships in a graph have a start and end node.

The graph representation for agents and homes directly shows relationships between agents and homes, eliminating indirect representation through nesting. Figure 19.7 depicts a graph for agent A871111 connected to two homes (H111111 and H222222) via Lists relationships. The nodes contain key-value pairs for agents and homes. The graph data model uses directional relationships with each *Lists* relationship from an agent (start node) to a home (end node) in Figure 19.7. However, relationships support bidirectional retrieval of start and end nodes. Relationships can also have properties such as an *On* property indicating the listing date in Figure 19.7. The entire graph database contains 3 graphs with each agent in a separate graph. Across all graphs, the graph database contains 7 nodes (3 agent nodes and 4 home nodes) and 4 relationships connecting agents and homes.

Although the graph data model can be augmented with a schema, commercial vendors and open-source projects have not produced the same schema features compared to the JSON schema for the document data model. The Structr platform provides a visual data modeling tool to create types for nodes, relationships, and properties. The visual data modeling tool lacks constraint and validation features of the JSON schema tools, however. The graph data model supports a level of referential integrity without a schema validation tool. The graph data model ensures

FIGURE 19.6
Components of the Graph Data Model



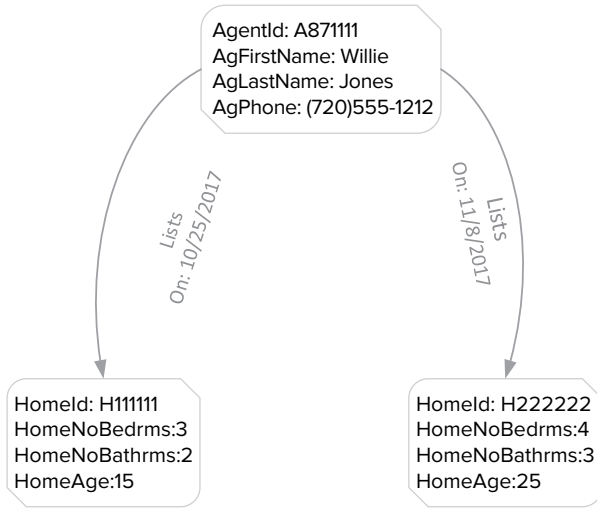


FIGURE 19.7
Graph for an Agent with Two Home Listings

that all relationships have a start and end node so a graph database does not contain incomplete links.

NoSQL DBMSs using the graph data model support large graphs with billions of nodes and relationships for computationally intensive applications. Important use cases for graph databases are fraud detection involving real-time analysis of purchase patterns, recommender systems generating personalized product suggestions, and social network analysis inferring relationships among individuals. Prominent NoSQL DBMSs supporting the graph data model are Neo4j, Titan, OrientDB, and GraphDB.

Columnar DBMS A columnar DBMS is a relational DBMS with a different storage model. Because most columnar DBMSs support SQL and some columnar DBMSs substantially predate NoSQL, designation of columnar DBMSs as NoSQL is questionable. The NoSQL label stems from the different storage model supported by columnar DBMSs rather than the data model.

Columnar DBMSs provide column-oriented storage exclusively or as the major storage approach. Chapter 8 (section 8.3.5) provides details about column-oriented storage and columnstore indexes so details are not repeated here. In review, the column-oriented approach reverses the basic method of storing data. The traditional storage approach, known as a row store, places entire rows in physical records. In contrast, the column-oriented approach places columns in physical records. A columnstore index compresses all or a subset of columns of a table. Since major enterprise DBMSs now support columnstore indexes, the distinguishing feature of columnar DBMSs is the exclusive or major usage of column-oriented storage.

Columnar DBMSs emphasize business intelligence queries often involving grouping and aggregate calculations on tables with large number of rows. In addition to column-oriented storage, columnar DBMSs provide horizontal scaling using distributed clusters of low cost hardware and automatic sharding to distribute data across clusters. The combination of column-oriented storage and horizontal scaling makes columnar DBMSs ideal for data warehouses. Amazon promotes its columnar DBMS, Amazon Redshift, as a data warehouse solution. Early columnar databases were developed many decades ago with Teradata Database in 1979, MonetDB in 1993, and Sybase IQ in 1994.

Apache Cassandra is a prominent open source project supporting a columnar approach. Unlike other columnar DBMSs, Cassandra provides the Cassandra Query Language as it does not fully support SQL. In addition, Cassandra is schema-less unlike other columnar DBMSs. Cassandra supports column families in which rows of a table may have different columns and columns may be nested.

19.5 DATABASE DEFINITION AND MANIPULATION WITH COUCHBASE N1QL

As the previous section indicated, NoSQL DBMSs have wide variation in data models. These variations have impeded development of standardized tools, especially query languages. Development of query languages for document databases has much activity because of the importance of JavaScript and JSON as well as the close link between document representation and XML. A variety of query languages have been developed for document databases in commercial products and open source projects. Some of the notable query languages for document databases are Couchbase N1QL, Microsoft Azure DocumentDB, the open source JSONiq project (www.jsoniq.org), and the open source JMESPath (jmespath.org). Some of these languages extend XQuery, an emerging W3C standard for XML document databases.

This section provides a flavor for query languages for document databases. Since document databases typically use JSON objects, the first section provides syntax for JSON. The second section presents N1QL (pronounced nickel), the document query language for Couchbase. Syntax and examples of important N1QL statements are depicted.

19.5.1 JavaScript Object Notation (JSON)

Section 19.4 introduced basic ideas behind the JavaScript Object Notation (JSON), but did not provide precise details about its syntax. This section provides precise syntax and more examples to depict the syntax. Couchbase N1QL, covered in the next subsection, manipulates JSON databases so a clear understanding of JSON is a prerequisite to understanding Couchbase N1QL.

JSON is not JavaScript, an important programming language for web applications. Rather JSON provides a language independent, data interchange format for objects. JSON format is text so it can be used by web servers and parsed by programming languages. Since JSON is based on the syntax of objects in JavaScript, it can be easily parsed into JavaScript objects. Thus, the JSON name involves JavaScript as the first two parts of the acronym.

The basic syntax for JSON involves key-value pairs. Keys must be enclosed in double quotes. Values can use data types as summarized in Table 19-12. In an object, key-value pairs are separated by commas. Curly braces contain objects, while square brackets contain arrays.

The power of JSON involves objects and arrays as values. Figures 19.9 and 19.10 recast Figures 19.3 and 19.4 using precise JSON syntax. Figure 19.9 contains an object with key *Agent* and value containing an array of three objects. Likewise, Figure 19.10 contains an object with key *Home* and value containing an array of four objects. Note that curly braces surround key-value pairs inside an object, while square brackets surround values inside an array.

TABLE 19-12
JSON Data Types

Data Type	Notes	Value Examples
String	Enclosed in double quotes	"abc", "text example"
Number	Integer, decimal, or floating point format	10, 10.1, 3.1415e8
Object	Enclosed in curly braces {}	{"age":10, "name":"John"}
Array	Enclosed in square brackets []	[1,2,3,4], ["abc", "text example"]
Boolean	Two states (true or false)	true, false
Null	Absence of a value	null

```
{"Agent: ["
 {"AgentId":"A871111", "AgFirstName":"Willie", "AgLastName":"Jones", "AgPhone":"(720)555-1212"},
 {"AgentId": "A991111", "AgFirstName":"Jorge","AgLastName":"Lopez", "AgPhone":"(303)435-9999"},
 {"AgentId":"A999222", "AgFirstName":"Aimee", "AgLastName":"Chan", "AgPhone":"(303)555-8888"} ] }
```

FIGURE 19.9
Agent Object in JSON Syntax

```
{"Home": [
 {"HomeId":"H111111","HomeNoBdrms":3,"HomeNoBathrms":2,"HomeAge":15,"AgentId": "A871111"},
 {"HomeId":"H222222","HomeNoBdrms":4,"HomeNoBathrms":3,"HomeAge":25,"AgentId": "A871111"},
 {"HomeId":"H333333","HomeNoBdrms":2,"HomeNoBathrms":2,"HomeAge":3,"AgentId": "A991111"},
 {"HomeId":"H444444","HomeNoBdrms":5,"HomeNoBathrms":3,"HomeAge":10,"AgentId": "A999222"} ]}
```

FIGURE 19.10
Home Object in JSON Syntax

JSON supports nesting of homes in agents as shown in Figure 19.11, an extension of Figure 19.5 with precise JSON syntax. Figure 19.11 contains an object with key *Agent* and value containing an array of three objects. The last key-value pair in each object contains an array as a value. For the first object, the key *Home* contains two objects. For the second and third objects, the key "Home" contains an array with one object. The nesting of curly braces and square brackets makes JSON tedious to read and write[7].

JSON is a schema-less notation so additional key-value pairs can be added. In addition, values with different data types can be provided for key-value pairs with the same key for flexibility. Figure 19.12 extends Figure 19.11 with additional key-value pairs and different data types. The object with *HomeId* "H555555" contains the null value for *HomeAge* instead of an integer value in other objects. The object with *HomeId* "H666666" contains the *HomeSold* key but not the *HomeAge* key.

Many external tools, programming languages, and DBMSs support JSON. External tools provide parsing, validation, conversion, and formatted display of JSON. For example, the JSON Viewer provides features for hierarchical display, alignment of objects (known as beautify), minimizing display (knowns as minify), validation, and conversion to XML and CSV. Many programming languages provide functions to parse JSON into internal representation and convert internal objects into JSON. For example, JavaScript provides the *parse* function to convert JSON text (typically from a web server) into JavaScript objects. The *stringify* function converts a JavaScript object into JSON. Both relational DBMSs and NoSQL DBMSs support JSON. Many relational DBMSs provide a JSON data type (such as MySQL and IBM Informix) or native

```
{"Agent": [
 {"AgentId":"A871111", "AgFirstName":"Willie", "AgLastName":"Jones", "AgPhone":"(720)555-1212",
   "Home":[{"HomeId":"H111111","HomeNoBdrms":3,"HomeNoBathrms":2,"HomeAge":15},
           {"HomeId":"H222222","HomeNoBdrms":4,"HomeNoBathrms":3,"HomeAge":25} ] },
 {"AgentId": "A991111", "AgFirstName":"Jorge","AgLastName":"Lopez", "AgPhone":"(303)435-9999",
   "Home":[{"HomeId":"H333333","HomeNoBdrms":2,"HomeNoBathrms":2,"HomeAge":3} ] },
 {"AgentId":"A999222", "AgFirstName":"Aimee", "AgLastName":"Chan", "AgPhone":"(303)555-8888",
   "Home": [{"HomeId":"H444444","HomeNoBdrms":5,"HomeNoBathrms":3,"HomeAge":10} ] } ] }
```

FIGURE 19.11
Home Objects Nested Inside Agent Objects in JSON

[7] JSON examples in Figures 19.9 to 19.12 were validated using the JSON validator at jsonlint.com.

FIGURE 19.12
Home Objects with Additions to Key-Value Pairs

```
{"Agent": [
 {"AgentId":"A871111", "AgFirstName":"Willie", "AgLastName":"Jones", "AgPhone":"(720)555-1212",
   "Home":[{"HomeId":"H111111","HomeNoBdrms":3,"HomeNoBathrms":2,"HomeAge":15},
            {"HomeId":"H222222","HomeNoBdrms":4,"HomeNoBathrms":3,"HomeAge":25 } ]  },
 {"AgentId": "A991111", "AgFirstName":"Jorge","AgLastName":"Lopez", "AgPhone":"(303)435-9999",
   "Home": [{"HomeId":"H333333","HomeNoBdrms":2,"HomeNoBathrms":2,"HomeAge":3} ] },
 {"AgentId":"A999222", "AgFirstName":"Aimee", "AgLastName":"Chan", "AgPhone":"(303)555-8888",
   "Home": [{"HomeId":"H444444","HomeNoBdrms":5,"HomeNoBathrms":3,"HomeAge":10},
            {"HomeId":"H555555","HomeNoBdrms":3,"HomeNoBathrms":2,"HomeAge":null},
            {"HomeId":"H666666","HomeNoBdrms":4,"HomeNoBathrms":2,"HomeSold":false}] } ] }
```

support for JSON with transactions, views, and queries (such as Oracle and Microsoft SQL Server). The next subsection demonstrates query language statements for JSON documents in Couchbase Server.

19.5.2 Couchbase N1QL Statements

Couchbase Server, available under an open core license, supports key-value stores and JSON document databases with a NoSQL architecture. An organization can use the free community edition or the subscription commercial edition. Although Couchbase Server supports both key-value stores and document databases, this section only covers JSON-style document databases. Couchbase N1QL was designed for document databases, not key-value stores. As a NoSQL DBMS, Couchbase Server provides a cluster manager, document indexes, and scalability for data, indexes, and query services. To support data scalability, Couchbase Server provides automatic sharding across clusters.

This section provides an overview of N1QL (pronounced nickel) statements for JSON documents in Couchbase Server. N1QL statements look similar to SQL statements, but important differences involve nested documents. This section covers basics of the SELECT statement to manipulate a fully nested document design (Figure 19.11), flat document design (Figures 19.9 and 19.10), and partially nested document design with a parent document just containing unique identifiers of child documents.

Storage of JSON Documents in Buckets Couchbase buckets[8] store collections of documents (JSON objects). A bucket differs from a table in that a bucket can contain documents with varying structure. As bucket examples, Figures 19.9 to 19.11 show documents in the *Agent* bucket, *Home* bucket, and *AgentHome* bucket. The *Agent* and *Home* buckets represent a traditional table design with a foreign key in home documents. The *AgentHome* bucket represents a fully nested design with homes nested inside agent documents.

After creating a bucket, a collection of documents can be stored in the bucket using the N1QL INSERT statement. Examples 19.38 to 19.40 show INSERT statements to store documents in the *Agent*, *Home*, and *AgentHome* buckets. The INTO clause identifies the bucket name before the keywords (KEY, VALUE). The first part of the VALUES clause is the document identifier ("A871111" in Example 19.38) required by N1QL. The first key (*AgentId* in Example 19.38) repeats the document identifier although not required by N1QL. After the document identifier, the INSERT statement contains a JSON object. Appendix 19.A contains the complete set of INSERT statements for all three buckets.

[8] Although Couchbase documentation refers to both keyspaces and buckets, this section uses only buckets. Buckets are created using the Couchbase Web Console.

Example 19.38

INSERT statement for the *Agent* bucket

```
INSERT INTO Agent (KEY, VALUE)
VALUES ("A871111",
  {"AgentId":"A871111", "AgFirstName":"Willie",
   "AgLastName":"Jones", "AgPhone":"(720)555-1212"} );
```

Example 19.39

INSERT statement for the *Home* bucket with a foreign key representation using an agent identifier in a home document. *HomeAddr* is an additional key with an object value beyond the keys shown in Figure 19.10

```
INSERT INTO Home (KEY, VALUE)
VALUES ("H111111",
 {"HomeId":"H111111","HomeNoBdrms":3,
  "HomeNoBathrms":2,"HomeAge":15,"AgentId":"A871111",
  "HomeAddr":{"City":"Denver","State":"CO","ZipCode":80113} } );
```

Example 19.40

INSERT statement for the *AgentHome* bucket with a fully nested representation of homes inside agents. The document inserted into the *AgentHome* bucket contains an additional key (*HomeAddr*) beyond the keys shown in Figure 19.11

```
INSERT INTO AgentHome (KEY, VALUE)
 VALUES 871111",
  {"AgentId":"A871111", "AgFirstName":"Willie",
   "AgLastName":"Jones", "AgPhone":"(720)555-1212",
    "Home":[
     {"HomeId":"H111111","HomeNoBdrms":3,
     "HomeNoBathrms":2,"HomeAge":15,
     "HomeAddr":{"City":"Denver","State":"CO","ZipCode":80110}},
     {"HomeId":"H222222","HomeNoBdrms":4,
     "HomeNoBathrms":3,"HomeAge":25,
     "HomeAddr":{"City":"Denver","State":"CO","ZipCode":80113}} ] }
  );
```

SELECT Statements using Fully Nested Documents The N1QL SELECT statement has the same basic clauses (SELECT, FROM, WHERE, GROUP BY, HAVING) as the SQL SELECT statement. However, the N1QL SELECT statement uses and generates JSON documents rather than tables. Since document databases emphasize nested documents, many queries involve a single bucket[9]. Thus, the first set of SELECT statements involve the *AgentHome* bucket, a fully nested representation.

Example 19.41 depicts a simple example with a WHERE condition and uniform result. In a uniform result, all result documents contain the same key-value pairs. Example 19.42 extends Example 19.41 with a non-uniform result using the * operator, generating all key-value pairs for each document including nested documents. Because JSON documents are schema-less, some documents may contain different sets of key-value pairs as well as arrays of nested objects. The Couchbase Query Workbench provides options to view a query result as JSON, table, tree, or plain text.

Example 19.41

Retrieve selected agent key-value pairs for first names equal to Willie. The double slash indicates a comment

```
SELECT AgLastName, AgPhone
 FROM AgentHome
 WHERE AgFirstName = "Willie";
// JSON result
[
  {
    "AgLastName": "Jones",
    "AgPhone": "(720)555-1212"
  }
]
```

Example 19.42

Retrieve all agent key-value pairs for first names equal to Willie

```
SELECT *
 FROM AgentHome
 WHERE AgFirstName = "Willie";
// JSON result
 [
  {
    "AgentHome": {
      "AgFirstName": "Willie",
      "AgLastName": "Jones",
      "AgPhone": "(720)555-1212",
      "AgentId": "A871111",
      "Home": [
        {
          "HomeAddr": {
```

[9] Couchbase Server has a default limit of 10 buckets per cluster showing a difference between tables and buckets.

```
            "City": "Denver",
            "State": "CO",
            "ZipCode": 80110
          },
          "HomeAge": 15,
          "HomeId": "H111111",
          "HomeNoBathrms": 2,
          "HomeNoBdrms": 3
        },
        {
          "HomeAddr": {
            "City": "Centennial",
            "State": "CO",
            "ZipCode": 80112
          },
          "HomeAge": 25,
          "HomeId": "H222222",
          "HomeNoBathrms": 3,
          "HomeNoBdrms": 4
        }
      ]
    }
  }
]
```

To test conditions on collections, N1QL provides a variety of comparison operators. The ANY operator returns true if one member in a collection satisfies the condition following the SATISFIES keyword. The identifier following the ANY keyword provides an index for the array object. In Example 19.43, the identifier ("h")[10], indexes elements of the Home array. The END keyword terminates the ANY operator.

Example 19.43

Retrieve first name, last name, and homes of agents for agents listing at least one home with number of bedrooms greater than 3. The last document contains a key (*HomeSold*) not contained in other documents

```
SELECT AgFirstName, AgLastName, Home
 FROM AgentHome
 WHERE ANY h IN Home SATISFIES h.HomeNoBdrms > 4 END;
// JSON result
[
  {
    "AgFirstName": "Aimee",
    "AgLastName": "Chan",
    "Home": [
      {
        "HomeAddr": {
          "City": "Aurora",
```

[10] Identifiers in N1QL are case sensitive.

```
            "State": "CO",
            "ZipCode": 80107
          },
          "HomeAge": 10,
          "HomeId": "H444444",
          "HomeNoBathrms": 3,
          "HomeNoBdrms": 5
        },
        {
          "HomeAddr": {
            "City": "Centennial",
            "State": "CO",
            "ZipCode": 80112
          },
          "HomeAge": null,
          "HomeId": "H555555",
          "HomeNoBathrms": 2,
          "HomeNoBdrms": 3
        },
        {
          "HomeAddr": {
            "City": "Aurora",
            "State": "CO",
            "ZipCode": 80109
          },
          "HomeId": "H666666",
          "HomeNoBathrms": 2,
          "HomeNoBdrms": 4,
          "HomeSold": false
        }
      ]
    }
]
```

To retrieve an element of an array, an index can be used inside square brackets. To retrieve all array members, the * should be used inside square brackets as shown in Example 19.44.

The EVERY operator returns true if all members in an array satisfy a specified condition. The syntax for the EVERY operator closely follows the syntax of the ANY operator except for the different keywords as shown in Example 19.45.

Example 19.44

Retrieve first and last names of agents and addresses and ages of homes for agents with at least one home having age less than 10

```
SELECT AgFirstName, AgLastName, Home[*].HomeAddr, Home[*].HomeAge
 FROM AgentHome
 WHERE ANY h IN Home SATISFIES h.HomeAge < 10 END;
// JSON result
[
  {
    "AgFirstName": "Jorge",
    "AgLastName": "Lopez",
    "HomeAddr": [
```

```
      {
        "City": "Denver",
        "State": "CO",
        "ZipCode": 80104
      }
    ],
    "HomeAge": [
      3
    ]
  }
]
```

Example 19.45

Retrieve first and last names of agents and addresses and ages of listed homes for agents with all homes with age greater than 10

```
SELECT AgFirstName, AgLastName, Home[*].HomeAddr, Home[*].HomeAge
 FROM AgentHome
 WHERE EVERY member IN Home SATISFIES member.HomeAge > 10 END;
// JSON result
[
  {
    "AgFirstName": "Willie",
    "AgLastName": "Jones",
    "HomeAddr": [
      {
        "City": "Denver",
        "State": "CO",
        "ZipCode": 80110
      },
      {
        "City": "Centennial",
        "State": "CO",
        "ZipCode": 80112
      }
    ],
    "HomeAge": [
      15,
      25
    ]
  }
]
```

A summary calculation over an array of objects is a subtle part of N1QL. N1QL provides a limited collection of functions to summarize arrays. Examples 19.46 and 19.47 demonstrate the ARRAY_COUNT and ARRAY_SUM functions to summarize the *Home* array. With array summary functions, the GROUP BY clause is not used.

For summary calculations beyond the limited functions for arrays, nested objects (arrays of objects) should be unnested using the **UNNEST operator**. The UNNEST operator appears immediately after the FROM clause as depicted in Example 19.48.

UNNEST operator
flattens an array of objects into a tabular representation by repeating parent details for each object in an array. The UNNEST operator appears immediately after the FROM clause in an N1QL SELECT statement.

Example 19.46

Retrieve first and last names of agents and count of listed homes by the agent

```
SELECT AgFirstName, AgLastName, ARRAY_COUNT(Home) AS HomeCount
 FROM AgentHome;
// JSON result
[
  {
    "AgFirstName": "Willie",
    "AgLastName": "Jones",
    "HomeCount": 2
  },
  {
    "AgFirstName": "Jorge",
    "AgLastName": "Lopez",
    "HomeCount": 1
  },
  {
    "AgFirstName": "Aimee",
    "AgLastName": "Chan",
    "HomeCount": 3
  }
]
```

Example 19.47

Retrieve first and last names of agents and average age of listed homes by the agent

```
SELECT AgFirstName, AgLastName,
        ARRAY_AVG(Home[*].HomeAge) AS HomeAgeAvg
 FROM AgentHome;
// JSON result
[
  {
    "AgFirstName": "Willie",
    "AgLastName": "Jones",
    "HomeAgeAvg": 20
  },
  {
    "AgFirstName": "Jorge",
    "AgLastName": "Lopez",
    "HomeAgeAvg": 3
  },
  {
    "AgFirstName": "Aimee",
    "AgLastName": "Chan",
    "HomeAgeAvg": 10
  }
]
```

The UNEST operator uses a key (*Home*) containing an array of objects. The alias following the *Home* key is optional. The result contains two documents with the same last name ("Jones") in each document.

Example 19.48

Retrieve the last name and home address properties of agent with last name "Jones". Unnest the list of home addresses

```
SELECT AgentHome.AgLastName, h.HomeAddr
 FROM AgentHome
 UNNEST Home h
 WHERE AgentHome.AgLastName = "Jones";
// JSON result
[
  {
    "AgLastName": "Jones",
    "HomeAddr": {
      "City": "Denver",
      "State": "CO",
      "ZipCode": 80110
    }
  },
  {
    "AgLastName": "Jones",
    "HomeAddr": {
      "City": "Centennial",
      "State": "CO",
      "ZipCode": 80112
    }
  }
]
```

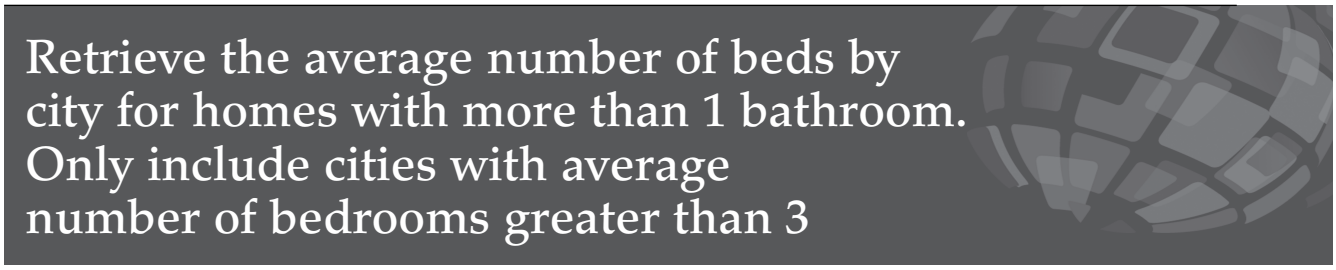
The UNNEST operator combines with the GROUP BY clause and summary functions for nested objects. Examples 19.49 and 19.50 combine the UNNEST operator and the GROUP BY clause. The AVG function in Example 19.50 requires a key name (*h.HomeNoBdrms*). The HAVING clause as shown in Example 19.50 applies to summary functions, similar to the HAVING clause in the SQL SELECT statement.

Example 19.49

Retrieve the count of homes by zip code for homes in Denver

```
SELECT h.HomeAddr.ZipCode, COUNT(*) AS HomeCount
 FROM AgentHome
 UNNEST Home h
 WHERE h.HomeAddr.City = "Denver"
 GROUP BY h.HomeAddr.ZipCode;
//JSON result
[
  {
    "HomeCount": 1,
    "ZipCode": 80104
  },
  {
    "HomeCount": 1,
    "ZipCode": 80110
  }
]
```

Example 19.50

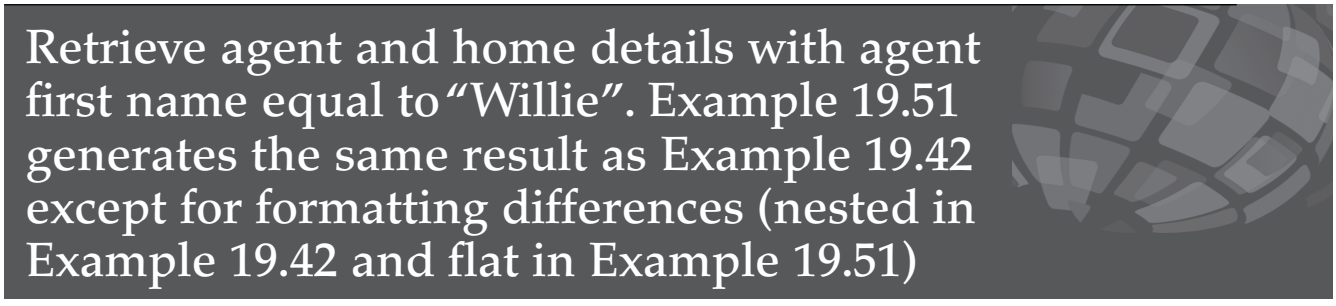



```
SELECT h.HomeAddr.City, AVG(h.HomeNoBdrms) AS AvgHomeBeds
 FROM AgentHome
 UNNEST Home h
 WHERE h.HomeNoBathrms > 1
 GROUP BY h.HomeAddr.City
 HAVING AVG(h.HomeNoBdrms) > 3;
// JSON Result
[
  {
    "AvgHomeBeds": 4.5,
    "City": "Aurora"
  },
  {
    "AvgHomeBeds": 3.5,
    "City": "Centennial"
  }
]
```

N1QL JOIN operator combines documents in different buckets using a key to match documents. The bucket containing join keys must be placed before the JOIN keyword followed by the second bucket after the JOIN keyword.

JOIN Operator for Tabular Document Representation N1QL provides the JOIN operator to combine documents from different buckets. Most databases have multiple buckets even if with document nesting so the JOIN operator is still important for document databases. The **N1QL JOIN operator** is not symmetric as the bucket containing join keys must be placed before the JOIN keyword. In Example 19.51, *Home* (child bucket) precedes the JOIN keyword. The foreign key in Home documents follows the ON KEYS keywords. In Example 19.51, *Home.AgentId* is the foreign key in *Home* documents.

Example 19.51




```
SELECT *
 FROM Home JOIN Agent ON KEYS Home.AgentId
 WHERE Agent.AgFirstName = "Willie";
// JSON result
[
  {
    "Agent": {
      "AgFirstName": "Willie",
```

```
      "AgLastName": "Jones",
      "AgPhone": "(720)555-1212",
      "AgentId": "A871111"
    },
    "Home": {
      "AgentId": "A871111",
      "HomeAddr": {
        "City": "Denver",
        "State": "CO",
        "ZipCode": 80113
      },
      "HomeAge": 15,
      "HomeId": "H111111",
      "HomeNoBathrms": 2,
      "HomeNoBdrms": 3
    }
  },
  {
    "Agent": {
      "AgFirstName": "Willie",
      "AgLastName": "Jones",
      "AgPhone": "(720)555-1212",
      "AgentId": "A871111"
    },
    "Home": {
      "AgentId": "A871111",
      "HomeAddr": {
        "City": "Centennial",
        "State": "CO",
        "ZipCode": 80112
      },
      "HomeAge": 25,
      "HomeId": "H2222222",
      "HomeNoBathrms": 3,
      "HomeNoBdrms": 4
    }
  }
]
```

Other examples indicate that N1QL SELECT statements look similar to SQL SELECT statements for a document representation matching a table design with foreign keys. Examples 19.52 to 19.54 recast previous examples using a document representation with foreign keys. Array functions, array references, and unnesting are not needed in these examples because of the foreign key representation of document relationships.

JOIN and NEST Operators for Partial Document Nesting Previous examples in this subsection demonstrated a fully nested document representation (*AgentHome* buckets) and a tabular representation (*Agent* and *Home*). A partially nested representation uses an array of document identifiers in a parent document. Examples 19.55 and 19.56 show INSERT statements for a partial nested representation with an array of home identifiers in an agent document and a home document without a reference to an agent document. A partial nested representation provides an independent representation for each type of document in a 1-M relationship.

The N1QL JOIN operator can combine two buckets with a partially nested representation. Example 19.57 demonstrates the JOIN operator in which the parent bucket contains matching document identifiers. The parent bucket (*Agent2*) appears before the child bucket (*Home2*) because agent documents contain home identifiers. The JOIN operator requires qualification of a key name with a bucket name or alias (*Agent2.AgFirstName*).

Example 19.52

Retrieve first and last names of agents and addresses and ages of homes for agents with at least one home with age less than 10. Example 19.52 generates the same result as Example 19.44 except for nesting in Example 19.42 and tabular in Example 19.51

```
SELECT A.AgFirstName, A.AgLastName, H.HomeAddr, H.HomeAge
 FROM Home H JOIN Agent A ON KEYS H.AgentId
 WHERE H.HomeAge < 10;
// JSON result
[
  {
    "AgFirstName": "Jorge",
    "AgLastName": "Lopez",
    "HomeAddr": {
      "City": "Denver",
      "State": "CO",
      "ZipCode": 80104
    },
    "HomeAge": 3
  }
]
```

Example 19.53

Retrieve first and last names of agents and count of listed homes by the agent. The JSON result follows the N1QL SELECT statement. Example 19.53 generates the same result as Example 19.46

```
SELECT A.AgFirstName, A.AgLastName, COUNT(*) as HomeCount
 FROM Home H JOIN Agent A ON KEYS H.AgentId
 GROUP BY A.AgFirstName, A.AgLastName;
// JSON result
[
  {
    "AgFirstName": "Jorge",
    "AgLastName": "Lopez",
    "HomeCount": 1
  },
  {
    "AgFirstName": "Aimee",
    "AgLastName": "Chan",
    "HomeCount": 3
  },
  {
    "AgFirstName": "Willie",
    "AgLastName": "Jones",
    "HomeCount": 2
  }
]
```

Example 19.54

Retrieve the average number of beds by city for homes with more than 1 bathroom. Only include cities with average number of bedrooms greater than 3. Example 19.54 generates the same result as Example 19.50

```
 SELECT H.HomeAddr.City, AVG(H.HomeNoBdrms) AS AvgHomeBeds
 FROM Home H JOIN Agent A ON KEYS H.AgentId
 WHERE H.HomeNoBathrms > 1
 GROUP BY H.HomeAddr.City
 HAVING AVG(H.HomeNoBdrms) > 3;
// JSON result
[
  {
    "AvgHomeBeds": 3.5,
    "City": "Centennial"
  },
  {
    "AvgHomeBeds": 4.5,
    "City": "Aurora"
  }
]
```

Example 19.55

INSERT statement for the *Agent2* bucket with nested home identifiers

```
INSERT INTO Agent2 (KEY, VALUE)
VALUES ("A871111",
  {"AgentId":"A871111", "AgFirstName":"Willie",
   "AgLastName":"Jones", "AgPhone":"(720)555-1212",
   "HomeId":["H111111","H222222"] } );
```

Example 19.56

INSERT statement for the *Home2* bucket without an identifier referring to a home document

```
INSERT INTO Home2 (KEY, VALUE)
VALUES ("H111111",
 {"HomeId":"H111111","HomeNoBdrms":3,
  "HomeNoBathrms":2,"HomeAge":15,
  "HomeAddr":{"City":"Denver","State":"CO","ZipCode":80113} } );
```

Example 19.57

Retrieve agent and home details with agent first name equal to "Willie". Example 19.57 generates the same result as Example 19.51 except for different bucket names

```
SELECT *
 FROM Agent2 JOIN Home2 ON KEYS Agent2.HomeId
 WHERE Agent2.AgFirstName = "Willie";
// JSON result
[
  {
    "Agent2": {
      "AgFirstName": "Willie",
      "AgLastName": "Jones",
      "AgPhone": "(720)555-1212",
      "AgentId": "A871111",
      "HomeId": [
        "H111111",
        "H222222"
      ]
    },
    "Home2": {
      "HomeAddr": {
        "City": "Denver",
        "State": "CO",
        "ZipCode": 80113
      },
      "HomeAge": 15,
      "HomeId": "H111111",
      "HomeNoBathrms": 2,
      "HomeNoBdrms": 3
    }
  },
  {
    "Agent2": {
      "AgFirstName": "Willie",
      "AgLastName": "Jones",
      "AgPhone": "(720)555-1212",
      "AgentId": "A871111",
      "HomeId": [
        "H111111",
        "H222222"
      ]
    },
    "Home2": {
      "HomeAddr": {
        "City": "Centennial",
        "State": "CO",
        "ZipCode": 80112
      },
      "HomeAge": 25,
      "HomeId": "H2222222",
      "HomeNoBathrms": 3,
      "HomeNoBdrms": 4
    }
  }
]
```

Example 19.57 demonstrates a fundamental difference between the N1QL JOIN operator and SQL JOIN operator. The N1QL JOIN operator uses document identifiers in either a parent document or child document. Conditions in a SQL JOIN operation only use a foreign key in a child table. However, the N1QL JOIN operator is not symmetric like the SQL JOIN operator because the bucket order matters. The FROM clauses in Examples 19.52 to 19.54 can be rewritten for the partially nested representation in the *Agent2* and *Home2* buckets.

The **NEST operator** provides another way to combine buckets with a partially nested design. Conceptually, the NEST operator yields the opposite of UNNEST. NEST creates an array of child objects inside a parent object, while UNNEST flattens an array of objects, repeating each parent object with a child object. Example 19.58 demonstrates the NEST operator as an alternative to the JOIN operator. The result of Example 19.57 contains two flat objects, each combining an agent and a home object. In contrast, the result of Example 19.58 contains one nested object consisting of an agent object and an array of home objects. The NEST operator requires qualification of a key name with a bucket name or alias (*Agent2.AgFirstName*).

NEST operator combines documents in different buckets as an alternative to the JOIN operator. Like the JOIN operator, the NEST operator requires a key to link parent and child documents. NEST creates an array of child objects inside a parent object.

Example 19.58

Retrieve agent and home details with agent first name equal to "Willie". Nest home documents inside agent documents. Example 19.58 generates the same result as Example 19.57 except for nesting of home objects inside agent objects

```
SELECT *
 FROM Agent2 NEST Home2 ON KEYS Agent2.HomeId
 WHERE Agent2.AgFirstName = "Willie";
// JSON result
[
  {
    "Agent2": {
      "AgFirstName": "Willie",
      "AgLastName": "Jones",
      "AgPhone": "(720)555-1212",
      "AgentId": "A871111",
      "HomeId": [
        "H111111",
        "H222222"
      ]
    },
    "Home2": [
      {
        "HomeAddr": {
          "City": "Denver",
          "State": "CO",
          "ZipCode": 80113
        },
        "HomeAge": 15,
        "HomeId": "H111111",
        "HomeNoBathrms": 2,
        "HomeNoBdrms": 3
      },
```

```
      {
        "HomeAddr": {
          "City": "Centennial",
          "State": "CO",
          "ZipCode": 80112
        },
        "HomeAge": 25,
        "HomeId": "H2222222",
        "HomeNoBathrms": 3,
        "HomeNoBdrms": 4
      }
    ]
  }
]
```

This subsection covered basic parts of N1QL. To master N1QL, you need extended study about array functions and referencing, chaining operators for combining buckets (JOIN, NEST, and UNNEST), subqueries, set operators, one-sided outer joins, and other statements (DELETE, UPDATE, and UPSERT). To master document databases, you need study about more complex data modeling patterns, particularly M-N relationships with attributes. The coverage in this subsection should provide a strong foundation for your extended study.

CLOSING THOUGHTS

This chapter described DBMS extensions for two alternative database representations to support applications with non-traditional data and big data requirements. Object database technology supports applications that integrate complex and simple data and software productivity problems due to type mismatches between DBMSs and programming languages. NoSQL (Not only SQL) database technology supports applications processing huge levels of relatively simple data. Object database technology has been developed for decades in industry and universities resulting in a detailed specification in the SQL standard. NoSQL database technology is a more recent development, spurred by big data demands in ecommerce, finance, medicine, engineering, and science.

To provide a more concrete view of object databases, Section 19.2 presented the object database definition and manipulation features of SQL:2016. User-defined types support new kinds of complex data. Expressions in queries can reference columns based on user-defined types and use methods of user-defined types. SQL:2016 supports inheritance for user-defined types as well as set inclusion relationships for subtable families. Due to the complexity of SQL:2016, few DBMSs conform to all object features in the SQL standard. Object features in Oracle were presented to demonstrate the implementation of many SQL:2016 object features. Section 19.3 provided many examples of Oracle SQL statements for definition and retrieval involving user-defined-types and typed tables. Oracle supports all major object features except subtable families as well as additional features for nested tables and the XML data type.

The last two sections of this chapter covered features common to many NoSQL DBMSs as well as details about Couchbase Server, a prominent NoSQL DBMS. NoSQL DBMSs provide schema-less data models emphasizing simplicity and flexibility to achieve high performance. Section 19.4 depicted common features in NoSQL DBMSs and an overview of data models supported by NoSQL DBMSs. The first part of Section 19.5 provided precise details about the JavaScript Object Notation (JSON) as a

prerequisite to the N1QL query language provided by Couchbase Server. The second part of Section 19.5 depicted the INSERT and SELECT statements of N1QL with examples using nested and tabular JSON documents.

Object database and NoSQL technology have extended rather than disrupted the market for database products and services. Relational database technology disrupted the database market in the 1980s. The dominate DBMSs in the 1970s only survived as legacy products after the onslaught of relational database technology. However, relational DBMSs have maintained dominance partially by integrating object and NoSQL database features. The new class of object-oriented DBMSs developed in the 1990s have almost entirely been swept away by lack of demand for some object features and extension of the SQL standard with object features. Many features initially developed for NoSQL DBMSs have been implemented in relational DBMSs so the distinction between NoSQL and relational DBMSs has blurred over time.

REVIEW CONCEPTS

- Examples of complex data that can be stored in digital format
- Applications needing to integrate simple and complex data as a motivation for object database technology
- Type mismatches as a motivation for object database technology
- User-defined types in SQL:2016 for defining complex data and operations
- Subtable families in SQL:2016: inheritance and set inclusion
- Relationship of subtable families and user-defined types
- Use of the SQL:2016 row type and reference type in object tables
- Use of path expressions and the dereference operator (→) in SQL:2016 SELECT statements
- Referencing subtables in SELECT statements
- Defining and using user-defined types and typed tables in Oracle
- Differences in object features between Oracle and SQL:2016
- XMLType data type to store XML documents in columns and rows
- XQuery functions and notation for manipulating XML documents in a SELECT statement
- Use cases in financial trading, environmental monitoring, cell network monitoring, and customer profiles for NoSQL database technology
- NoSQL database technology emphasizing simplicity and flexibility in data modeling and performance on specialized applications
- Prominent features of NoSQL DBMSs: schema-less data models, in memory transaction processing, proprietary query languages, automatic sharding, and load balancing
- Data models for NoSQL DBMSs: key-value, document, and graph
- Columnar DBMS, relational DBMS with a different storage model
- JavaScript Object Notation (JSON) providing a language independent, data interchange format for objects
- Representation of documents in JSON: fully nested, flat, and partially nested
- Couchbase N1QL for manipulating JSON nested objects
- Couchbase buckets for storage of JSON objects
- N1QL SELECT statement for retrieval of JSON objects

- Clauses in the N1QL SELECT statement: SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY clauses
- ANY and EVERY operators for testing conditions on nested objects
- N1QL array functions for summary calculations on arrays of objects
- N1QL UNNEST operator to flatten nested objects
- N1QL JOIN operator to combine objects with flat or partially nested representation
- N1QL NEST operator to combine objects with a partially nested representation

QUESTIONS

1. How does the use of complex data drive the need for object database technology?
2. What problems are caused by mismatches between the types provided by a DBMS and a programming language?
3. Present an example application that uses both simple and complex data. Use a different application than discussed in Section 19.1.3.
4. What are the components of a user-defined type in SQL:2016?
5. What are the differences between SQL:2016 methods, functions, and procedures?
6. How are SQL:2016 user-defined types used in table definitions and expressions?
7. What is a row type? How are row types used in SQL:2016 table definitions?
8. Explain the differences in encapsulation for user-defined types versus typed tables in SQL:2016.
9. What is a typed table?
10. How do you define a subtable?
11. Discuss the relationship of subtable families and set inclusion.
12. What side effects occur when a row is inserted in a subtable?
13. What side effects occur when a subtable row is updated?
14. What side effects occur when a subtable row is deleted?
15. What is the difference between a foreign key and a reference?
16. When should you use a SELECT statement as part of an INSERT statement when adding objects to a typed table?
17. What is the difference in notation between combining tables that are linked by a foreign key versus a column with a reference type?
18. What is a path expression? When do you use a path expression?
19. When do you need to use the dereference operator (→) in a path expression?
20. What is the purpose of the ONLY keyword in a SQL:2016 SELECT statement?
21. Compare and contrast methods in SQL:2016 with methods in Oracle.
22. What are criteria for overriding a method in Oracle?
23. What is the most significant limitation for object databases in Oracle as compared to SQL:2016?
24. Briefly discuss the importance of object features in Oracle that are not part of SQL:2016.
25. Briefly indicate the uses of the XMLType data type in Oracle.

26. In SQL:2016, what is the difference between the ARRAY and MULTISET collection types?
27. What are the Oracle counterparts of the SQL:2016 collection types?
28. What is the role of nested tables in table design and database application development?
29. What are common prebuilt user-defined data types that are commercially available in enterprise DBMSs?
30. What is XML/SQL duality in Oracle?
31. Evaluate the usage of type substitution as a means of supporting subtable families in Oracle.
32. Evaluate the usage of object view hierarchies as a means of supporting subtable families in Oracle.
33. Evaluate the business case for nested tables.
34. What is the meaning of NoSQL?
35. What is the major driver of NoSQL database technology?
36. Indicate a use case depicting applications with performance requirements not met by enterprise relational DBMSs.
37. What is a schema-less data model?
38. Is the distinction between SQL and NoSQL DBMSs likely to persist over the next 10 years?
39. Briefly describe the key-value data model.
40. Briefly describe the document data model.
41. What is the JavaScript Object Notation (JSON)?
42. Briefly describe the graph data model.
43. What is a columnar DBMS?
44. Is a columnar DBMS properly classified as a NoSQL DBMS?
45. Identify a columnar DBMS that is properly classified as NoSQL.
46. How is JSON related to JavaScript?
47. Briefly explain data types supported in JSON.
48. Briefly explain the syntax of JSON objects.
49. What is a JSON schema?
50. Briefly describe support for JSON in external tools, programming languages, and DBMSs.
51. What is a bucket in Couchbase Server?
52. What is fully nested document design?
53. What is flat or tabular document design?
54. What is partially nested document design?
55. What are the major clauses in the N1QL SELECT statement?
56. Briefly describe the N1QL JOIN operator.
57. How does the N1QL JOIN operator differ from the SQL JOIN operator?
58. Briefly describe the N1QL UNNEST operator.
59. Briefly describe the N1QL NEST operator.
60. Briefly describe the N1QL ANY and EVERY operators.
61. In N1QL, how are elements of arrays referenced?
62. In N1QL, what is the difference between a summary function for arrays and an aggregate function used with the GROUP BY clause?

PROBLEMS

The Part 1 problems provide practice with using SQL:2016 and Oracle (either 11.2g or 12c) to define user-defined types and typed tables as well as to use typed tables. The Part 2 problems provide practice with JSON documents and Couchbase N1QL.

Part 1 Problems for Object Database Manipulation Problems 1 to 26 involve SQL:2016, while problems 27 to 57 involve Oracle. The problems involve the financial database as depicted in Figure 19.P1 except for problems 55 to 58.

1. Using SQL:2016, define a user-defined type for a time series. The variables of a time series include an array of floating point values (maximum of 365), the begin date, the duration (maximum number of data points in the time series), the calendar type (personal or business), and the period (day, week, month, or year). Define methods as listed in Table 19-P1. You need to define the parameters for the methods, not the code to implement the methods. The *TimeSeries* parameter refers to the implicit *TimeSeries* object.
2. Using SQL:2016, define a security type and a typed security table. A security has fields for the unique symbol, the security name, and a time series of closing prices. Both the security type and table have no parent.

FIGURE 19.P1
ERD for the Financial Database

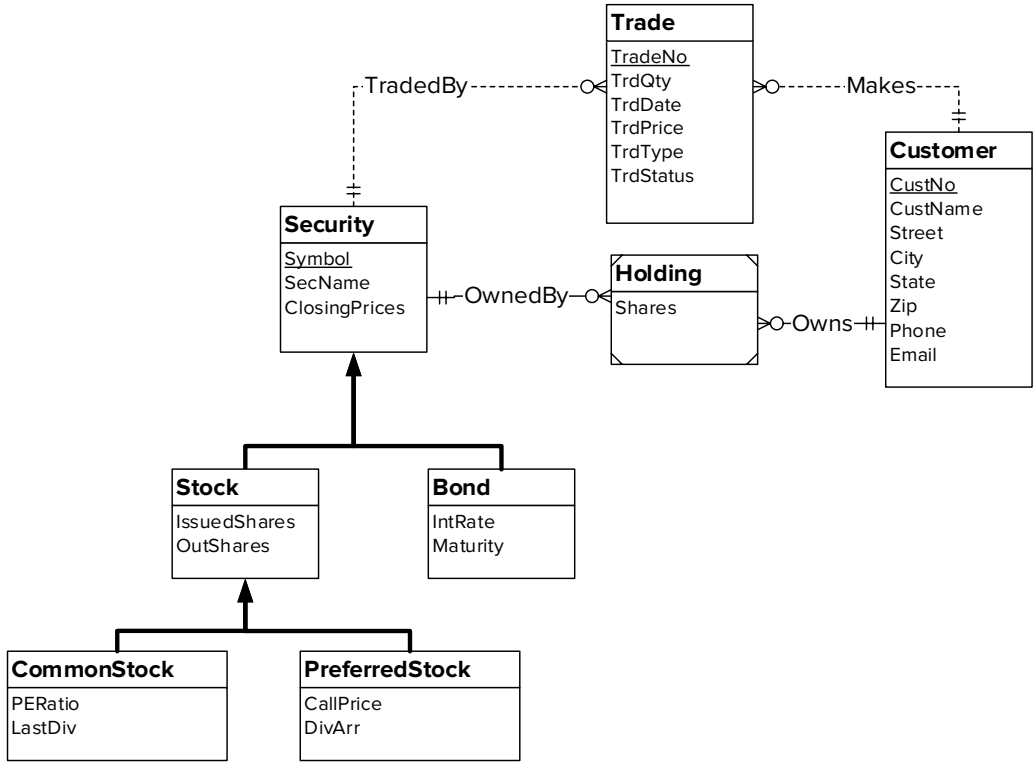


TABLE 19-P1
List of Methods for the TimeSeries Type

Name	Parameters	Result
WeeklyAvg	TimeSeries	TimeSeries
MonthlyAvg	TimeSeries	TimeSeries
YearlyAvg	TimeSeries	TimeSeries
MovingAvg	TimeSeries, Start Date, Number of Values	Float
RetrieveRange	TimeSeries, Start Date, Number of Values	TimeSeries

3. Using SQL:2016, define a stock type and a typed stock table. A stock has fields for the number of issued shares, the number of outstanding shares, and the time series of closing prices. The stock table inherits from the security table, and the stock type inherits from the security type.
4. Using SQL:2016, define a bond type and a typed bond table. A bond has fields for the interest rate and the maturity date. The bond table inherits from the security table, and the bond type inherits from the security type.
5. Using SQL:2016, define a common stock type and a typed common stock table. A common stock has fields for the price earnings ratio and the last dividend amount. The common stock table inherits from the stock table, and the common stock type inherits from the stock type.
6. Using SQL:2016, define a preferred stock type and a typed preferred stock table. A preferred stock has fields for the call price and dividend in arrears. The preferred stock table inherits from the stock table, and the preferred stock type inherits from the stock type.
7. Using SQL:2016, define a customer type and a typed customer table. A customer has fields for the unique customer number, the name, the address, the phone, and the e-mail address. The address field is a row type with fields for street, city, state, and zip. The phone field is a row type with fields for country code, area code, and local number. You should define types for the address and phone so that the types can be reused. Both the customer type and table have no parent.
8. Using SQL:2016, define a portfolio holding type and a typed portfolio holding table. A holding has fields for the customer (reference data type), the security (reference data type), and the shares held. The primary key of the *Holding* table is a combination of the *CustNo* field of the related customer and the *Symbol* field of the related security. Define referential integrity or SCOPE constraints to limit the range of the customer reference and the security reference. Both the holding type and table have no parent.
9. Using SQL:2016, define a trade type and a typed trade table. A trade has fields for the unique trade number, customer (reference data type), security (reference data type), trade date, quantity, unit price, type (buy or sell), and status (pending or complete). The primary key of the *Trade* table is the trade number. Define referential integrity or SCOPE constraints to limit the range of the customer reference and the security reference. Both the trade type and table have no parent.
10. Using SQL:2016, insert an object into the typed *CommonStock* table for Microsoft common stock.
11. Using SQL:2016, insert an object into the typed *CommonStock* table for Dell Corporation common stock.
12. Using SQL:2016, insert an object into the typed *CommonStock* table for IBM common stock. Enter a value in the closing prices (time series type) column by specifying the array of values, the period, the calendar type, the begin date, and the duration.
13. Using SQL:2016, insert an object into the typed *Bond* table for an IBM corporate bond.
14. Using SQL:2016, insert an object into the typed *Customer* table. Use 999999 as the customer number, John Smith as the customer name, and Denver as the city.
15. Using SQL:2016, insert an object into the typed *Customer* table. Use 999998 as the customer number and Sue Smith and Boulder as the city.
16. Using SQL:2016, insert an object into the typed *Holding* table. Connect the holding object to the Microsoft *Security* object and the Sue Smith *Customer* object. Use 200 as the number of shares held.

17. Using SQL:2016, insert an object into the typed *Holding* table. Connect the holding object to the IBM *Security* object and the Sue Smith *Customer* object. Use 100 as the number of shares held.
18. Using SQL:2016, insert an object into the typed *Trade* table. Connect the trade object to the IBM common stock object and the Sue Smith *Customer* object. Use 100 as the quantity of shares traded, "buy" as the trade type, and other values of your choice for the other columns.
19. Using SQL:2016, insert an object into the typed *Trade* table. Connect the trade object to the Microsoft common stock object and the Sue Smith *Customer* object. Use 200 as the quantity of shares traded, "buy" as the trade type, and other values of your choice for the other columns.
20. Using SQL:2016, insert an object into the typed *Trade* table. Connect the trade object to the IBM corporate bond object and the John Smith *Customer* object. Use 150 as the quantity of shares traded, "buy" as the trade type, and other values of your choice for the other columns.
21. Using SQL:2016, update the customer reference column of the *Holding* object from problem 17 to the John Smith *Customer* object.
22. Using SQL:2016, update the customer reference column of the *Trade* object from problem 19 to the John Smith *Customer* object.
23. Using SQL:2016, write a SELECT statement to list the securities held by Denver customers. Only list the securities with more than 100 shares held. Include the customer name, symbol, and shares held in the result.
24. Using SQL:2016, write a SELECT statement to list securities purchased by Boulder customers. Include the customer name, security symbol, trade number, trade date, trade quantity, and unit price in the result.
25. Using SQL:2016, write a SELECT statement to list the customer name, security symbol, and the closing prices for each stock held by Denver customers.
26. Using SQL:2016, write a SELECT statement to list the customer name, security symbol, trade number, trade date, trade quantity, and unit price for common stock purchases by Boulder customers.
27. Using Oracle (either 11.2g or 12c), define a user-defined type for a time series. The variables of a time series include an array of floating point values (maximum of 365), the begin date, the duration (maximum number of data points in the time series), the calendar type (personal or business), and the period (day, week, month, or year). Define methods as previously listed in Table 19-P1. You need to define the parameters for the methods, not the code to implement the methods. The *TimeSeries* parameter refers to the implicit *TimeSeries* object.
28. Using Oracle (either 11.2g or 12c), define a security type and a typed security table. A security has fields for the unique symbol, the security name, and a time series of closing prices. Both the *Security* type and table have no parent.
29. Using Oracle (either 11.2g or 12c), define a stock type and a typed stock table. A stock has fields for the number of issued shares and the number of outstanding shares. The *Stock* table inherits from the *Security* table, and the *Stock* type inherits from the *Security* type.
30. Using Oracle (either 11.2g or 12c), define a bond type and a typed bond table. A bond has fields for the interest rate and the maturity date. The *Bond* table inherits from the *Security* table, and the *Bond* type inherits from the *Security* type.
31. Using Oracle (either 11.2g or 12c), define a common stock type and a typed common stock table. A common stock has fields for the price earnings ratio and the last dividend amount. The common stock table inherits from the stock table, and the common stock type inherits from the stock type.

32. Using Oracle (either 11.2g or 12c), define a preferred stock type and a typed preferred stock table. A preferred stock has fields for the call price and dividend in arrears. The preferred stock table inherits from the stock table, and the preferred stock type inherits from the stock type.
33. Using Oracle (either 11.2g or 12c), define a customer type and a typed customer table. A customer has columns for the unique customer number, the name, the address, the phone, and the e-mail address. The address field is a row type with fields for street, city, state, and zip. The phone field is a row type with fields for country code, area code, and local number. You can define types for the address and phone so that the types can be reused. Both the *Customer* type and table have no parent.
34. Using Oracle (either 11.2g or 12c), define a portfolio holding type and a typed portfolio holding table. A holding has fields for the customer (reference data type), the security (reference data type), and the shares held. The primary key of the *Holding* table is a combination of the *CustNo* column of the related customer and the *Symbol* field of the related security. Define referential integrity or SCOPE constraints to limit the range of the customer reference and the security reference. Both the *Holding* type and table have no parent.
35. Using Oracle (either 11.2g or 12c), define a trade type and a typed trade table. A trade has fields for the unique trade number, customer (reference data type), security (reference data type), trade date, quantity, unit price, type (buy or sell), and status (pending or complete). The primary key of the *Trade* table is the trade number. Define referential integrity or SCOPE constraints to limit the range of the customer reference and the security reference. Define CHECK constraints for the type and status columns. Both the trade type and table have no parent.
36. Using Oracle (either 11.2g or 12c), insert an object into the typed *CommonStock* table for Microsoft common stock. To manage subtables, you should also insert the same object into the typed *Stock* and *Security* tables.
37. Using Oracle (either 11.2g or 12c), insert an object into the typed *CommonStock* table for Dell Corporation common stock. To manage subtables, you should also insert the same object into the typed *Stock* and *Security* tables.
38. Using Oracle (either 11.2g or 12c), insert an object into the typed *CommonStock* table for IBM common stock. To manage subtables, you should also insert the same object into the typed *Stock* and *Security* tables. Enter a value in the closing prices (time-series type) column by specifying the array of values, the period, the calendar type, the begin date, and the duration.
39. Using Oracle (either 11.2g or 12c), insert an object into the typed *Bond* table for an IBM corporate bond.
40. Using Oracle (either 11.2g or 12c), insert an object into the typed *Customer* table. Use 999999 as the customer number, John Smith as the customer name, and Denver as the city.
41. Using Oracle (either 11.2g or 12c), insert an object into the typed *Customer* table. Use 999998 as the customer number, Sue Smith as the customer name, and Boulder as the city.
42. Using Oracle (either 11.2g or 12c), insert an object into the typed *Holding* table. Connect the holding object to the Microsoft *Security* object and the Sue Smith *Customer* object. Use 200 as the number of shares held.
43. Using Oracle (either 11.2g or 12c), insert an object into the typed *Holding* table. Connect the holding object to the IBM *Security* object and the Sue Smith *Customer* object. Use 100 as the number of shares held.
44. Using Oracle (either 11.2g or 12c), insert an object into the typed *Trade* table. Connect the trade object to the IBM common stock object and the Sue Smith

Customer object. Use 100 as the quantity of shares traded, "buy" as the trade type, and other values of your choice for the other columns

45. Using Oracle (either 11.2g or 12c), insert an object into the typed *Trade* table. Connect the trade object to the Microsoft common stock object and the Sue Smith *Customer* object. Use 200 as the quantity of shares traded, "buy" as the trade type, and other values of your choice for the other columns.
46. Using Oracle (either 11.2g or 12c), insert an object into the typed *Trade* table. Connect the trade object to the IBM corporate bond object and the John Smith *Customer* object. Use 150 as the quantity of shares traded, "buy" as the trade type, and other values of your choice for the other columns.
47. Using Oracle (either 11.2g or 12c), update the customer reference column of the *Holding* object from problem 42 to the John Smith *Customer* object.
48. Using Oracle (either 11.2g or 12c), update the customer reference column of the *Trade* object from problem 44 to the John Smith *Customer* object.
49. Using Oracle (either 11.2g or 12c), write a SELECT statement to list the securities held by Denver customers. Only list the securities with more than 100 shares held. Include the customer name, the symbol, and the shares held in the result.
50. Using Oracle (either 11.2g or 12c), write a SELECT statement to list securities purchased by Boulder customers. Include the customer name, security symbol, trade number, trade date, trade quantity, and unit price in the result.
51. Using Oracle (either 11.2g or 12c), write a SELECT statement to list the customer name, security symbol, and the number of shares held for each stock held by Denver customers.
52. Using Oracle (either 11.2g or 12c), write a SELECT statement to list the customer name, security symbol, trade number, trade date, trade quantity, and unit price for common stock purchases by Boulder customers.
53. Write DROP statements in a topological order to delete the types and tables created in problems 27 to 52. You might want to create a dependency diagram to help you determine a topological ordering for object removal.
54. Change the order of the DROP statements in problem 53 to another topological order for object removal.
55. Create a table for orders consisting of a unique order number, order details, and order amount. The order details column should use the Oracle XMLType.
56. Insert two rows into the order table. The order details should have XML data for the customer identifier, first name, last name, city, state, zip, and a list of order lines with product identifier, product name, unit price, and quantity for each order line. Insert one order with one order line and a second order with two order lines.
57. Retrieve the first and last name of orders with order amount greater than $100.
58. Retrieve the first and last name of orders with order amount greater than $100 with the order city of Denver.

Part 2 Problems for JSON Objects and Couchbase N1QL These problems provide practice with creating JSON objects and writing Couchbase N1QL statements. Tables 19-P2 to 19-P5 show sample rows used in these problems, a subset of the Order Entry Database tables in Chapter 4. For N1QL problems, you need to install the community edition of Couchbase Server. Problems 1 to 15 involve a tabular design. Problems 16 to 31 involve a nested design with both partial and full nesting. Problems 28, 30, and 31 involve N1QL queries on an M-N relationship in a nested design, a concept not covered in Section 19.5.2. These problems (28, 30, and 31) depict query formulation difficulties with a nested design for complex data modeling requirements.

TABLE 19-P2
Sample Customer Rows

CustNo	CustFirstName	CustLastName	CustStreet	CustCity	CustState	CustZip	CustBal
C0954327	Sheri	Gordon	336 Hill St.	Littleton	CO	80129-5543	230.00
C1010398	Jim	Glussman	1432 E. Ravenna	Denver	CO	80111-0033	200.00
C2388597	Beth	Taylor	2396 Rafter Rd	Seattle	WA	98103-1121	500.00

TABLE 19-P3
Sample OrderTbl Rows

OrdNo	OrdDate	CustNo	EmpNo	OrdName	OrdStreet	OrdCity	OrdState	OrdZip
O1116324	01/23/2017	C0954327	E8544399	Sheri Gordon	336 Hill St.	Littleton	CO	80129-5543
O1231231	01/23/2017	C1010398		Jim Glussman	1432 E. Ravenna	Denver	CO	80111-0033
O1241518	02/10/2017	C0954327	E9954302	Sheri Gordon	336 Hill St.	Littleton	CO	80129-5543
O2334661	01/14/2017	C2388597	E1329594	Beth Taylor	2396 Rafter Rd	Seattle	WA	98103-1121

TABLE 19-P4
Sample Product Rows

ProdNo	ProdName	ProdMfg	ProdQOH	ProdPrice	ProdNextShipDate
P0036566	17 inch Color Monitor	ColorMeg, Inc.	12	169.00	2/20/2017
P0036577	19 inch Color Monitor	ColorMeg, Inc.	10	319.00	2/20/2017
P1445671	8-Outlet Surge Protector	Intersafe	33	14.99	
P1556678	CVP Ink Jet Color Printer	Connex	8	99.00	1/22/2017
P6677900	Black Ink Jet Cartridge	Connex	44	25.69	
P9995676	Battery Back-up System	Cybercx	12	89.00	2/1/2017

TABLE 19-P5
Sample OrdLine Rows

OrdNo	ProdNo	Qty
O1116324	P1445671	1
O1231231	P0036566	1
O1231231	P1445671	1
O1241518	P0036577	1
O2334661	P6677900	1
O2334661	P9995676	1
O2334661	P1556678	1

1. Write JSON documents for the customer rows in Table 19-P2 using appropriate data types. Use a JSON validation tool such as JSONLint or JSON Viewer to ensure valid JSON documents.
2. Create a *Customer* bucket in Couchbase Server. Write an N1QL statement to create a primary index on the *Customer* bucket (see Appendix 19.A) for examples. Write N1QL INSERT statements to add the JSON documents to the bucket. Appendix 19.A shows INSERT syntax for adding multiple documents to a bucket in one INSERT statement.
3. Write JSON documents for the product rows in Table 19-P4 using appropriate data types. Use the null value for missing values. For the next shipment date,

use an object with three components for the month, day, and year. Use a JSON validation tool such as JSONLint or JSON Viewer to ensure valid JSON documents.

4. Create a *Product* bucket in Couchbase Server. Write an N1QL statement to create a primary index on the *Product* bucket (see Appendix 19.A) for examples. Write N1QL INSERT statements to add the JSON documents to the bucket. Appendix 19.A shows INSERT syntax for adding multiple documents to a bucket in one INSERT statement.
5. Write JSON documents for the order rows in Table 19-P3 using appropriate data types. Use the null value for missing values. For the order date, use an object with three components for the month, day, and year. Use a tabular design for the relationship from customer to order. Use a JSON validation tool Use a JSON validation tool such as JSONLint or JSON Viewer to ensure valid JSON documents.
6. Create an *OrderTbl* bucket in Couchbase Server. Write an N1QL statement to create a primary index on the *OrderTbl* bucket (see Appendix 19.A) for examples. Write N1QL INSERT statements to add the JSON documents to the bucket. Appendix 19.A shows INSERT syntax for adding multiple documents to a bucket in one INSERT statement.
7. Write JSON documents for the order line rows in Table 19-P3 using appropriate data types. Use the tabular design for the relationships from order to order line and product to order line. Use a JSON validation tool Use a JSON validation tool such as JSONLint (jsonlint.com) or JSON Viewer (codebeautify.org/jsonviewer) to ensure valid JSON documents.
8. Create an *OrdLine* bucket in Couchbase Server. Write an N1QL statement to create a primary index on the *OrdLine* bucket (see Appendix 19.A) for examples. Write N1QL INSERT statements to add the JSON documents to the bucket. Appendix 19.A shows INSERT syntax for adding multiple documents to a bucket in one INSERT statement. For the document key, concatenate the order number and product number of an order line such as ("O1116324-P1445671").
9. Write an N1QL SELECT statement to list the customer number, name (first and last), and balance of customers.
10. Write an N1QL SELECT statement to list the customer number, name (first and last), and balance of customers who reside in Colorado (CustState is CO).
11. Write an N1QL SELECT statement to list all columns of the *Product* table for products costing more than $50. Order the result by product manufacturer (*ProdMfg*) and product name.
12. Write an N1QL SELECT statement to list the customer number, name (first and last), city, and balance of customers who reside in Denver with a balance greater than $150 or who reside in Seattle with a balance greater than $300.
13. Write an N1QL SELECT statement to list the order number, order date, customer number, and customer name (first and last) of orders placed in January 2017 sent to Colorado recipients.
14. Write an N1QL SELECT statement to list the average customer balance and order count by customer city. Include only customers residing in Colorado (CO).
15. Write an N1QL SELECT statement to list the product name, sum of the quantity sold, and sum of cost (quantity sold times product price) of products manufactured by "ColorMeg, Inc." or "Connex". Only include products in the result with a sum of cost greater than $50.
16. Revise the JSON representation of customers from problem 1. Add an array of order numbers with key "OrdNo" in each customer. Remove the customer number key in the order documents. Use a JSON validation tool such as JSONLint or JSON Viewer to ensure valid JSON documents.

17. Revise the N1QL INSERT statements for customers from problem 2. Use the JSON documents from problem 16. You can create a new bucket for customers (*Customer2*) or delete documents[11] in the existing customer bucket before inserting new documents. If you create a new bucket, you need to create a primary index for the bucket before inserting documents.
18. Revise the representation of orders from problem 5. Remove the customer number key in order documents. Add another key (*OrdLine*) to contain an array of order lines with each order line containing a product number and quantity ordered. Use a JSON validation tool such as JSONLint or JSON Viewer to ensure valid JSON documents.
19. Revise the N1QL INSERT statements for orders from problem 6. Use the JSON documents from problem 18. You can create a new bucket for orders (*OrderTbl2*) or delete documents in the existing order bucket before inserting new documents. If you create a new bucket, you need to create a primary index for the bucket before inserting documents.
20. Revise the representation of products from problem 3. Add another key (*OrdNo*) to contain an array of order numbers for a partially nested design. Use a JSON validation tool such as JSONLint or JSON Viewer.
21. Revise the N1QL INSERT statements for products from problem 4. Use the JSON documents from problem 20 in the INSERT statements. You can create a new bucket for products (*Product2*) or delete documents in the existing product bucket before inserting new documents. If you create a new bucket, you need to create a primary index for the bucket before inserting documents.
22. Write an N1QL SELECT statement to list all details about customers in Colorado (CO). The result should show the related order numbers nested in a customer. Use the partially nested design starting in problem 16.
23. Write an N1QL SELECT statement to list all details about customers in Colorado (CO) along with all details of related orders. The result should nest the related orders for each customer. Use the partially nested design starting in problem 16.
24. Write an N1QL SELECT statement to list the order number, date, and order line details of orders in January 2017. The result should show the product numbers and quantities nested inside an order. Use the partially nested design starting in problem 16.
25. Write an N1QL SELECT statement to list the order number, date, and order line details of orders in January 2017. The result should unnest order lines product numbers and quantities nested inside an order. Use the partially nested design starting in problem 16.
26. Write an N1QL SELECT statement to list the order number, order date, and count of products ordered for orders in February 2017. Use the partially nested design starting in problem 16.
27. Write an N1QL SELECT statement to list the average customer balance and order count by customer city. Include only customers residing in Colorado (CO). Use the partially nested design starting in problem 16.
28. Write an N1QL SELECT statement to list the product name, product price, order number, and order date for products and related orders. Only include products manufactured by "ColorMeg, Inc." or "Connex". Use the nested design starting in problem 16. Hint: use a join key from the order line array with the notation array[*].joinkey.
29. Write an N1QL SELECT statement to list the order number, order date, and count of the products ordered for orders in January 2017. Use the nested design starting in problem 16.

[11] To delete all customer documents, you should use the syntax DELETE FROM Customer.

30. Write an N1QL SELECT statement to list the product name, order number, order date, and quantity ordered. Only include products manufactured by "ColorMeg, Inc." or "Connex". Use the nested design starting in problem 16. This problem goes a beyond material covered in Section 19.5.2. Hint: in the FROM clause, use the UNNEST operator on the order line array in order. Then, join the unnest result with product using the product number in the unnested array of order lines.
31. Write an N1QL SELECT statement to list the product name, sum of quantity sold, and sum of cost (quantity sold times product price) of products ordered. Only include orders in January 2017. Only include products in the result with a sum of cost greater than $50. Use the nested design starting in problem 16. This problem goes a beyond material covered in Section 19.5.2. Hint: in the FROM clause, use the UNNEST operator on the order line array in order. Then, join the unnest result with product using the product number in the unnested array of order lines.

REFERENCES FOR FURTHER STUDY

The most definitive sources about SQL:2016 are the standards documents available from the InterNational Committee for Information Technology Standards (www.incits.org). The Whitemarsh SQL Standards page (www.wiscorp.com/SQLStandards.html) provides a good summary about the current and historical development of SQL standards. Mimer Developer (developer.mimer.se) provides validation tools for historical SQL standards (1992, 1999, and 2003). Because the standards documents are rather difficult to read, you may prefer books about SQL:1999 by Gulutzan and Pelzer (1999) and Melton and Simon (2001). The major object database features have not changed much since the SQL:1999 standard. For more details about object-relational features in Oracle, you should consult the online database documentation in the Oracle technology network (www.oracle.com/technetwork).

Many sources contain details about NoSQL technology because of the diversity of technology and products. The NoSQL Database page (nosql-database.org) provides a good overview of NoSQL DBMSs. The Couchbase website (couchbase.com) contains documentation and downloads for Couchbase Server. The W3 Schools website contains a well-written tutorial about JSON at https://www.w3schools.com/js/js_json_intro.asp. Validation tools for JSON are JSONLint (jsonlint.com) and JSON Viewer (codebeautify.org/jsonviewer). Alternative query languages for JSON document databases are JSONiq (jsoniq.org), JMESPath (jmespath.org), and Microsoft SQL for DocumentDB (https://docs.microsoft.com/en-us/azure/cosmos-db/documentdb-sql-query).

BIBLIOGRAPHY

Abadi, D., Madden, S., and Hachem, N. "Column-Stores vs. Row-Stores: How Different Are They Really," in *Proceedings of the 2008 ACM SIGMOD Conference*, June 9 to 12, 2008, Vancouver, BC, Canada, 2008, pp. 967–980.

Abudali, A. and Abu-Addose, H. "Data Warehouse Critical Success Factors," *European Journal of Scientific Research* 42 (2), 2010, 326–335.

Armstrong, W. "Dependency Structures of Data Base Relationships," *IFIP Congress*, pp. 580–583, 1974.

Batini, C., Ceri, S., and Navathe, S. *Conceptual Database Design*, Redwood City, CA, Benjamin/Cummings, 1992.

Batra, D. "A Method for Easing Normalization of User Views," *Journal of Management Information Systems* 14, 1 (Summer 1997), 215–233.

Bernstein, P. "Middleware: A Model for Distributed Services," *Communications of the ACM* 39, 2 (February 1996), 86–97.

Bernstein, P. "Repositories and Object-Oriented Databases," in *Proceedings of BTW 97*, Ulm, Germany, Springer-Verlag, (1997), pp. 34–46 (reprinted in *ACM SIGMOD Record* 27 (1), March 1998).

Bernstein, P. and Dayal, U. "An Overview of Repository Technology," in *Proceedings of the 20th Conference on Very Large Data Bases*, Morgan Kaufman, San Francisco, CA, August 1994, pp. 705–713.

Bernstein, P. and Newcomer, E. *Principles of Transaction Processing*, Morgan Kaufmann, 1997.

Bonifati, A., Cattanco, F., Ceri, S., Fuggetta, A., and Paraboschi, S. "Designing Data Marts for Data Warehouses," *ACM Transactions on Software Engineering and Methodolgoy* 10, 4 (2001), 452–483.

Booch, G., Jacobson, I., and Rumbaugh, J. *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA, 1998.

Bouzeghoub, M., Fabret, F., and Matulovic-Broque, M. "Modeling Data Warehouse Refreshment Process as a Workflow Application," in *Proceedings on the International Workshop on Design and Management of Data Warehouses*, Heidelberg, Germany, (June 1999).

Bowman, J., Emerson, S., and Darnovsky, M. *The Practical SQL Handbook*, Reading, MA, Addison-Wesley, 4th Edition, 2001.

Brewer, E. "CAP Twelve Years Later: How the "Rules" Have Changed" *IEEE Computer*, 23–29, February 2012.

Carlis, J. and Maguire, J. *Mastering Data Modeling*, Addison-Wesley, 2001.

Castano, S., Figini, M., Giancarlo, M. and Pierangela, M. *Database Security*, Addison-Wesley, ISBN 0-201-59375-0, 1995.

Celko, J. *Joe Celko's SQL Puzzles & Answers*, San Francisco, CA, Morgan Kaufmann, 1997.

Ceri, S. and Pelagatti, G. *Distributed Databases: Principles and Systems*, New York, NY, McGraw-Hill, 1984.

Chang, L., Wang, Z., Ma, T., Jian, L., Ma, L., Goldshuv, A., Lonegran, L., Cohen, J. Welton, C., Sherry, G., and Bhandarkar, M. "HAWQ: A Massively Parallel Processing SQL Engine in Hadoop," in *Proceedings of the 14th ACM SIGMOD Conference*, Snowbird, UT, USA, June 2014.

Chaudhuri, S. "An Overview of Query Optimization in Relational Systems," in *Proceedings of the ACM Symposium on Principles of Database Systems*, Seattle, WA, 1998, pp. 34–43.

Chaudhuri, S. and Narasayya, V. "An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server," *in Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997, pp. 146–155.

Chaudhuri, S. and Narasayya, V. "Automating Statistics Management for Query Optimizers," *IEEE Transactions on Knowledge and Data Engineering* 13, 1 (January/ February 2001), 7–28.

Choobineh, J., Mannino, M., Konsynski, B., and Nunamaker, J. "An Expert Database Design System Based on Analysis of Forms," *IEEE Trans. Software Engineering* 14, 2 (February 1988), 242–253.

Choobineh, J., Mannino, M., and Tseng, V. "A Form-Based Approach for Database Analysis and Design," *Communications of the ACM*, 35, 2 (February 1992), 108–120.

Choudhary, R. "Key organizational factors in data warehouse architecture selection," Vivekanada Journal of Research (24), 2010, 24–32.

Codd, T. "A Relational Model for Large Shared Data Banks," *Communications of the ACM* 13, 6 (June 1970).

Date, C. "What is a Distributed Database System," in *Relational Database Writings 1985 – 1989*, C. J. Date (ed.) Addison-Wesley, Reading, MA, 1990.

Date, C. *Introduction to Database Systems*, Reading, MA, Addison-Wesley, 8th Edition, 2003.

Date, C. and Darwen, H. *A Guide to the SQL Standard*, Addison-Wesley, Reading, MA, 1997.

Dean, J. and Ghemawat, S. "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, USA, December 2004.

Elmasri, R. and Navathe, S. *Fundamentals of Database Systems*, Pearson Education Unlimited, Seventh Edition, ISBN 13: 978-1-292-09761-9, 2017.

Eckerson, W. "TDWI Benchmark Guide," TDWI Research Report, July 2007, tdwi.org.

Fagin, R "A Normal Form for Relational Databases That is Based on Domains and Keys," *ACM Transactions on Database Systems* 6, 3 (September 1981), 387–415.

Finkelstein, S., Schkolnick, M., and Tiberio, T. "Physical Database Design for Relational Databases," *ACM Transactions on Database Systems* 13, 1 (March 1988), 91–128.

Fisher, J. and Berndt, D. "Creating False Memories: Temporal Reconstruction Errors in Data Warehouses," in *Proceedings Workshop on Technologies and Systems (WITS 2001)*, New Orleans, (December 2001).

Fowler, M. and Scott, K. *UML Distilled*, Addison-Wesley, Reading, MA, 1997.

Friedman, T., Beyer, M., and Thoo, E. "Magic Quadrant for Data Integration Tools," *Gartner RAS Core Research Note G00171986*, November 2009.

Graefe, G. "Options for Physical Database Design," *ACM SIGMOD Record* 22, 3 (September 1993), 76–83.

Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

Groff, J. and Weinberg, P. *SQL: The Complete Reference*, 2nd Edition, New York, NY, Osborne McGraw Hill, 2002.

Gulutzan, P. and Pelzer, T. *SQL-99 Complete, Really*, R & D Books, Lawrence, Kansas, 1999.

Hawryszkiewycz, I. *Database Analysis and Design*, New York, NY, SRA, 1984.

Harizolpulos, S., Abadi, D., Madden, S., and Stonebraker, M. "OLTP Through the Looking Glass, and What We Found There," in *Proceedings of the ACM SIGMOD 08 Conference*, Vancouver, BC Canada (June 9–12, 2008), ACM 978-1-60558-102-6/08/06.

Imhoff, C. "Intelligent Solutions: Oper Marts: An Evolution in the Operational Data Store," *DM Review* 11, 9 (September 2001), 16–18.

Inmon, W. *Information Systems Architecture*, New York, NY, John Wiley & Sons, 1986.

Jarke, M. and Koch, J. "Query Optimization in Database Systems," *ACM Computing Surveys* 16, 2 (June 1984), 111–152.

Kent, W. "A Simple Guide to the Five Normal Forms in Relational Database Theory," *Communications of the ACM* 26, 2 (February 1983), 120–125.

Kimball, R. "Slowly Changing Dimensions," *DBMS* 9, 4 (April 1996) 18–22.

Kimball, R. *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses*, John Wiley and Sons, 2003.

Kimball, R. "The Soul of the Data Warehouse, Part 3: Handling Time," *Intelligent Enterprise Magazine*, April 2003, http://www.intelligententerprise.com.

Laney, D. "3D Data Management: Controlling Data Volume, Velocity and Variety," META Group (now Gartner), February 2001.

Loshin, D. *The Practitioner's Guide to Data Quality Improvement*, Elsevier, Inc., 2011, ISBN: 978-0-12-373717-5.

Mannino, M., Chu, P., and Sager, T. "Statistical Profile Estimation in Database Systems," *ACM Computing Surveys* 20, 3 (September 1988), 191–221.

Martin, J. *Strategic Data-Planning Methodologies*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

Mazon, J., Lechtenbörger, J. and Trujillo, J. "A survey on summarizability issues in multidimensional modeling," *Data and Knowledge Engineering* 68 (12), December 2009, 1452–1469.

McKinsey Global Institute, "Big data: The next frontier for innovation, competition, and productivity, May 2011.

Melton, J. and Simon, A. *Understanding the New SQL: A Complete Guide*, Morgan-Kaufman Publishers, San Mateo, CA, 1992.

Melton, J. and Simon, A. *SQL:1999 Understanding Relational Language Components*, Morgan-Kaufman Publishers, San Mateo, CA, 2001.

Moody, D. and Kortink, M. "From ER Models to Dimensional Models," *Business Intelligence Journal*, 2003, 7–24.

Muller, R. *Database Design for Smarties: Using UML for Data Modeling*, San Francisco, CA, Morgan Kaufmann Publishers, San Francisco, CA, February 1999.

Mullins, C. *Database Administration: The Complete Guide to Practices and Procedures*, Addison Wesley Professional, Second Edition, ISBN 978-0321822949, June 2012.

Nelson, M. and DeMichiel, L. "Recent Trade-Offs in SQL3," *ACM SIGMOD Record* 23, 4 (December 1994), 84–89.

Nijssen, G. and Halpin, T. *Conceptual Schema and Relational Database Design*, Prentice Hall of Australia, 1989.

Olson, J. *Data Quality: The Accuracy Dimension*, Morgan Kaufmann, New York, ISBN 1-55860-891-5, 2002.

Orfali, R., Harkey, D., and Edwards, J. *The Essential Client/Server Survival Guide*, John Wiley and Sons, 2nd Edition, 1996.

Ozsu, T. and Valduriez, P. *Principles of Distributed Database Systems*, Englewood Cliffs, NJ, Prentice-Hall, 1991.

Park, C., Kim, M., and Lee, Y. "Finding an Efficient Rewriting of OLAP Queries Using Materialized Views in Data Warehouses" *Decision Support Systems* 32, 12 (2002), 379–399.

Peinl, P., Reuter, A., and Sammer, H. "High Contention in a Stock Trading Database: A Case Study," in Proceedings of the *ACM SIGMOD Conference*, Chicago, IL, (May 1988), pp. 260–268.

Redman, T. *Data Quality: The Field Guide*, Digital Press, New York, ISBN 1-55558-251-6, 2001.

Romero, O and Abello, A. "A Survey of Multidimensional Modeling Methodologies," *International Journal of Data Warehousing and Mining* 5 (2), April 2009, 1–23.

Saaty, T. *The Analytic Hierarchy Process*, McGraw-Hill, New York, 1988.

Salido, J. and Voon, P. "A Guide to Data Governance for Privacy, Confidentiality, and Compliance (Part 2): People and Process," Microsoft Corporation, www.microsoft.com/privacy/datagovernance.aspx, Whitepaper, January 2010.

Salido, J. and Voon, P. "A Guide to Data Governance for Privacy, Confidentiality, and Compliance (Part 3): Managing Technological Risk," Microsoft Corporation, www.microsoft.com/privacy/datagovernance.aspx, Whitepaper, March 2010.

Salido, J. and Voon, P. "A Guide to Data Governance for Privacy, Confidentiality, and Compliance (Part 4): A Capability Maturity Model," Microsoft Corporation, www.microsoft.com/privacy/datagovernance.aspx, Whitepaper, April 2010.

Sen, A., Ramamurthy, K., and Sinha, P. "A Model of Data Warehousing Process Maturity," *IEEE Transactions on Software Engineering* (38:2), 2012, 336–353.

Shasha, D. and Bonnet, P. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*, Morgan Kaufmann, San Francisco, ISBN 1-55860-753-6, 2003.

Sheth, A., Georgakopoulos, D., and Hornrick, M. "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure," *Distributed and Parallel Databases* 3, (1995), 119–153, Kluwer Academic Publishers.

Shilling, L., Kwan, B., Drolshagen, C., and Hosokawa, P. "Scalable Architecture for Federated Translational Inquiries Network (SAFTINet) Technology Infrastructure for a Distributed Data Network," *PubMed eGEMS* 1, 1, (October 2013), US National Library of Medicine, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4371513/.

Sigal, M. "A Common Sense Development Strategy," *Communications of the ACM* 41, 9 (September 1998), 42–48.

Su, S., Dujmovic, J., Batory, D., Navathe, S., and Elnicki, R. "A Cost-Benefit Decision Model: Analysis, Comparison, and Selection of Data Management Systems," *ACM Transactions on Database Systems* 12, 3 (September 1987), 472–520.

Sutter, J. "Project-Based Warehouses," *Communications of the ACM* 41, 9 (September 1998), 49–51.

Teorey, T., Lightstone, S., Nadeau, T., and Jagadish, H. *Database Modeling and Design*, Elsevier Science and Technology Books, 5th Edition, ISBN 13: 978-0-12-685352-0, 2005.

Thomas, G. "The DGI Data Governance Framework," Data Governance Institute Whitepaper, www.DataGovernance.com, 2012.

Thomas, G. "Preventative and Detective Controls," Data Governance Institute Whitepaper, www.DataGovernance.com, 2012.

Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. "Speedy Transactions in Multicore In-Memory Databases," in *Proceedings ACM SOSP 13 Conference*, Nov. 3–6, 2013, Farmington, PA, USA, ACM 978-1-4503-2388-8/13/11, http://dx.doi.org/10.1145/2517349.2522713.

Westerman, P. *Data Warehousing: Using the Wal-Mart Model*, Morgan Kaufmann, 2000.

Zahedi, F. "The Analytic Hierarchy Process: A Survey of the Method and Its Applications," *Interfaces* 16, 4, (1986), 96–108.

INDEX

D

E

F

G

H

I

T