ST. PAUL THE APOSTLE
UNIVERSITY
OF INFORMATION SCIENCE AND TECHNOLOGY

PRACTICAL DATABASE DESIGN:
FROM CONCEPT TO IMPLEMENTATION

Dijana Capeska Bogatinoska
Aleksandar Karadimce

UNIVERSITY OF INFORMATION SCIENCE AND TECHNOLOGY (UIST)
"ST. PAUL THE APOSTLE", OHRID

Ohrid, 2026

ST. PAUL THE APOSTLE
UNIVERSITY
OF INFORMATION SCIENCE AND TECHNOLOGY

PRACTICAL DATABASE DESIGN:

FROM CONCEPT TO IMPLEMENTATION

Dijana Capeska Bogatinoska

Aleksandar Karadimce

Ohrid, 2026

UNIVERSITY OF INFORMATION SCIENCE AND TECHNOLOGY

"ST. PAUL THE APOSTLE", OHRID

FACULTY OF INFORMATION SYSTEMS, VISUALIZATION,

MULTIMEDIA AND ANIMATION

PRACTICAL DATABASE DESIGN:

FROM CONCEPT TO IMPLEMENTATION

Dijana Capeska Bogatinoska

Aleksandar Karadimce

Ohrid, 2026

Authors:
Dijana Capeska Bogatinoska, PhD
Aleksandar Karadimce, PhD

Title of the issue:
Practical Database Design: From Concept to Implementation

Editor of the publishing activity of UINT:
Ivan Bimbilovski, PhD

Editors of the issue:
Dijana Capeska Bogatinoska, PhD
Aleksandar Karadimce, PhD

Reviewers:
Atanas Hristov, PhD
Rasim Salkoski, PhD

Cover design:
Andrej Bogatinoski

Technical processing:
Dijana Capeska Bogatinoska, PhD

Proofreading in the English language:
Lela Ivanovska, PhD

Publisher:
University of Information Science and Technology "St. Paul the Apostle"

Place and year of issue:
Ohrid, 2026

Preface

Relational databases, though a technology with decades of history, continue to hold their position as the most widely used data management systems in the world. Despite the rapid rise of alternative database paradigms collectively known as NoSQL — optimized for massive, unstructured, or distributed data — research and industry surveys consistently show that relational databases remain the first choice of the vast majority of developers and organizations. The reasons are several and compelling: the solid mathematical foundation rooted in relational theory; the standardized query language SQL, which enables portability across different platforms and vendors; strict adherence to the ACID principles — Atomicity, Consistency, Isolation, and Durability — which guarantee the reliability and consistency of data even in complex, concurrent environments; mature and well-tested tooling developed over decades of industrial use; and robust transaction support that makes relational systems the backbone of financial, healthcare, and enterprise applications worldwide. These characteristics make relational databases an indispensable pillar of modern information technology.

Although quality literature exists in this field, most available textbooks take a predominantly theoretical approach, without detailed, step-by-step guidance through the process of designing and implementing databases. The authors' many years of experience working with students at the University of Information Science and Technology "St. Paul the Apostle" (UIST) in Ohrid clearly demonstrated the need for precisely this kind of textbook — one that gives students a solid theoretical foundation while simultaneously showing them, practically and systematically, how to apply those concepts in real situations. The ultimate goal is to equip young professionals with the skills to work independently and confidently with relational databases, supported by a deep theoretical understanding.

A defining characteristic of this textbook is the use of two consistent case studies — the **University Management System** and the **Company/Business Database** — which are carried throughout every chapter of the book. The idea for this approach stems from the authors' conviction that concepts are better understood when applied to concrete, real-world examples. Every new topic is introduced and illustrated through the same two examples, so that the reader progressively builds a complete picture of both systems — from requirements analysis and conceptual modeling, through logical design and implementation, to normalization. The university example was chosen for its familiarity to students, while the company example offers directly applicable skills for the business world. The two examples are developed in parallel and comparatively, demonstrating the universality of the concepts and the breadth of their application.

This textbook is intended primarily for undergraduate students encountering the field of relational databases for the first time. It is designed as the required textbook for the course **Databases I** across all study programs at UIST, and as supplementary reading for **Management of Information Systems** in the Digital Business Analytics program at the Faculty of Information and Communication Sciences. However, the book is also useful to programmers, analysts, and technical professionals who wish to deepen their knowledge of

relational database design and implementation. The detailed, step-by-step approach also makes it accessible to those studying the material independently.

The textbook is organized into four thematic parts:

Part 1 — Foundations of Database Systems lays the groundwork. Chapter 1 introduces the concept of a database, database management systems (DBMS), the historical evolution of the technology, system architecture, and the ACID principles. Chapter 2 presents the complete design process — the phases of development, the database lifecycle, and requirements analysis.

Part 2 — Conceptual Design develops the skills for data modeling. Chapter 3 introduces the fundamentals of conceptual design and the methodology for working with requirements. Chapter 4 is dedicated to ER modeling in Chen notation — entities, attributes, relationships, cardinality, and participation constraints, with a complete step-by-step construction of diagrams for both case studies. Chapter 5 extends the ER model with Enhanced ER (EER) modeling — specializations, generalizations, inheritance hierarchies and lattices, and all associated constraint types.

Part 3 — Logical Design and Implementation bridges conceptual models and working databases. Chapter 6 introduces the principles of logical design and prepares readers for the systematic transformation of ER and EER diagrams into relational schemas. Chapter 7 covers the relational model in full — the mapping rules, integrity constraints, and complete relational schemas for both case studies. Chapter 8 addresses relational algebra — the foundational operations that underlie the SQL language. Chapter 9 covers SQL implementation — DDL, DML, complex queries, joins, and views, with practical application to both case studies.

Part 4 — Advanced Topics rounds out the theoretical picture. Chapter 10 covers normalization — functional dependencies and normal forms from 1NF through BCNF — with concrete examples drawn from both case studies.

The book concludes with **Chapter 11 — Integrated Case Studies**, which serves as a consolidated reference for both systems: the complete relational schemas and sample datasets used throughout the book.

One of the distinctive methodological features of this textbook is the systematic approach to requirements analysis when building ER diagrams. Business requirements are analyzed sentence by sentence — each sentence is broken down to identify the entities, attributes, and relationships it contains. This approach gives readers a concrete, repeatable technique for translating narrative descriptions into precise diagrams systematically rather than intuitively. The same method is applied consistently to both case studies across all chapters, so that readers build a durable, transferable skill rather than simply memorizing a specific example.

On the implementation side, the book covers two SQL dialects in parallel throughout: **Microsoft SQL Server**, which is used as the primary platform in the Databases I laboratory course, and **MySQL**, used in courses such as Web Development. Working with both dialects side by side not only reflects realistic professional practice — where different projects use different systems — but also makes the transition to other SQL environments, such as

PostgreSQL, Oracle PL/SQL, or any other relational platform, straightforward and natural. The differences between the dialects are noted where relevant, so students understand both what is standardized and what is vendor-specific.

The writing style throughout the book is descriptive and narrative in style, aimed at giving readers a clear mental picture of every concept before formal definitions are introduced. Each chapter concludes with review questions, practical exercises, and a list of references related to the chapter's content.

The authors are aware that every first edition can be improved through suggestions, experiences, and observations of its readers. We therefore sincerely invite all readers — students, instructors, and practitioners alike — to share their comments, proposals, and corrections. All constructive feedback is welcome and will be carefully considered in the preparation of future editions. The authors can be reached at:

Assoc. Prof. Dr. Dijana Capeska Bogatinoska: dijana.c.bogatinoska@uist.edu.mk Assoc. Prof. Dr. Aleksandar Karadimce: aleksandar.karadimce@uist.edu.mk

We wish all readers every success in mastering the material and much satisfaction on the journey through the world of relational databases.

Ohrid, May 2026

The Authors,

Assoc. Prof. Dr. Dijana Capeska Bogatinoska

Assoc. Prof. Dr. Aleksandar Karadimce

CONTENT

FOUNDATIONS OF DATABASE SYSTEMS 1

1. Introduction to Databases 4
1.1 Introduction: The Information Age Challenge 4
1.2 What Are Databases? 4
1.3 Why Databases Matter: The Data-Driven World 5
1.4 Database Management Systems: The Software Foundation 6
1.4.1 What Does a DBMS Do? 6
1.5 The Evolution of Database Systems: From File Systems to Relational Databases 7
1.5.1 File-Based Systems: The Dark Ages of Data Management 7
1.5.2 The Relational Revolution: E.F. Codd's Breakthrough 8
1.5.3 Mathematical Theory: Why It Matters 8
1.6 ACID Properties and Transaction Reliability 9
1.7 Components of Database Systems: A Complete Ecosystem 9
1.7.1 Database System Components 10
1.8 Relational Databases Today: Still the Dominant Choice 10
1.8.1 Relational vs. NoSQL: Choosing the Right Tool 11
1.9 Our Approach: Learning Through Case Studies 11
1.9.1 Our Pedagogical Approach 12
1.10 Looking Forward: Your Database Design Journey 12
1.11 Summary: Databases as the Foundation of Modern Systems 13
Review Questions 13
References and Recommended Readings 16

2. Database Design Process 18
2.1 Introduction: From Chaos to Structure 18
2.2 Understanding the Database Development Lifecycle (DDLC) 19
2.2.1 The DBMS-Independent to DBMS-Specific Transition 20
2.3 Phase 1: Requirements Collection and Analysis 20
2.3.1 The Nature of Requirements 21
2.3.2 Discovery Techniques: Extracting Knowledge from Stakeholders 22
2.3.3 The Challenge of Non-Technical Clients 22
2.3.4 Discovering Business Rules 23
2.3.5 Critical Success Factors for Requirements Analysis 23
2.4 Phase 2: Conceptual Design 24
2.4.1 Purpose and Characteristics of Conceptual Design 24
2.4.2 From Requirements to ER Model 25
2.4.3 Why Chen Notation? 26
2.4.4 The Role of Iteration and Refinement 26
2.5 Phase 3: Logical Design 26
2.5.1 The Transformation from ER to Relational Model 27
2.5.2 Normalization: Refining the Logical Structure 27
2.5.3 Maintaining DBMS Category Independence 28
2.6 Phase 4: Physical Design and Implementation 28
2.6.1 Physical Design Decisions 28
2.6.2 Implementation: Creating the Physical Database 29

2.6.3 Data Loading and Conversion 29
2.6.4 Testing and Validation 30
2.7 The Iterative Nature of Database Design 30
2.7.1 Why Iteration Occurs 30
2.7.2 Managing Iteration Productively 31
2.8 Stakeholder Engagement: Working with Non-Technical Clients 31
2.8.1 Understanding Stakeholder Perspectives 31
2.8.2 Effective Communication Strategies 32
2.8.3 Success Principles for Stakeholder Engagement 32
2.9 From Requirements to Reality: University and Company Examples 33
2.10 Summary: Database Design as Systematic Transformation 35
Review Questions 36
References and Recommended Readings 39

CONCEPTUAL DESIGN 41

3. Conceptual Design 43
3.1 Introduction: The Art of Understanding Before Building 43
3.2 Understanding Conceptual Database Design 44
3.2.1 Why Abstraction Matters 44
3.2.2 From Business Requirements to ER Elements 44
3.2.3 Characteristics of Effective Conceptual Models 45
3.3 Why Conceptual Design Determines Project Success 45
3.3.1 The Foundation Effect 45
3.3.2 The Cost of Change Curve 46
3.3.3 A Failure Story: When Conceptual Design is Skipped 47
3.4 The Three-Phase Conceptual Design Process 48
3.4.1 Phase 1: Requirements Discovery 48
3.4.2 Phase 2: Data Analysis and Modeling 49
3.4.3 Phase 3: Validation and Refinement 49
3.4.4 How This Connects to ER/EER Modeling 50
3.5 Working with Business Stakeholders 50
3.5.1 The Communication Challenge 50
3.5.2 Practical Communication Techniques 51
3.5.3 Managing Conflicting Requirements 51
3.6 Business Rules: The Hidden Foundation of Database Design 51
3.6.1 Four Categories of Business Rules 52
3.6.2 Discovering Hidden Rules 52
3.6.3 From Business Rules to ER Constraints 53
3.6.4 Why This Matters for ER Modeling 53
3.7 Quality Assurance: Getting Your Models Right 53
3.7.1 The Four Quality Standards 54
3.7.2 Validation Techniques 54
3.7.3 Common Quality Problems and Solutions 54
3.7.4 Preparing for ER Diagram Validation 55
3.8 Tools for Conceptual Design 55
3.8.1 Choosing Your Modeling Notation 55
3.8.2 Software Tool Categories 56
3.8.3 Getting Ready for Chapter 4 57

3.9 Summary: Your Foundation for ER and EER Modeling 58
3.9.1 What You've Learned 58
3.9.2 The Bridge to ER and EER Modeling 58
3.9.3 Your Next Steps 59
Review Questions 59
References and Recommended Readings 61

4. Entity-Relationship (ER) Modeling 63
4.1 The Art of Seeing Data Relationships 63
4.2 Introduction to ER Modeling 63
4.3 The Role of ER Diagrams in Database Design 64
4.4 Chen Notation: The Language of Precise Design 65
4.4.1 Historical Foundation and Design Principles 65
4.4.2 Advantages of Chen Notation 65
4.4.3 Comparison with Alternative Notations 66
4.5 ER Diagram Symbols and Components 66
4.5.1 Symbol Set 66
4.5.2 Connection Types and Constraints 66
4.5.3 Naming Conventions in ER Modeling 67
4.6 Entities: The Foundation of Data Models 69
4.6.1 What is an Entity? 69
4.6.2 Categories of Entities 70
4.6.3 Entity Types and Instances 70
4.6.4 Unique Identification of Entity Instances 71
4.6.5 Recognizing Entities in Requirements 72
4.6.6 Entity Classification: Strong vs. Weak Entities 74
4.6.7 Strong Entities 74
4.6.8 Weak Entities 75
4.6.9 Graphical Representation 76
4.6.10 Naming Best Practices 77
4.6.11 Common Mistakes When Identifying Entities 78
4.7 Attributes: The Characteristics of Entities 79
4.7.1 What is an Attribute? 79
4.7.2 Attributes in Our Case Studies 79
4.7.3 Attribute Domains 80
4.7.4 Categories of Attributes 82
4.7.4.1 Simple (Atomic) Attributes 83
4.7.4.2 Composite (Complex) Attributes 85
4.7.4.3 Multivalued Attributes 87
4.7.4.4 Derived Attributes 89
4.7.5 Key Attributes 92
4.7.6 Integrated Attribute Examples 96
4.7.7 Attribute Design Principles 99
4.8 Relationships: The Connections That Give Meaning to Data 101
4.8.1 What is a Relationship? 101
4.8.2 Relationships in Our Case Studies 102
4.8.3 Graphical Representation 103
4.8.4 Degree of Relationships 104
4.8.4.1 Unary (Recursive) Relationships 104
4.8.4.2 Binary Relationships 105

4.8.4.3 Ternary Relationships 106
4.8.4.4 Higher-Degree Relationships 107
4.8.5 Cardinality of Relationships 107
4.8.5.1 One-to-One (1:1) Relationships 108
4.8.5.2 One-to-Many (1:N) Relationships 109
4.8.5.3 Many-to-Many (M:N) Relationships 110
4.8.5.4 Ternary Relationship Cardinality (M:N:K) 112
4.8.5.5 Cardinality Summary and Selection Guidelines 114
4.8.6 Participation Constraints 115
4.8.6.1 Total Participation (Existence Dependency) 115
4.8.6.2 Partial Participation 117
4.8.6.3 Mixed Participation Scenarios 118
4.8.7 Weak Entities and Their Identifying Relationships 119
4.8.7.1 Characteristics of Identifying Relationships 119
4.8.8 Relationship Design Principles 121
4.8.8.1 Strategic Design Guidelines 121
4.8.9 Common Relationship Design Mistakes 122
4.8.9.1 Structural Design Errors 122
4.8.9.2 Cardinality and Constraint Errors 123
4.8.9.3 Complexity Management Errors 124
4.9 Creating ER Diagrams from Scratch: A Step-by-Step Approach 124
4.9.1 The Step-by-Step Methodology 125
4.9.2 Case Study 1: University Database System 125
4.9.3 Sentence-by-Sentence Analysis 126
4.9.4 Constructing the Complete ER Diagram 139
4.9.5 Case Study 2: Company Database System 141
4.9.6 Sentence-by-Sentence Analysis 142
4.9.7 Summary of Company Database Components 154
4.9.8 Constructing the Complete Company ER Diagram 157
4.10 Bringing It All Together 159
4.11 Chapter Summary 160
Review Questions 161
Exercises 161
References and Recommended Readings 166

5. Enhanced ER (EER) Modeling 168
5.1 From ER to EER: When Basic Modeling Isn't Enough 168
5.1.1 Example of the Evolution of Requirements 168
5.1.2 Limitations of ER Modeling 168
5.1.3 New Scenarios Requirements 170
5.1.4 The Need for Semantic Richness 171
5.1.5 The IS-A Relationship 171
5.1.6 Preview: The EER Solution 172
5.1.7 The Path Forward 172
5.2 Subclasses and Superclasses: The Foundation of EER 172
5.2.1 The Concept of Subclasses and Superclasses 172
5.2.2 Identity of the Entity 173
5.2.3 The Superclass/Subclass Relationship 174
5.2.4 When Subclasses Are Needed 174
5.2.5 Attribute and Relationship Inheritance 176

5.2.6 Mathematical Foundation 177
5.2.7 Advantages of Subclass/Superclass Modeling 178
5.3 Specialization and Generalization 179
5.3.1 Specialization: Top-Down Approach 179
5.3.1.1 Graphical Representation of Specialization 179
5.3.1.2 The Specialization Process: Case Study Applications 183
5.3.2 Generalization: Bottom-Up Approach 191
5.3.2.1 Graphical Representation of Generalization 191
5.3.2.4 Alternative Notation: The Unified Circle Symbol 195
5.3.3 Specialization vs. Generalization: Understanding the Relationship 196
5.3.3.1 Design Process in Practice 197
5.3.4 Systematic EER Development Methodology 199
5.3.4.1 Understanding the Requirements 199
5.3.4.2 Building the EER Diagram Incrementally 200
5.3.4.3 Validating the Model 200
5.3.4.4 Balancing Semantic Richness and Implementation Efficiency 201
5.3.4.5 Methodology Principles 202
5.4 Constraints on Specialization and Generalization 202
5.4.1 The Two Constraint Dimensions 202
5.4.2 Disjointness Constraints 203
5.4.2.1 Disjoint Specialization 203
5.4.2.2 Overlapping Specialization 204
5.4.3 Completeness Constraints 205
5.4.3.1 Total Specialization 205
5.4.3.2 Partial Specialization 206
5.4.4 Combined Constraint Scenarios 206
5.4.4.1 Scenario 1: Disjoint, Total (d, t) 207
5.4.4.2 Scenario 2: Disjoint, Partial (d, p) 207
5.4.4.3 Scenario 3: Overlapping, Total (o, t) 207
5.4.4.4 Scenario 4: Overlapping, Partial (o, p) 208
5.4.5 Determining Correct Constraints 209
5.4.6 Constraint Notation Preview 210
5.5 Specialization Hierarchies and Lattices 211
5.5.1 Specialization Hierarchies 211
5.5.1.1 Characteristics of Hierarchical Structures 211
5.5.1.2 Advantages of Hierarchical Structures 215
5.5.2 Specialization Lattices 216
5.5.2.1 Characteristics of Lattice Structures 216
5.5.3 Hierarchies vs. Lattices 221
5.5.3.1 When to Use Hierarchies 221
5.5.3.2 When to Use Lattices 221
5.5.3.3 Implementation Complexity Trade-offs 222
5.5.4 Depth Considerations in EER Structures 222
5.6 Chen Notation for EER Constructs 223
5.6.1 Basic Chen EER Symbols 224
5.6.2 Constraint Representation in Chen Notation 224
5.6.2.1 Disjointness Constraints 225
5.6.2.2 Completeness Constraints 225
5.6.3 Integrated Notation Example: University System 226
5.6.4 Integrated Notation Example: Company System 228

5.6.5 Attribute-Defined Specialization Notation 231
5.6.5.1 Conceptual Foundation 231
5.6.5.2 Chen Notation for Attribute-Defined Specialization 232
5.6.6 Multi-Level Specialization Representation 234
5.6.7 Chen Notation Best Practices 236
5.6.7.1 Essential Notation Guidelines 236
5.7 Complete EER Case Studies: University and Company Systems 237
5.7.1 Extended University System: Complete EER Model 237
5.7.1.1 EER Extension Requirements 237
5.7.1.2 Systematic Modeling Process 237
5.7.1.3 Validation Through Business Scenarios 241
5.7.1.4 Complete Model Summary 243
5.7.1.5 Integrated EER and ER Model 244
5.7.2 Extended Company System: Complete EER Model 247
5.7.2.1 EER Extension Requirements 247
5.7.2.2 Systematic Modeling Process 248
5.7.2.3 Validation Through Business Scenarios 250
5.7.2.4 Complete EER Model Summary 251
5.7.2.5 Integrated EER and ER Model 252
5.7.3 EER Design Lessons from Case Studies 256
5.7.3.1 Contrasting Patterns: Lattice vs. Multiple Independent Specializations 257
5.8 Chapter Summary 257
Review Questions 258
Exercises 259
References and Recommended Readings 266

LOGICAL DESIGN & IMPLEMENTATION 268

6. Logical Design and Implementation 270
6.1 Introduction: From Vision to Reality 270
6.2 Understanding Logical Database Design 270
6.2.1 Key Aspects of Logical Design 271
6.3 Why Logical Design Matters 271
6.3.1 Why Separate Logical Design Phase? 271
6.4 The Logical Design Process 272
6.5 The Relational Model: Our Foundation 273
6.6 Relational Algebra: The Query Language 274
6.7 SQL Implementation: Making It Real 274
6.8 Our Journey Through Part 3 275
6.9 Success Principles for Logical Design 275
6.10 The Path Forward 276
6.11 Summary: Preparing for Transformation 276
Review Questions 277
References and Recommended Readings 279

7. Relational Model 281
7.1 Introduction: Relational Model 281
7.2 What is the Relational Model? 282
7.2.1 History of the Relational Model 283

7.3 Elements of the Relational Model 284
7.3.1 Attribute in Relational Model 285
7.3.2 The Role of Domain 288
7.3.3 Tuple in Relational Model 288
7.3.4 The Degree of Relation 290
7.3.5 The Cardinality of Relation 290
7.3.6 The Relational Database 291
7.3.7 Characteristics of Relations 291
7.3.8 Relational Model Notation 293
7.4 Database Relation 293
7.4.1 Mathematical Relation 294
7.4.2 Relational Database Schema 295
7.4.3 Description of Relation Instance 298
7.4.4 Properties of Database Relation 299
7.4.5 Relational Keys 300
7.4.5.1 The Role of Keys in Relational Databases 300
7.4.6 Entity Integrity, Referential Integrity, and Foreign Keys 303
7.4.7 Summary of Relational Model Fundamentals 305
7.5 ER and EER to Relational Model Mapping 305
7.5.1 Why Do We Need Mapping? 306
7.5.2 The Conceptual to Logical Transition 306
7.5.3 What Happens During Mapping? 307
7.5.4 Benefits of Systematic Mapping 308
7.5.5 ER-to-Relational Mapping Algorithm Overview 309
7.5.6 EER-to-Relational Mapping Algorithm Overview 311
7.6 Creating the Relational Model Schema 314
7.6.1 University System Entities 315
7.6.2 University System Relationships and Keys 317
7.6.3 University System Multivalued Attributes and N-ary Relationships 323
7.6.4 University System Integrity Constraints 325
7.6.5 The Complete Relational Model Schema of the University System 326
7.6.6 Company System Entities 327
7.6.7 Company System Relationships and Keys 329
7.6.8 Company System Integrity Constraints 333
7.6.9 The Complete Relational Model Schema of the Company System 334
7.7 Mapping EER Constructs to Relations 334
7.7.1 Introduction to EER Mapping Challenges 334
7.7.2 University System EER Mapping 335
7.7.3 Complete University EER Relational Schema 340
7.7.4 Company System EER Mapping 343
7.7.5 Complete Company EER Relational Schema 347
7.8 Summary 350
Review Questions 351
Exercises 353
References and Recommended Readings 359

8. Relational Algebra 361
8.1 Introduction to Relational Algebra 361
8.2 Relational Algebra 362
8.3 Relational Algebra Operations 364

8.3.1 Categories of Operations 365
8.3.2 Operation Notation 366
8.4 Unary Relational Operations 369
8.4.1 The Selection Operation 369
8.4.2 The Projection Operation 372
8.4.3 The Rename Operation 373
8.4.4 The Distinct Operation 375
8.5 Set Operations 377
8.5.1 The Union Operation 377
8.5.2 The Intersection Operation 379
8.5.3 The Set Difference Operation 381
8.5.4 The Cartesian Product Operation 383
8.6 Binary Relational Operations 386
8.6.1 The Natural Join Operation 387
8.6.2 The Equijoin Operation 389
8.6.3 The Theta Join Operation 391
8.6.4 The Outer Join Operation 394
8.6.5 The division Operation 398
8.7 Extended Relational Algebra 401
8.7.1 Aggregate Functions 401
8.7.2 Grouping Operation 403
8.7.3 Sorting Operation 404
8.8 Complex Relational Algebra 406
8.8.1 Multi-Step Queries 407
8.8.2 Alternative approaches comparison 412
8.8.3 Query Optimization Concepts 418
8.8.4 Performance Selection and Projection Pushdown Principles 424
8.9 Summary 429
Review Questions 430
Exercises 431
References and Recommended Readings 434

9. SQL Implementation 435
9.1 Introduction: The Information Age Resolution 435
9.2 Why SQL Matters for Databases 436
9.3 Introduction to SQL 437
9.3.1 History of SQL 437
9.3.2 SQL Importance and Database Implementations 438
9.3.3 SQL Command Syntax 439
9.3.4 Database Variables and Data Types 440
9.4 Creating a Database 441
9.4.1 DDL Overview and Schema Concepts 442
9.4.2 Mapping the Relational Model to SQL DDL Commands 442
9.4.3 Creating Database and Tables 444
9.5 From Relational Algebra to SQL DML Queries 463
9.5.1 Working with Data in a Database 464
9.5.2 Basic SQL Queries 466
9.5.3 Substring Pattern Matching, Concatenation, and Arithmetic Operations 471
9.5.4 SQL Multiset Comparison and Ordering of Query Results 475
SELECT City 482

9.5.5 The EXISTS function in SQL 484
9.5.6 Joined Tables in SQL 485
9.5.7 Aggregate and Grouping Functions in SQL 495
9.5.8 Views (Virtual Tables) in SQL 498
9.5.9 Advanced SQL Queries 500
9.5.10 Query Optimization and Execution 510
9.6 Data Security: DCL Commands 521
9.7 Differences between SQL Server and MySQL 523
9.7.1 SQL Server vs. MySQL: General Differences 523
9.7.2 SQL Server vs. MySQL: Implementation Differences 524
9.7.3 SQL Server vs. MySQL: CREATE DATABASE Syntax Comparison 525
9.8 Summary 526
Review Questions 527
Exercises 528
References and Recommended Readings 530

ADVANCED TOPICS 531

10. Normalization 533
10.1 Why Normalization Matters 533
10.1.1 The Problem with Storing Everything Together 533
10.1.2 Update Anomalies 534
10.1.3 Insertion Anomalies 535
10.1.4 Deletion Anomalies 535
10.1.5 The Root Cause: Mixing Independent Facts 535
10.2 Functional Dependencies 536
10.2.1 Definition and Notation 536
10.2.2 Full and Partial Functional Dependency 538
10.2.3 Transitive Functional Dependency 540
10.2.4 Keys Revisited: Superkeys, Candidate Keys, and Primary Keys 541
10.3 First Normal Form (1NF) 542
10.3.1 Definition and Requirements 542
10.3.2 Recognizing 1NF Violations 543
10.3.3 Achieving 1NF: University and Company Examples 543
10.4 Second Normal Form (2NF) 544
10.4.1 Definition and Requirements 545
10.4.2 Identifying Partial Dependencies 545
10.4.3 Decomposition to 2NF: University Example 545
10.4.4 Decomposition to 2NF: Company Example 546
10.5 Third Normal Form (3NF) 547
10.5.1 Definition and Requirements 547
10.5.2 Identifying Transitive Dependencies 547
10.5.3 Decomposition to 3NF: University Example 548
10.5.4 Decomposition to 3NF: Company Example 548
10.6 Boyce-Codd Normal Form (BCNF) 549
10.6.1 Definition and Relationship to 3NF 549
10.6.2 When 3NF Is Not Enough: Identifying BCNF Violations 550
10.6.3 Decomposition to BCNF 551
10.7 Beyond BCNF: A Brief Overview 552

10.8 Normalization and ER-Based Design 552
10.8.1 Why Good ER Design Produces Normalized Schemas 552
10.8.2 Verifying Our University and Company Schemas 553
10.9 When to Denormalize 554
10.10 Summary 555
Review Questions 556
Exercises 557
References and Recommended Readings 562

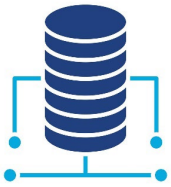
11. Integrated Case Studies 563
11.1 Introduction 563
11.2 University Management System 563
11.2.1 Relational Schema 563
11.2.2 Sample Data 564
11.3 Company/Business Database 569
11.3.1 Relational Schema 569
11.3.2 Sample Data 570

Part 1

Foundations of Database Systems

Introduction to Databases

Database Design Process



Learning Objectives:

By completing Part 1, you will be able to:

- Understand the fundamental concepts, purpose, and importance of database systems in modern organizations
- Explain the evolution from file-based systems to modern relational databases and the problems this evolution solved
- Describe the components of database systems and how Database Management Systems (DBMS) provide essential capabilities
- Understand ACID properties and why they are critical for reliable transaction processing
- Compare relational databases with alternative approaches (NoSQL) and make informed technology choices
- Comprehend the complete database design lifecycle from requirements gathering through physical implementation
- Identify and engage effectively with different stakeholder types during requirements analysis
- Apply systematic methodologies for transforming business requirements into database designs
- Recognize the distinct phases of database development and understand how they connect
- Appreciate the role of database design in overall information systems architecture

***Chapter Overview*:**

Chapter 1: Introduction to Databases establishes a foundation by defining databases and Database Management Systems, explaining their critical role in modern organizations. It traces the evolution from problematic file-based systems, through Dr. E.F. Codd's relational revolution, to contemporary database technologies. The chapter introduces ACID properties, database system components, and explains why relational databases remain the dominant choice despite being over 50 years old. Previews the University and Company case studies that will guide learning throughout the book.

Chapter 2: Database Design Process presents the complete database development lifecycle, showing how systematic methodologies transform organizational requirements into functioning database systems. It covers the phases of requirements collection, conceptual design, logical design, physical design, and implementation. The chapter demonstrates stakeholder engagement techniques, requirements analysis methods, and the iterative refinement process. It explains how the conceptual, logical, and physical design phases connect and build upon each other, establishing a roadmap for the comprehensive journey through Parts 2 and 3.

Our Comprehensive Case Study Approach

Throughout Part 1, we introduce two real-world case studies that will guide your learning journey throughout the entire book: the **University Management System** (modeling students, courses, professors, faculties, and academic relationships) and the **Company/Business Database** (handling employees, departments, customers, orders, products, and organizational structures).

In Chapter 1, you will see how these domains demonstrate universal database principles. Both require managing complex relationships, maintaining data integrity, supporting concurrent access, and enabling sophisticated queries. In Chapter 2, you will learn how systematic design processes apply to both academic and business contexts, learning to gather requirements from different stakeholder types and recognize patterns that transcend specific domains.

These foundational chapters establish the context and methodology that Parts 2 and 3 develop in depth. You will see how conceptual design (Part 2) captures business requirements independently of implementation technology, how logical design (Part 3) transforms those requirements into structured relational schemas, and how physical implementation (Part 3) creates functioning database systems. The systematic approach introduced here becomes a framework for all subsequent learning, ensuring that every technique and concept is connected to practical database development, from requirements through deployment.

By establishing these foundations, you will be prepared to master ER and EER modeling techniques in Part 2 as well as relational implementation skills in Part 3, while maintaining an understanding of how each technique fits within the larger database development lifecycle.

Chapter 1

1. Introduction to Databases

1.1 Introduction: The Information Age Challenge

Imagine walking into a large university library in 1950. You are searching for information about a specific research topic. The librarian directs you to massive card catalogs, wooden cabinets filled with thousands of index cards, each carefully typed and filed alphabetically. Finding a single book requires searching through multiple catalogs: one by author, another by title, yet another by subject. If you need information from multiple sources, you must repeat this process. Updating the catalog when new books arrive requires typing new cards and manually inserting them into the correct position across all catalog systems.

When managing a modern library with millions of books, thousands of daily transactions, and hundreds of simultaneous users, the traditional card catalog approach simply cannot scale. A fundamentally different solution is required — one that can store vast amounts of information, retrieve it instantly, maintain consistency across multiple access points, and adapt to changing requirements. This is where **databases** become essential.

This scenario illustrates why databases have become the fundamental infrastructure of modern society. Whenever users search online, make a purchase, check their bank balance, book a flight, or post on social media, they interact with databases. Universities track millions of students and their academic records. Companies manage complex supply chains, customer relationships, and financial transactions. Hospitals maintain patient histories and coordinate care across multiple departments. Governments process tax records, issue licenses, and maintain public services (Elmasri & Navathe, 2016).

This chapter explores what databases are, why they are important, and how they evolved into the sophisticated systems we rely on today. The reader will understand why relational databases remain the dominant technological choice even after 50 years, and you will see how the systematic approach you will learn in this book applies to real-world scenarios through our University and Company case studies.

1.2 What Are Databases?

At its most fundamental level, a **database** is an organized collection of related data that serves a specific purpose (Date, 2003). But this simple definition hides remarkable sophistication. Let's unpack what makes databases powerful.

> **Definition: Database**
>
> A **database** is a shared, integrated collection of persistent data that models a portion of the real world (the universe of discourse or mini world), structured to support the efficient storage, retrieval, and manipulation of information for specific applications (Elmasri & Navathe, 2016).

This definition reveals several critical characteristics:

Shared: Multiple users and applications can access the same data simultaneously without conflicts or inconsistencies. When a university registrar updates a student's grade, that information becomes immediately available to the student and the professor. They all view the same record.

Integrated: Data from different sources combine into a unified whole. A company database integrates customer information from sales, billing, and support systems, eliminating the redundancy and inconsistencies that arise when each department maintains separate files.

Persistent: Data survives beyond the execution of any single program. Unlike variables in a running application that disappear when the program ends, database content persists indefinitely, surviving system restarts and application updates.

Structured: Data is organized according to specific rules and relationships that reflect real-world meaning. A university database doesn't just store random facts about students and courses. It captures the semantic relationships (students *enroll in* courses, professors *teach* courses) that define how universities actually operate.

1.3 Why Databases Matter: The Data-Driven World

We live in an era where data has become one of the most valuable resources organizations possess (DAMA International, 2017). Consider these realities:

Scale and Complexity: Modern organizations generate data at unprecedented rates. Amazon processes millions of transactions daily. Facebook stores hundreds of billions of photos. Healthcare systems maintain detailed medical histories for entire populations. Managing this scale requires sophisticated database technology.

Real-Time Decision Making: Business decisions that once took weeks of analysis now happen in seconds. Airline reservation systems adjust pricing dynamically based on demand. Banks detect fraudulent transactions as they occur. Emergency services coordinate responses using real-time location data. None of this would be possible without databases that support instant queries across massive datasets (Coronel & Morris, 2017).

Data Integrity and Consistency: When your bank account shows a balance, you trust that the number is correct. When a hospital retrieves your medical records, accuracy is literally life-or-death. Databases enforce rules that maintain consistency even when thousands of updates occur simultaneously, preventing the data corruption that plagued earlier systems.

Business Intelligence and Analytics: Raw data becomes valuable only when transformed into actionable insights. Databases provide the foundation for analyzing

customer behavior, optimizing operations, forecasting trends, and making evidence-based decisions (Connolly & Begg, 2015).

> **Why Study Database Systems?**
>
> Understanding database design and implementation is essential for modern technology professionals because:
>
> 1. **Ubiquitous Technology**: Nearly every software application relies on databases. Web services, mobile apps, enterprise systems, and embedded devices all depend on effective data management.
> 2. **Career Opportunities**: Database skills remain among the most sought-after technical competencies. According to industry surveys, over 60% of developers work with relational databases regularly (Stack Overflow, 2025).
> 3. **System Architecture Foundation**: Database design decisions profoundly impact application performance, scalability, and maintainability. Poor database design creates technical debt that haunts systems for years.
> 4. **Problem-Solving Skills**: Database design teaches systematic analysis, logical thinking, and the ability to model complex real-world scenarios; skills transferable far beyond database work itself.

1.4 Database Management Systems: The Software Foundation

While databases store data, we need sophisticated software to create, access, and manage that data effectively. This software is called a **Database Management System** (DBMS; Silberschatz et al., 2019).

> **Definition: Database Management System (DBMS)**
>
> A **Database Management System (DBMS)** is a software system that enables users to define, create, maintain, and control access to databases. The DBMS serves as an intermediary between users/applications and the physical database, providing a systematic and controlled approach to storing and retrieving information (Elmasri & Navathe, 2016).

Think of the DBMS as a sophisticated librarian for digital information. Just as a skilled librarian knows exactly where every book belongs, helps patrons find information, maintains the catalog system, and ensures materials don't disappear or get damaged, a DBMS manages all aspects of database operations. Users never directly manipulate physical storage. They interact through DBMS, which handles complex details of how and where data is stored.

1.4.1 What Does a DBMS Do?

Modern DBMS software provides extensive capabilities (Date, 2003; Coronel & Morris, 2017):

Data Definition: Through **Data Definition Language (DDL)**, users specify what data to store and how it's structured. You define tables, columns, data types, and constraints that

enforce business rules. When a university creates a STUDENT table with specific columns (Student_ID, Name, Email, Average_Grade), they're using DDL.

Data Manipulation: **Data Manipulation Language (DML)** enables querying and modifying data. Users can insert new records, update existing information, delete obsolete data, and retrieve exactly the information they need through sophisticated queries. The query "*Find all Computer Science students with an average grade above 8*" uses DML.

Data Control: **Data Control Language (DCL)** manages security and access permissions. Database administrators specify who can view, modify, or delete different types of data, implementing organizational security policies directly at the database level.

Transaction Management: DBMS software ensures that complex operations either complete entirely or leave the database unchanged. When you transfer money between bank accounts, the DBMS guarantees that both the debit from one account and the credit to another happen together, never just one without the other.

Concurrent Access Control: Multiple users can access and modify data simultaneously without corrupting the database or seeing inconsistent information. The DBMS coordinates these concurrent operations transparently.

Recovery and Backup: When systems fail (and they always eventually do), the DBMS can restore the database to a consistent state, minimizing data loss.

1.5 The Evolution of Database Systems: From File Systems to Relational Databases

Understanding where we are requires knowing where we've been. Database systems evolved through several distinct generations, each solving problems that troubled its predecessors.

1.5.1 File-Based Systems: The Dark Ages of Data Management

Before database systems existed, organizations stored data in **file-based systems**, collections of independent files managed by individual applications (Connolly & Begg, 2015). Each application maintained its own files using custom formats. This approach created severe problems:

Data Redundancy and Inconsistency: Different applications stored duplicate copies of the same information. A customer's address might appear in sales, billing, and shipping files, each of which may show different values. When the customer moved, updating every copy became a nightmare, and missed updates created inconsistent data that nobody could trust.

Data Isolation: With data scattered across multiple files in different formats, writing applications to retrieve information from multiple sources required complex custom programming. Simple questions like "Which customers purchased products that are now out of stock?" required programmers to write specialized code to extract data from separate inventory and order files.

Application-Data Dependence: File structures were embedded directly in application code. Changing a file format, such as adding a new field, for example, required modifying

every program that accessed that file. This tight coupling made systems rigid and expensive to maintain.

Security Problems: File systems offered minimal security controls. Protecting sensitive information required custom application logic, and mistakes could expose private data.

Concurrent Access Anomalies: When multiple users access the same files simultaneously, updates can be lost, or data can become corrupted. File systems provided no mechanisms for coordinating concurrent access.

These problems became increasingly unacceptable as organizations grew and applications proliferated. The computing industry needed a fundamentally better approach (Elmasri & Navathe, 2016).

1.5.2 The Relational Revolution: E.F. Codd's Breakthrough

In 1970, Dr. Edgar F. Codd, a researcher at IBM, published a paper that would transform computing: "A Relational Model of Data for Large Shared Data Banks" (Codd, 1970). This seminal work introduced the **relational model**, a mathematically rigorous approach to organizing and manipulating data that solved file system problems while providing a solid theoretical foundation.

> **Codd's Key Insights**
>
> **Mathematical Foundation**: Data organization based on set theory and first-order predicate logic, providing rigorous definitions and provable properties.
>
> **Data Independence**: Applications interact with logical data structures (tables) without knowing or caring about physical storage details. Change how data is physically stored without modifying applications.
>
> **Declarative Queries**: Users specify *what* information they want, not *how* to retrieve it. The DBMS determines the optimal execution strategy.
>
> **Data Integrity**: Formal mechanisms (keys, constraints) enforce business rules automatically at the database level, preventing inconsistent data.

Codd's relational model organized data into **relations** (tables) containing **tuples** (rows) with **attributes** (columns). Each table represents an entity type, each row represents a specific entity instance, and each column describes an entity property. Relationships between entities emerge through shared values (foreign keys) rather than physical pointers (Date, 2019).

This elegant simplicity hides remarkable power. The relational model's mathematical foundation enables sophisticated query optimization, automatic integrity enforcement, and formal methods for eliminating data redundancy—capabilities impossible in earlier systems.

1.5.3 Mathematical Theory: Why It Matters

The relational model rests on solid mathematical foundations, specifically set theory and relational algebra (Codd, 1970; Date, 2019). This mathematical basis provides:

Formal Definitions: Precise definitions of concepts such as keys, functional dependencies, and normal forms enable rigorous database design methods.

Query Optimization: Mathematical equivalence rules enable DBMS software to automatically transform queries into equivalent, more efficient forms. Users write declarative queries; the optimizer determines the fastest execution path.

Design Theory: Normalization theory, based on functional dependencies, provides systematic methods for eliminating data redundancy and update anomalies, techniques we'll explore thoroughly in this book.

1.6 ACID Properties and Transaction Reliability

Consistently storing data is only half the challenge. The other half is ensuring that every operation that *modifies* data leaves the database in a trustworthy state, even when things go wrong. Relational databases address this through four fundamental properties known as ACID.

Relational databases follow **ACID** principles that guarantee reliable transaction processing (Härder & Reuter, 1983):

> **ACID Properties of Transactions**
>
> **Atomicity**: A transaction is a logical unit of work that either completes entirely or has no effect whatsoever. When you transfer funds between bank accounts, either both the debit and credit occur, or neither does—never just one without the other.
>
> **Consistency**: Transactions transform the database from one consistent state to another, preserving all defined integrity constraints. A transaction cannot leave the database in an invalid state that violates business rules.
>
> **Isolation**: Concurrent transactions execute as if they were running sequentially, without interfering with each other. Your bank balance inquiry returns an accurate snapshot, even as thousands of other transactions are being processed simultaneously.
>
> **Durability**: Once a transaction completes successfully, its effects persist permanently, surviving system crashes, power failures, and other catastrophes. The DBMS guarantees that committed changes will not be lost.

These properties distinguish database systems from simpler data storage mechanisms. ACID guarantees enable the reliable, mission-critical applications that organizations depend on.

1.7 Components of Database Systems: A Complete Ecosystem

A database system comprises more than just data and software. Understanding the complete environment helps you appreciate how all pieces work together (Elmasri & Navathe, 2016).

1.7.1 Database System Components

Data: The actual information stored in the database—the facts about students, courses, employees, products, transactions, and whatever the database models. Data is the reason the system exists.

Software: The DBMS itself, along with associated utilities for backup, recovery, performance monitoring, and system administration. Examples include SQL Server, MySQL, PostgreSQL, Oracle Database.

Hardware: Physical computing infrastructure—servers, storage systems, networks—that host the DBMS and store data. Modern systems often use distributed hardware across multiple data centers for reliability and performance.

Users: The people who interact with database systems fall into several categories:

- **End Users**: People using applications that access databases (students checking grades, customers placing orders)
- **Application Developers**: Programmers who build software that interacts with databases
- **Database Administrators (DBAs)**: Technical specialists who manage database systems, ensure availability, optimize performance, and handle security
- **Database Designers**: Professionals who create database structures that accurately model organizational requirements—the skills you'll develop in this book

These components interact through well-defined interfaces. The DBMS mediates between users/applications and physical data storage, providing the abstraction layers that make databases usable (Silberschatz et al., 2019).

1.8 Relational Databases Today: Still the Dominant Choice

Given the relational model's age, over 50 years, one might expect newer technologies to have supplanted it. Yet relational databases remain overwhelmingly dominant. Why?

Proven Reliability and Maturity: Decades of development have produced robust, feature-rich DBMS products. Organizations trust relational databases for their most critical applications because they work reliably at scale (Stonebraker & Hellerstein, 2005).

Standardized Query Language: SQL (Structured Query Language) provides a vendor-neutral standard for interacting with relational databases. Skills and applications transfer across different DBMS products with minimal changes (Date & Darwen, 1997).

Rich Ecosystem: Extensive tooling, libraries, training resources, and professional expertise surround relational databases. Organizations can easily find skilled staff and solutions for any relational database challenge.

Performance and Scalability: Modern relational databases handle enormous scale. Systems managing petabytes of data and billions of transactions daily rely on relational technology (Coronel & Morris, 2017).

Strong Consistency Guarantees: ACID properties provide the reliable, consistent behavior that financial, healthcare, government, and other critical applications require. NoSQL alternatives often sacrifice these guarantees for other benefits.

Recent surveys confirm the continued dominance of relational databases. According to Stack Overflow's 2025 Developer Survey, over 60% of professional developers regularly use relational databases, with MySQL, PostgreSQL, and SQL Server ranking as the most popular database technologies (Stack Overflow, 2025).

1.8.1 Relational vs. NoSQL: Choosing the Right Tool

The emergence of NoSQL databases (document stores like MongoDB, key-value stores like Redis, graph databases like Neo4j) doesn't diminish relational databases' importance; it expands options (Sadalage & Fowler, 2012). Different data management approaches suit different problems:

Choose Relational Databases When:

- Data has a clear structure with well-defined relationships
- ACID transactions are critical (financial systems, e-commerce)
- Complex queries across multiple related tables are common
- Data integrity and consistency are paramount
- You need mature, proven technology with extensive tooling

Consider NoSQL When:

- Data is unstructured, or schema changes frequently
- Horizontal scalability across many servers is essential
- Application prioritizes availability over immediate consistency
- Specialized access patterns (graph traversal, document retrieval) dominate

For most business applications, relational databases remain the best choice. This book focuses on relational database design and implementation because these skills apply broadly across industries and remain highly valuable for decades-long careers (Elmasri & Navathe, 2016).

1.9 Our Approach: Learning Through Case Studies

Database design is both a technical discipline and a practical skill. You can understand concepts abstractly, but real mastery requires seeing how principles apply to concrete scenarios. That's why this book uses two comprehensive case studies throughout:

University Management System: Models the academic world you know intimately—students enrolling in courses, professors teaching classes, faculties organizing departments, academic records tracking progress. This familiar domain lets you focus on database concepts without struggling to understand the business requirements.

🏢 **Company/Business Database**: Represents commercial operations—employees working in departments, customers placing orders, products stored in warehouses, suppliers providing inventory. This case study demonstrates how databases support business processes, giving you directly applicable skills for professional work.

1.9.1 Our Pedagogical Approach

Systematic Progression: Each chapter builds methodically on previous concepts. You'll start with conceptual design (understanding business requirements), progress to logical design (creating relational schemas), and continue in physical implementation (writing SQL to create functioning databases).

Dual Case Studies: Every major concept applies to both the University and the Company examples. Seeing the same principles in different contexts deepens understanding and demonstrates their universal applicability.

Theory and Practice: We balance mathematical rigor with practical guidance. You'll understand the theoretical foundations that make databases work while learning the hands-on skills that employers value.

Complete Examples: Rather than fragmentary code snippets, you'll see complete database designs developed from start to finish. This shows not just what the final design looks like, but how to get there systematically.

1.10 Looking Forward: Your Database Design Journey

This book guides you through the complete database development lifecycle:

Part 1 (Foundations) establishes the context. You now understand what databases are and why they matter. Chapter 2 will overview the complete design process, showing how conceptual, logical, and physical design fit together.

Part 2 (Conceptual Design) teaches you to understand and model business requirements. You'll learn Entity-Relationship and Enhanced Entity-Relationship modeling using Chen notation, the most precise approach for capturing organizational data needs. Chapters 3-5 develop these crucial design skills.

Part 3 (Logical Design & Implementation) transforms conceptual models into implementable database structures. You'll master the relational model, learn relational algebra for expressing queries formally, and implement complete databases using SQL. Chapters 6-9 bring your designs to life.

Part 4 (Advanced Topics) covers normalization for optimizing database structures and complete case study implementations that tie everything together.

By the end, you'll possess comprehensive database design and implementation skills, which will serve you throughout your technology career.

1.11 Summary: Databases as the Foundation of Modern Systems

Databases have become the invisible infrastructure supporting modern society. From the university tracking your academic progress to the business processing your online orders, sophisticated database systems enable the applications we depend on daily.

Key Insights: Why Databases Matter

Databases provide structured, reliable storage for the massive amounts of data modern organizations generate and depend upon (Elmasri & Navathe, 2016).

Database Management Systems (DBMS) offer sophisticated capabilities for defining, manipulating, and controlling access to data that are impossible with simple file systems (Date, 2003).

The relational model's mathematical foundation provides rigorous design principles and enables powerful query optimization, explaining its 50+ year dominance (Codd, 1970; Date, 2019).

ACID properties guarantee reliable transaction processing, making databases suitable for mission-critical applications where consistency and durability are essential (Härder & Reuter, 1983).

Relational databases remain the dominant technology choice, with over 60% of developers regularly using them, due to their proven reliability, standardization, and rich ecosystem (Stack Overflow, 2025).

This chapter established the foundation. You understand what databases are, why organizations need them, how database systems evolved, and why the relational model remains dominant. Chapter 2 will overview the complete database design process, showing how the systematic approach you'll learn in subsequent chapters transforms business requirements into functioning database systems.

The journey from requirements to implementation requires careful methodology. Let's explore that process next.

■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■

Review Questions

Conceptual Understanding Questions

1.1. Definition and Purpose: Define what a database is and explain how it differs from a simple collection of files. What are the key characteristics that make a database more than just stored data?

1.2. Database vs. DBMS: Explain the distinction between a database and a Database Management System (DBMS). Why is this distinction important for understanding database systems?

1.3. ACID Properties: Describe each of the ACID properties (Atomicity, Consistency, Isolation, Durability) and explain why they are critical for reliable database systems. Provide a real-world example of each property in which violating it would cause problems.

1.4. Historical Significance: Dr. E.F. Codd's 1970 paper introduced the relational model. What fundamental problems with earlier database systems did the relational model solve? Why was mathematical rigor important?

1.5. Database System Components: List and explain the four main components of a database system (Data, Software, Hardware, Users). How do these components interact to provide database functionality?

Application and Analysis Questions

1.6. File Systems vs. Databases: A small business currently stores customer information in Excel spreadsheets. Identify at least five specific problems they might encounter as they grow, and explain how a database system would address each problem.

1.7. DBMS Capabilities: Explain the difference between DDL (Data Definition Language), DML (Data Manipulation Language), and DCL (Data Control Language). Provide specific examples of operations in each category.

1.8. Real-World Applications: Choose two very different domains (e.g., healthcare and e-commerce) and describe what types of data each would store in its databases and why database reliability is critical for each.

1.9. Transaction Scenarios: Consider transferring $500 from your checking account to your savings account. Walk through this operation explaining how each ACID property ensures the transaction completes correctly.

1.10. Shared Data Benefits: Explain how the "shared" characteristic of databases benefits organizations. What problems arise when different departments maintain separate copies of customer data?

Critical Thinking Questions

1.11. Relational vs. NoSQL Decision: You're designing a system to store and query a social network's connections (friends, followers). Compare using a relational database versus a graph database (a type of NoSQL). What factors would influence your decision?

1.12. DBMS Abstraction Layers: Explain why the abstraction that a DBMS provides between applications and physical data storage is valuable. What would happen if applications accessed physical storage directly?

1.13. Evolution Assessment: The chapter mentions that relational databases have dominated for 50+ years. Given the rapid technological change in other areas, why have relational databases remained dominant? Will they still be relevant in another 20 years?

1.14. Security and Access Control: Modern databases store sensitive information (medical records, financial data, personal information). How do database systems help protect this data compared to file-based systems?

1.15. Concurrent Access: Explain the problems that arise when multiple users simultaneously access and modify the same data. How do database systems prevent these problems?

Practical Application Questions

1.16. University Database Scope: Consider the University Management System case study mentioned in this chapter. List at least 10 different types of data a university database would need to store and explain the relationships between them.

1.17. Company Database Requirements: For the Company/Business Database case study, identify at least 8 different types of data that would be essential and explain why each is necessary for business operations.

1.18. Technology Selection: You're advising a startup that needs to choose database technology. They'll store user profiles, transaction histories, and product catalogs. What questions would you ask before recommending relational vs. NoSQL, and why?

1.19. Data Independence Benefits: Explain data independence and provide a specific scenario where it provides value. For example, what happens when you need to add a new field to customer records?

1.20. Career Preparation: The chapter mentions that over 60% of developers use relational databases. Based on this chapter, what specific database skills should you prioritize learning to prepare for professional work?

Integration and Synthesis Questions

1.21. Design Process Preview: Based on what you learned in this chapter, what do you expect the database design process to involve? What kinds of activities and decisions will be necessary to create a good database?

1.22. Case Study Connections: Why does this book use both a University and a Company case study? How might learning from two different domains strengthen your database design skills compared to studying just one?

1.23. Historical Perspective: Trace the evolution from file-based systems to modern databases. What were the key innovations at each stage, and what problems did they solve?

1.24. Theory Meets Practice: The chapter emphasizes that the relational model has a mathematical foundation. Why does this theoretical rigor matter for practical database work? Give specific examples.

1.25. Looking Ahead: Based on this introduction, what aspects of database design seem most challenging? What skills from this chapter will be essential as you progress through conceptual design (Part 2) and implementation (Part 3)?

Advanced Thinking Questions

1.26. Database as Business Asset: Explain how a well-designed database becomes a strategic asset for organizations. What makes database design important beyond just making applications work?

1.27. Professional Roles: The chapter mentioned different types of database users, including end users, developers, DBAs, and designers. How do the perspectives and priorities of these roles differ? How might this impact design decisions?

1.28. Integration with Chapter 2: This chapter explained what databases are and why they matter. What topics do you expect Chapter 2 (Database Design Process) to cover? How will understanding the design process build on the foundations established here?

References and Recommended Readings

Primary References

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM, 13*(6), 377-387.

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of Database Systems* (7th ed.). Pearson.

Date, C. J. (2003). *An Introduction to Database Systems* (8th ed.). Addison-Wesley.

Historical and Theoretical Foundations

Härder, T., & Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys, 15*(4), 287-317.

Date, C. J. (2019). *Database Design and Relational Theory: Normal Forms and All That Jazz* (2nd ed.). O'Reilly Media.

Stonebraker, M., & Hellerstein, J. M. (Eds.). (2005). *Readings in Database Systems* (4th ed.). MIT Press.

Contemporary Practice

Coronel, C., & Morris, S. (2017). *Database Systems: Design, Implementation, & Management* (12th ed.). Cengage Learning.

Connolly, T. M., & Begg, C. E. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson.

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database System Concepts* (7th ed.). McGraw-Hill.

Database Management Systems

Date, C. J., & Darwen, H. (1997). *A Guide to the SQL Standard* (4th ed.). Addison-Wesley.

Ramakrishnan, R., & Gehrke, J. (2003). *Database Management Systems* (3rd ed.). McGraw-Hill.

NoSQL and Alternative Approaches

Sadalage, P. J., & Fowler, M. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley.

Strauch, C. (2011). NoSQL databases. *Lecture Notes, Stuttgart Media University*.

Professional Standards and Practice

DAMA International. (2017). *DAMA-DMBOK: Data Management Body of Knowledge* (2nd ed.). Technics Publications.

Contemporary Statistics and Trends

Stack Overflow. (2025). *Stack Overflow Developer Survey 2025*. Retrieved from https://survey.stackoverflow.co/2025/

Chapter 2

2. Database Design Process

2.1 Introduction: From Chaos to Structure

Imagine you're meeting with the Head of Academic Affairs at a university. She explains enthusiastically: "We need a new database system! Right now, we have spreadsheets everywhere—the registrar has student records, each department maintains their own course information, professors keep grade books in different formats, and nobody can generate a complete transcript without calling three different offices. We need something better."

You nod encouragingly, notebook ready. "Tell me more about how students register for courses," you prompt.

"Well," she begins, "the Teaching Committee prepares a list of available courses each semester. Students have core courses that are mandatory for their faculty; everyone must take those. Then there are major electives, which are technical courses since we're a university for information science and technology, and minor electives from other fields. Then there are major electives, which are technical courses, as the university specializes in information science and technology, and minor electives from other fields. Students can contact professors directly or attend introductory lectures to learn more about the electives before making their decisions. They then complete registration forms and submit them to the Student Affairs Office. Students pay registration fees, although some receive full or partial scholarships. Students from single-parent households may be eligible for reduced fees. Of course, some courses have prerequisites that must be completed first. Each course includes lectures and tutorials, and some also include laboratory sessions.

Forty-five minutes later, you have pages of notes filled with partially-understood requirements, implicit assumptions, and questions you forgot to ask. How do you transform this conversational, informal information—with its many unstated details—into a precise, complete, functioning database system?

This scenario illustrates the central challenge of database design: bridging the gap between what stakeholders need (expressed in everyday language filled with assumptions and ambiguity) and what computers require (precise, unambiguous, formally structured specifications). The database design process provides a systematic methodology for reliably and consistently implementing this transformation (Connolly & Begg, 2015).

Database design isn't a single activity. It's a multi-phase journey with distinct stages, each building on the previous one. Do it right, and you create systems that serve organizations reliably for decades. Do it wrong, and you create technical debt that haunts every subsequent modification, slowing development and increasing costs exponentially (Hoffer et al., 2016).

In this chapter, we will explore the complete database development lifecycle. You will understand how requirements collection transforms into conceptual designs, how conceptual designs become logical schemas, and how logical schemas translate into physical implementations. More importantly, you will learn the practical skills for engaging stakeholders, discovering hidden requirements, and validating your designs before committing to implementation.

2.2 Understanding the Database Development Lifecycle (DDLC)

Database development follows a structured lifecycle consisting of distinct phases, each with specific activities, deliverables, and quality criteria (Elmasri & Navathe, 2016). While different sources identify varying numbers of phases, the fundamental progression remains consistent: from understanding requirements to creating conceptual models, transforming those models into logical structures, and finally implementing physical databases.

> **The Database Development Lifecycle (DDLC)**
>
> **The Database Development Lifecycle (DDLC) is the systematic process of creating, implementing, and maintaining database systems, comprising distinct phases that transform organizational requirements into operational databases.** The DDLC is an integral part of the broader Software Development Life Cycle (SDLC), but focuses specifically on data management aspects (Connolly & Begg, 2015).

The database development lifecycle typically includes the following major phases:

Requirements Collection and Analysis: Understanding what data the organization needs to store, how that data will be used, and what business rules govern its behavior. This phase produces detailed requirements specifications that guide all subsequent design work (Teorey et al., 2011).

Conceptual Design: Creating a high-level, abstract model of the data requirements using notation systems like Entity-Relationship (ER) diagrams. Conceptual design is *DBMS-independent*. It describes **what** data to store and how pieces of data relate, without specifying **how** any particular database management system will implement it (Elmasri & Navathe, 2016).

Logical Design: Transforming the conceptual model into a structured format suitable for implementation in a specific category of DBMS. In relational databases, this means creating a relational schema with tables, columns, keys, and constraints. Logical design begins to consider DBMS characteristics but remains relatively implementation-independent (Date, 2003).

Physical Design: Making concrete implementation decisions for a specific DBMS product—choosing data types, defining indexes, partitioning tables, allocating storage, and

optimizing for performance. Physical design transforms logical schemas into actual database structures (Coronel & Morris, 2017).

Implementation and Loading: Creating the database using DDL (Data Definition Language) statements, loading data from existing sources, and verifying that everything works as designed (Silberschatz et al., 2019).

Testing and Validation: Systematically verifying that the implemented database meets all requirements, handles all specified transactions correctly, and performs adequately under expected loads (Connolly & Begg, 2015).

Operation and Maintenance: Using the database in production, monitoring performance, backing up data, managing security, and modifying the structure as requirements evolve (Hoffer et al., 2016).

2.2.1 The DBMS-Independent to DBMS-Specific Transition

A crucial distinction divides these phases into two categories:

DBMS-Independent Phases (Requirements Collection, Conceptual Design): These phases focus on understanding and modeling organizational data needs without considering any specific database technology. The conceptual ER diagram you create applies equally whether you later implement it in MySQL, Oracle, SQL Server, or PostgreSQL. This independence provides flexibility and ensures designs reflect business requirements rather than technology limitations (Elmasri & Navathe, 2016).

DBMS-Specific Phases (Logical Design, Physical Design, Implementation): These phases increasingly commit to specific database technologies, ultimately producing schemas that leverage particular DBMS features and capabilities. While logical design maintains some flexibility, physical design makes concrete choices that optimize for specific software (Date, 2003).

Understanding this transition helps you make appropriate design decisions at each phase. Conceptual design asks, "*What data exists and how does it relate?*" Logical design asks, "*How do we structure this data in tables?*" Physical design asks, "*How do we implement these tables efficiently in SQL Server?*"

The database design process follows established methodology first formalized by Elmasri and Navathe (2016). Figure 2.1 presents this lifecycle, adapted to our pedagogical focus on practical relational database design, with emphasis on the transition from DBMS-independent to DBMS-specific.

2.3 Phase 1: Requirements Collection and Analysis

Requirements collection and analysis is arguably the most critical phase in database development; everything else builds on the foundation established here (Connolly & Begg, 2015). Get requirements wrong, and you'll build the wrong database correctly. Get requirements incomplete, and you'll face expensive modifications later. Get requirements ambiguous, and different stakeholders will interpret them differently, leading to conflicts.

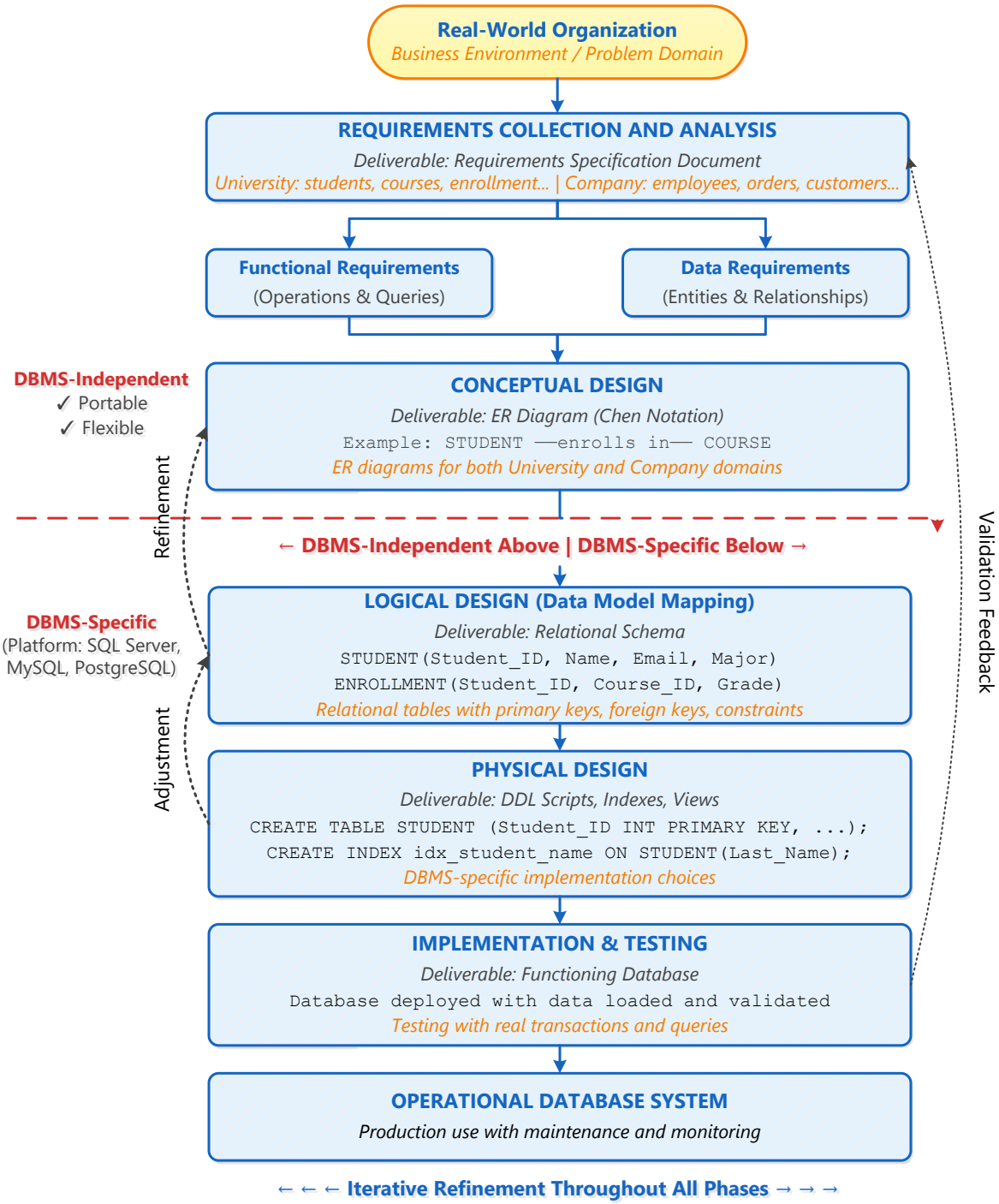


Figure 2.1: Database Design Process Lifecycle (Adapted from Elmasri, R., & Navathe, S. B. (2016). Fundamentals of Database Systems (7th ed.), Figure 7.1, p. 201. Modifications include simplified terminology, emphasis on the relational data track, and integration of University and Company case study examples.)

Note: Table schemas shown are illustrative; complete relational schemas for both case studies are presented in Section 2.9.

2.3.1 The Nature of Requirements

Database requirements fall into two primary categories:

Data Requirements specify **what** information the database must store. For a university database: "*The system must maintain information about students, courses, professors, enrollments, grades, departments, and degree programs*." Data requirements identify **entities** (things of interest), **attributes** (properties of those things), and **relationships** (connections between things) (Elmasri & Navathe, 2016).

Functional Requirements specify **how** users will interact with the data and what operations they need to perform. For a university database: "Students must be able to view available courses, register for courses (subject to prerequisites), and access their unofficial transcripts. Professors must record grades. The Head of Academic Affairs must be able to access students' complete academic histories. The registrar must generate official transcripts and enrollment reports" (Teorey et al., 2011).

Both categories matter. Data requirements drive conceptual design; they determine the ER diagram's structure. Functional requirements influence database design decisions, validate completeness (if the design can't support required operations, it's incomplete), and drive application development that uses the database (Connolly & Begg, 2015).

2.3.2 Discovery Techniques: Extracting Knowledge from Stakeholders

Most stakeholders are not database experts. They think in terms of their daily work, not in entities and relationships. Your job is to extract their implicit knowledge about data and transform it into explicit specifications. Several techniques prove effective:

Interviews and Conversations: Direct one-on-one or small-group discussions with stakeholders, asking open-ended questions that encourage detailed explanations. "Walk me through a typical day," "What information do you need to do your job?" and "What happens when..." questions reveal processes and data needs (Hoffer et al., 2016).

Observation: Watching people actually do their work reveals details they might not mention in interviews. You see what forms they fill out, what reports they generate, what information they look up, and what decisions require what data (Connolly & Begg, 2015).

Document Analysis: Examining existing forms, reports, spreadsheets, and legacy systems shows what data the organization currently tracks and how it's structured. While you're designing a *new* system, understanding the *current* system provides valuable context (Teorey et al., 2011).

Prototyping and Examples: Showing stakeholders sketches or mockups of potential designs helps them articulate what's missing or incorrect. "Is this what you meant?" Conversations reveal misunderstandings early when they're cheap to fix (Elmasri & Navathe, 2016).

2.3.3 The Challenge of Non-Technical Clients

A university registrar will not say, "I need a many-to-many relationship between STUDENT and COURSE entities, with an associative entity ENROLLMENT containing grade and semester attributes." Instead, she will say something like "Students take multiple classes, and classes have multiple students, and we need to track what grade each student got in

each class each semester." Your task is to translate conversational descriptions into precise data models (Connolly & Begg, 2015).

This translation requires:

Active Listening: Hearing not just what stakeholders say explicitly, but recognizing implications and assumptions they don't articulate. When someone says "customers place orders," they're implicitly describing entities (CUSTOMER, ORDER) and a relationship (places) with specific cardinality (Hoffer et al., 2016).

Clarifying Questions: Asking follow-up questions that expose hidden details. "Can a student register for the same course multiple times in different semesters?" "What happens if a student withdraws from a course midway through the semester?" "Are lab sections separate entities or part of the parent course?" Each question reveals additional requirements (Teorey et al., 2011).

Vocabulary Management: Organizations use domain-specific terminology that means different things to different people. "Course" might mean the abstract concept (CS101 Introduction to Programming) or a specific offering (CS101 Section 3, Fall 2025). Establishing clear, consistent terminology prevents confusion (Date, 2003).

Validating Understanding: Periodically summarizing what you've learned and checking with stakeholders: "So if I understand correctly, prerequisites are always between courses (not sections), and a student must pass all prerequisites before registering. Is that right?" This prevents misunderstandings from arising immediately (Connolly & Begg, 2015).

2.3.4 Discovering Business Rules

Business rules are policies, regulations, constraints, and procedures that govern how an organization operates. They're critical for database design because they determine what constraints and relationships your database must enforce (Elmasri & Navathe, 2016).

Consider these examples from a university:

- "Students must complete 180 or 240 credits to graduate."
- "A student cannot register for more than 30 credits per semester without special permission."
- "Prerequisites must be completed with a grade of 8 or better."
- "Professors cannot teach more than four courses per semester."
- "Graduate students can also serve as teaching assistants."

Some business rules translate directly into database constraints (maximum credits per semester). Others require application logic (checking prerequisites during registration). Still others affect the data model itself (students who are also teaching assistants might indicate a need for role-based modeling) (Connolly & Begg, 2015).

2.3.5 Critical Success Factors for Requirements Analysis

Successful requirements analysis requires thoroughness, precision, and validation:

- **Thoroughness**: Interview representatives from all stakeholder groups. Missing requirements discovered late are exponentially more expensive to accommodate (Hoffer et al., 2016).
- **Precision**: Vague requirements like "the system should be flexible" or "we need good reporting" don't provide enough detail for design. Push for specifics (Teorey et al., 2011).
- **Validation**: Review requirements with stakeholders before proceeding to design. Misunderstandings caught here cost hours; misunderstandings caught after implementation cost months (Connolly & Begg, 2015).

The output of this phase is a **requirements specification document** that clearly describes all data to be stored, the operations to be supported, the business rules to be enforced, and the quality constraints (performance, security, reliability) the system must meet. This document becomes the contract between you and stakeholders; the standard against which you'll evaluate all subsequent design work (Elmasri & Navathe, 2016).

2.4 Phase 2: Conceptual Design

Conceptual design transforms requirements specifications into formal models that precisely capture organizational data needs while remaining independent of any specific DBMS (Date, 2003). This phase produces the **conceptual schema**—typically an Entity-Relationship (ER) diagram using Chen notation—that serves as the blueprint for all subsequent design work.

2.4.1 Purpose and Characteristics of Conceptual Design

Conceptual design serves several crucial purposes:

Communication: A well-constructed ER diagram communicates complex data structures to both technical and non-technical stakeholders. Business users can verify that the model captures their understanding; developers can see what needs to be built (Connolly & Begg, 2015).

Abstraction: By modeling concepts at a high level, conceptual design separates what we store from how we store it. This abstraction prevents premature commitment to implementation details that might constrain the design inappropriately (Elmasri & Navathe, 2016).

Validation: Creating a formal model reveals inconsistencies, ambiguities, and omissions in requirements. The discipline of defining entities, attributes, and relationships exposes gaps in understanding (Teorey et al., 2011).

Foundation: The conceptual schema becomes the authoritative specification from which logical and physical designs derive. Changes to the conceptual model propagate to all subsequent phases (Date, 2003).

Effective conceptual design exhibits specific quality characteristics:

DBMS Independence: The conceptual model shouldn't assume any particular database technology. Whether you later implement in MySQL, Oracle, SQL Server, or PostgreSQL,

the same ER diagram applies. This independence ensures designs reflect business reality, not technology limitations (Elmasri & Navathe, 2016).

Completeness: Every entity, attribute, relationship, and constraint identified during requirements analysis must appear in the conceptual model. Incompleteness creates gaps that surface as missing functionality later (Connolly & Begg, 2015).

Clarity: Stakeholders must be able to understand and validate the model. Overly complex diagrams with hundreds of entities obscure rather than clarify (Hoffer et al., 2016).

Accuracy: The model must faithfully represent actual organizational data structures and relationships. Inaccuracies create databases that don't match how the organization really operates (Teorey et al., 2011).

2.4.2 From Requirements to ER Model

Transforming requirements into ER diagrams follows systematic patterns:

Identify Entities: Nouns in requirements specifications often indicate entities. "Students enroll in courses taught by professors" suggests at least three entities: STUDENT, COURSE, PROFESSOR (Elmasri & Navathe, 2016).

Determine Attributes: Descriptive information about entities becomes attributes. "Students have names, student IDs, email addresses, and admission dates" identifies four attributes for STUDENT. The question "What information do we need to track about X?" reveals attributes (Date, 2003).

Establish Relationships: Verbs connecting entities typically indicate relationships. "Students *enroll in* courses" describes an *enrollment* relationship between STUDENT and COURSE entities. "Professors *teach* courses" describes a *teaches* relationship between PROFESSOR and COURSE (Connolly & Begg, 2015).

Specify Cardinality: Business rules determine how many instances of one entity can relate to instances of another. "Each student can enroll in multiple courses" and "Each course can have multiple students" indicate a many-to-many relationship. "Each department has exactly one department chair" indicates a one-to-one relationship (Teorey et al., 2011).

Identify Keys: Determine which attributes uniquely identify entity instances. Student_ID uniquely identifies students; Course_ID uniquely identifies courses. These become primary keys (Elmasri & Navathe, 2016).

Capture Constraints: Business rules translate into constraints on the model. "Students cannot register for more than 30 credits per semester" becomes a constraint on the ENROLLMENT relationship (Date, 2003).

This transformation isn't purely mechanical. It requires judgment, domain knowledge, and often multiple iterations as you refine your understanding (Connolly & Begg, 2015).

2.4.3 Why Chen Notation?

This book uses **Chen notation** for ER diagrams because it provides the most precise, unambiguous representation of database concepts (Elmasri & Navathe, 2016). While other notations exist (crow's foot, UML), Chen notation:

- Clearly distinguishes entities (rectangles), attributes (ovals), and relationships (diamonds)
- Explicitly represents relationship cardinality using notation on relationship lines
- Shows participation constraints (total/partial) directly in the diagram
- Handles complex scenarios (weak entities, ternary relationships, multivalued attributes) with formal precision
- Has been the standard in academic database education for decades, making it the lingua franca of database design

Chapters 4 and 5 will thoroughly cover Chen notation, providing the tools you need to create professional-quality ER and EER (Enhanced Entity-Relationship) diagrams.

2.4.4 The Role of Iteration and Refinement

Conceptual design rarely succeeds on the first pass. Instead, you will create initial models, validate them with stakeholders, identify omissions or errors, refine the models, and repeat the process (Connolly & Begg, 2015). This iterative refinement is healthy and expected; it's far better to discover problems in the conceptual design phase (where fixing them means redrawing diagrams) than in the implementation phase (where fixing them means restructuring databases and rewriting application code) (Teorey et al., 2011).

Common refinements include:

- Adding entities that were initially missed
- Splitting complex entities into separate entities with relationships
- Adjusting relationship cardinalities based on new understanding
- Adding attributes to capture additional information needs
- Recognizing specialized subtypes that warrant enhanced modeling (EER)

The conceptual design phase concludes when stakeholders agree that the ER diagram accurately, completely, and clearly represents organizational data requirements, and when you're confident the model provides a solid foundation for logical design (Elmasri & Navathe, 2016).

2.5 Phase 3: Logical Design

Logical design transforms the conceptual schema into structured formats suitable for implementation in a specific category of DBMS—most commonly, relational database management systems (Connolly & Begg, 2015). This phase produces the **logical schema**:

a detailed specification of tables, columns, keys, constraints, and relationships that represents the ER model in relational terms.

2.5.1 The Transformation from ER to Relational Model

Logical design follows systematic transformation rules that convert ER components into relational structures (Elmasri & Navathe, 2016):

Entities Become Tables: Each entity in the ER diagram becomes a table in the logical schema. The entity STUDENT becomes the table STUDENT. Entity attributes become table columns (Date, 2003).

Relationships Become Foreign Keys or Tables: How relationships translate depends on their cardinality:

- One-to-many relationships: Add a foreign key to the "many" side
- Many-to-many relationships: Create an associative/junction table containing foreign keys to both participating entities
- One-to-one relationships: Add a foreign key to either side (design choice) or merge into a single table

Keys Transfer: Primary keys from the ER model become table primary keys. Foreign keys implement relationships between tables (Teorey et al., 2011).

Constraints Formalize Business Rules: Business rules identified in requirements and captured in the ER model are implemented as table constraints—primary key constraints, foreign key constraints, unique constraints, check constraints, and not-null constraints (Connolly & Begg, 2015).

These transformation rules are precisely defined and will be covered in detail in Chapters 6 and 7. The key insight here is that logical design is *systematic*, not arbitrary. There are correct and incorrect ways to transform ER models into relational schemas (Elmasri & Navathe, 2016).

2.5.2 Normalization: Refining the Logical Structure

Logical design includes **normalization**, a mathematically rigorous process for eliminating data redundancy and update anomalies (Date, 2003). Normalization applies formal rules (normal forms) that progressively restructure tables to ensure:

- No redundant data storage (the same fact appears in only one place)
- No update anomalies (changes in one place automatically reflect everywhere)
- No insertion anomalies (you can add data without having unrelated data)
- No deletion anomalies (removing data doesn't accidentally delete other data)

Well-normalized databases are easier to maintain, more reliable, and less prone to inconsistencies. Chapter 10 will cover normalization theory and practice (Elmasri & Navathe, 2016).

2.5.3 Maintaining DBMS Category Independence

While logical design considers DBMS characteristics, it still maintains relative implementation independence. You're designing for "a relational DBMS" rather than "SQL Server specifically" or "MySQL specifically." The logical schema specifies tables, columns, and relationships using generic relational concepts that apply to all relational DBMS products (Connolly & Begg, 2015).

This approach provides flexibility. If organizational requirements change and you need to switch from MySQL to PostgreSQL, or if you need to deploy the same logical design across multiple DBMS products, the logical schema remains largely unchanged. Only physical design commitments require modification (Date, 2003).

Key Insight: Why Correct Logical Design Matters

Getting logical design wrong is expensive. If your ER diagram is incorrect, you discover the problem when validating with stakeholders (cheap fix: redraw the diagram). If your logical schema is incorrect but your ER diagram is correct, you discover the problem during implementation (moderate cost: redo transformation, modify SQL scripts). If your logical schema is incorrect *and* you don't discover this until after implementation, when users report that the system doesn't work as expected, the cost becomes enormous: restructuring the database, migrating data, and modifying all application code that interacts with the database (Teorey et al., 2011).

This is why we emphasize getting conceptual and logical design right **before** committing to physical implementation. The time invested in careful design saves exponentially more time in maintenance and rework (Connolly & Begg, 2015).

2.6 Phase 4: Physical Design and Implementation

While the database development lifecycle treats physical design and implementation as separate phases, in practice, they are closely connected and often carried out together. Physical design decisions are immediately reflected in DDL statements, and implementation results often inform adjustments to those choices. Physical design makes concrete decisions about how to implement the logical schema in a specific DBMS product, optimizing for performance, storage efficiency, and operational requirements (Coronel & Morris, 2017). This phase produces the **physical schema** or **internal schema** — the actual database structure as it exists in the DBMS.

Physical design makes concrete decisions about how to implement the logical schema in a specific DBMS product, optimizing for performance, storage efficiency, and operational requirements (Coronel & Morris, 2017). This phase produces the **physical schema** or **internal schema**—the actual database structure as it exists in the DBMS.

2.6.1 Physical Design Decisions

Physical design involves numerous specific choices:

Data Types: Selecting appropriate data types for each column. Should Student_Name be VARCHAR(50), VARCHAR(100), or NVARCHAR(100)? Should Age be TINYINT or INT?

These decisions affect storage requirements and query performance (Silberschatz et al., 2019).

Indexes: Determining which columns to index for faster query performance. Indexes dramatically speed up data retrieval but slow down insert/update operations and consume storage. The challenge is identifying the right balance (Elmasri & Navathe, 2016).

Storage Structures: Deciding how tables are physically stored—file organization methods, clustering approaches, partitioning strategies for very large tables. These choices significantly impact query performance (Date, 2003).

Views: Creating views that simplify complex queries, provide security by limiting data access, or present data in forms convenient for specific applications (Connolly & Begg, 2015).

Security: Implementing authentication, authorization, and access control. Who can view which data? Who can modify what? How do we audit data access? (Coronel & Morris, 2017)

Backup and Recovery: Planning backup schedules, retention policies, and recovery procedures. How frequently do we back up? How quickly must we recover after failures? (Silberschatz et al., 2019)

Physical design requires intimate knowledge of the chosen DBMS product because different products offer different features, use different optimization strategies, and have different performance characteristics (Elmasri & Navathe, 2016).

2.6.2 Implementation: Creating the Physical Database

Implementation translates physical design specifications into actual database objects using SQL Data Definition Language (DDL) statements (Date, 2003):

- **CREATE DATABASE** statements establish the database
- **CREATE TABLE** statements define table structures with columns, data types, and constraints
- **CREATE INDEX** statements add indexes for performance
- **CREATE VIEW** statements define views
- **GRANT** and **REVOKE** statements configure security permissions

Chapter 9 covers SQL DDL comprehensively, showing exactly how to implement physical databases from logical designs (Connolly & Begg, 2015).

2.6.3 Data Loading and Conversion

After creating the database structure, you must populate it with data. For new systems, this might involve importing data from existing spreadsheets or legacy databases, or performing manual data entry. For replacement systems, this involves **data conversion**—extracting data from the old system, transforming it to match the new schema, and loading it into the new database (Coronel & Morris, 2017).

Data conversion is often more complex than it seems. Legacy data is frequently inconsistent, incomplete, or violates constraints enforced by the new design. You might discover that the old system allowed students to register for courses without completing prerequisites (violating business rules the new system must enforce). Cleaning and validating data during conversion is essential (Silberschatz et al., 2019).

2.6.4 Testing and Validation

Before deploying a new database to production, systematic testing verifies that:

- All required data is present and accessible
- All business rules are correctly enforced by constraints
- All required queries execute successfully
- Performance meets requirements
- Security restrictions work as designed
- Backup and recovery procedures function correctly

Testing should include not just "happy path" scenarios (everything works as expected) but also error cases (what happens when users try to violate constraints, when systems fail, when loads exceed expectations) (Connolly & Begg, 2015).

2.7 The Iterative Nature of Database Design

The database development lifecycle diagram suggests a linear progression: requirements → conceptual → logical → physical → implementation. Reality is messier and more iterative (Hoffer et al., 2016).

2.7.1 Why Iteration Occurs

Requirements Evolve: As stakeholders review conceptual designs, they recall additional requirements or realize that what they initially asked for do not fully reflect their needs. "Oh, we also need to track student minors, not just majors," emerges during ER diagram review (Connolly & Begg, 2015).

Design Reveals Problems: Creating the conceptual model exposes ambiguities and inconsistencies in requirements. "Wait, if adjunct professors aren't employees, how do we track them?" This question sends you back to requirements collection (Teorey et al., 2011).

Validation Identifies Errors: When you show stakeholders the logical schema or a prototype implementation, they recognize misunderstandings. "No, grades aren't just numbers; we also record flags for incomplete or withdrawn." Back to conceptual design to add these details (Elmasri & Navathe, 2016).

Implementation Constraints Influence Design: Physical design sometimes reveals that conceptual or logical design decisions create performance problems or implementation difficulties. You might need to revisit earlier design choices (Date, 2003).

2.7.2 Managing Iteration Productively

Iteration is natural and healthy, but it must be controlled. Unmanaged iteration leads to "analysis paralysis"—endless refinement that never reaches implementation (Hoffer et al., 2016). Productive iteration involves:

Version Control: Maintaining clear versions of requirements, conceptual models, and logical schemas. When changes occur, you know exactly what changed and why (Connolly & Begg, 2015).

Change Management: Establishing procedures for evaluating proposed changes, assessing their impact, and deciding whether to incorporate them now or defer them to future releases (Teorey et al., 2011).

Progress Milestones: Setting clear criteria for completing each phase. "The conceptual design is complete when all stakeholders have reviewed and approved the ER diagram, and no major entities or relationships are missing." This prevents indefinite revision (Elmasri & Navathe, 2016).

Scope Management: Distinguishing "must-have" requirements that are essential for initial release from "nice-to-have" features that can be added later. This focus prevents feature creep that delays delivery indefinitely (Hoffer et al., 2016).

The goal is **disciplined flexibility**—accommodating necessary changes while maintaining steady progress toward implementation (Connolly & Begg, 2015).

2.8 Stakeholder Engagement: Working with Non-Technical Clients

Database designers serve as translators between two worlds: the business domain (where stakeholders think in terms of their daily work) and the technical domain (where developers think in terms of tables, keys, and SQL) (Connolly & Begg, 2015). Success requires communication skills as much as technical skills.

2.8.1 Understanding Stakeholder Perspectives

Different stakeholders have different priorities and different concerns (Hoffer et al., 2016):

End Users care about: Can I do my job efficiently? Is the system easy to use? Does it give me the information I need? They don't care about normalization or indexing strategies—they care about whether the system helps or hinders their work (Elmasri & Navathe, 2016).

Business Managers care about: Does this system support our business processes? Does it improve productivity? Can we get the reports and analytics we need for decision-making? They think in terms of business value rather than technical elegance (Teorey et al., 2011).

IT Operations cares about: Is this system reliable? Can we back it up and restore it? Does it have reasonable performance characteristics? Will it integrate with our existing infrastructure? They focus on operational concerns (Date, 2003).

Executives care about: What's the return on investment? When will it be ready? What's the risk? They need a high-level understanding without technical details (Connolly & Begg, 2015).

Your challenge is to communicate appropriately with each audience and present information at the right level of abstraction for their needs (Hoffer et al., 2016).

2.8.2 Effective Communication Strategies

Use Their Language: When talking with business stakeholders, use business terminology, not database jargon. Don't say "We'll create an associative entity to resolve the many-to-many relationship." Say "We'll track which students are in which courses, including when they registered and what grade they received" (Connolly & Begg, 2015).

Visual Communication: ER diagrams are powerful communication tools *when explained* properly. Walk stakeholders through the diagram, explaining what each box represents, what the connecting lines mean, and how it reflects their business processes. Visual models are often more accessible than text descriptions (Elmasri & Navathe, 2016).

Scenarios and Examples: Abstract discussions confuse people. Concrete examples clarify. Instead of "What's the cardinality of the department-employee relationship?" ask "Can an employee work in multiple departments simultaneously? Can a department exist without any employees?" Real scenarios make concepts tangible (Teorey et al., 2011).

Iterative Validation: Don't just present finished designs for approval. Show work in progress, gather feedback early and often, and incorporate suggestions. This collaborative approach builds stakeholder buy-in and catches problems early (Connolly & Begg, 2015).

Documenting Decisions: Keep clear records of what was decided, who decided it, and why. When questions arise later (and they will), documentation prevents "I thought we agreed on..." disputes (Hoffer et al., 2016).

2.8.3 Success Principles for Stakeholder Engagement

Effective stakeholder engagement requires empathy, patience, and clarity:

- **Listen more than you talk**: Stakeholders know their domain better than you do. Your job is to learn from them, not lecture them (Connolly & Begg, 2015).
- **Validate understanding constantly**: Repeat back what you've heard in your own words and ask, "Is that correct?" This prevents misunderstandings from arising immediately (Teorey et al., 2011).
- **Respect expertise**: Business stakeholders are experts in their domain, even if they're not database experts. Value their knowledge (Elmasri & Navathe, 2016).

- **Be honest about constraints**: If stakeholder requests create technical problems or conflicts with other requirements, explain this directly rather than agreeing to impossible commitments (Hoffer et al., 2016).

2.9 From Requirements to Reality: University and Company Examples

Throughout this book, we'll use two comprehensive case studies to illustrate database design principles: a **University Management System** and a **Company/Business Database**. The following sections provide a preview of how the design process applies to each.

University Management System

Requirements Collection: You interview various stakeholders, the Head of Academic Affairs, faculty coordinators, professors, and students themselves. Each contributes different requirements:

- Registrar: "We need to track student enrollment, generate transcripts, verify degree requirements, and manage course scheduling."
- Professors: "I need to see my class rosters, record grades, and track prerequisites for my courses."
- Student Affairs Office: "We need to process registrations, track scholarship eligibility, and verify prerequisite completion."
- Students: "I want to register, see my unofficial transcript, check my degree progress."

Conceptual Design: These requirements translate into entities — FACULTY, STUDENT, PROFESSOR, COURSE, CLASS_SESSION, and TEXTBOOK — connected through specific relationships. The ER diagram captures the key scenarios: students enroll in courses (ENROLLS_IN), professors teach courses, courses have prerequisites (PREREQUISITE), class sessions are linked to courses and tracked for attendance (ATTENDS), and professors are assigned textbooks for the courses they teach (ASSIGNMENT). Each entity belongs to a faculty, which organizes the university's academic structure (Elmasri & Navathe, 2016).

Logical Design: The ER model transforms into a relational schema. A representative selection of the core tables is shown below:

FACULTY(Faculty_ID, Name, Budget, Location)

STUDENT(Student_ID, First_name, Last_name, Birth_date, Street, City, Country, Zip_code, Faculty_ID*)

PROFESSOR(Professor_ID, First_name, Last_name, Office, Email, Hire_date, Academic_title, Faculty_ID*, Heads_Faculty_ID*)

COURSE(Course_ID, Title, Credits, Description, Faculty_ID*, Professor_ID*)

ENROLLS_IN(Student_ID*, Course_ID*, Enrollment_date, Grade)

CLASS_SESSION(Course_ID*, Session_number, Year, Room, Semester, Time, Day, Duration)

Foreign keys enforce relationships between tables; constraints enforce business rules such as valid enrollment dates and grade values (Date, 2003).

Physical Design: Implement in SQL Server, choosing appropriate data types — Student_ID as VARCHAR(10), Grade as VARCHAR(10), Credits as INT — creating indexes on frequently-queried columns such as Student_ID and Course_ID, and defining views for common reports like student transcripts and course enrollment lists (Coronel & Morris, 2017).

🏢 *Company System Database*

Requirements Collection: You interview employees, managers, HR personnel, sales representatives, and inventory managers:

- HR: "We need employee information, job history, salary records, and benefits enrollment."
- Sales: "Track customers, orders, products, pricing, shipping information."
- Operations: "Manage inventory levels, supplier relationships, purchase orders."
- Management: "Generate sales reports, analyze customer trends, forecast inventory needs."

Conceptual Design: The requirements reveal seven entities — EMPLOYEE, DEPARTMENT, WAREHOUSE, CUSTOMER, ORDER, PRODUCT, and SUPPLIER — connected through specific relationships. The ER diagram captures the key scenarios: employees work for departments (works_for) and work in warehouses (works_at), one employee manages each department (manages), customers place orders (places), employees process those orders (processes), each order contains multiple order lines (contains), products are stored in warehouses (stored_in), and suppliers provide products (supplies). ORDER_LINE is a weak entity that depends on ORDER for its identification (Elmasri & Navathe, 2016).

Logical Design: The ER model transforms into a relational schema. A representative selection of the core tables is shown below:

DEPARTMENT(Department_ID, Dept_name, Location, Budget)

EMPLOYEE(Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Country, Zip_code, Department_ID*, Warehouse_ID*, Manages_Dept_ID*)

CUSTOMER(Customer_ID, First_name, Last_name, Email, Join_date, Street, City, Country, Zip_code)

ORDER(Order_ID, Status, Order_date, Required_date, Shipped_date, Employee_ID*, Customer_ID*)

ORDER_LINE(Order_ID*, Line_number, Unit_price, Quantity, Discount, Product_ID*)

PRODUCT(Product_ID, Product_name, Description, Category, Price, Weight)

STORED_IN(Product_ID*, Warehouse_ID*, Quantity_on_hand, Reorder_level)

Phone numbers for employees, customers, and suppliers are stored in separate tables (EMPLOYEE_PHONE, CUSTOMER_PHONE, SUPPLIER_PHONE) to correctly handle their multivalued nature. Normalization ensures data integrity throughout the schema (Date, 2003).

Physical Design: Implement in SQL Server with careful attention to transaction performance — orders must process quickly even under concurrent load — reporting efficiency, since analytical queries should not slow down order processing, and security, as salary information and customer data require restricted access through appropriate user roles and permissions (Coronel & Morris, 2017).

Universal Principles, Different Contexts

Both examples follow the same design process, use the same ER notation, transform according to the same logical design rules, and implement with the same SQL DDL. The *techniques* are universal; only the specific *content* differs (Elmasri & Navathe, 2016).

By studying both academic and business contexts, you will recognize patterns that transcend specific domains. You will see that:

- Many-to-many relationships arise everywhere (students-courses, orders-products)
- Hierarchical structures are common (organizational hierarchies, course prerequisites)
- Temporal data matters (enrollment semesters, order dates)
- Business rules vary, but the mechanisms for enforcing them are standard

This dual approach prepares you to design databases for any domain, not just the specific examples covered in this book.

2.10 Summary: Database Design as Systematic Transformation

Database design transforms informal, conversational requirements into precise, formal, implementable database structures through a systematic multi-phase process (Hoffer et al., 2016). Success depends on getting each phase right before proceeding to the next, because errors compound: a mistake in requirements leads to an incorrect conceptual model, which in turn leads to a flawed logical schema, resulting in a database that doesn't meet organizational needs (Teorey et al., 2011).

Key Principles of Database Design

Database design is a systematic, multi-phase transformation from requirements to implementation:

The **database development lifecycle** consists of distinct phases: requirements collection and analysis, conceptual design (creating ER diagrams), logical design (creating relational schemas), physical design (making DBMS-specific decisions), and implementation (building actual databases) (Elmasri & Navathe, 2016).

DBMS independence characterizes early phases: Conceptual design describes *what* data to store without committing to *how* specific DBMS products will implement it. This independence prevents technology constraints from limiting the data model (Date, 2003).

Iteration is normal and necessary: Design rarely succeeds in a single linear pass. Requirements are clarified through design, stakeholder feedback reveals misunderstandings, and validation exposes errors. Productive iteration improves designs while maintaining forward progress (Connolly & Begg, 2015).

Communication bridges domains: Database designers translate between business stakeholders (who think in operational terms) and technical implementation (which requires formal specifications). Effective communication requires understanding both worlds (Hoffer et al., 2016).

Early investment saves exponential later costs: Time spent gathering complete requirements, creating accurate conceptual models, and validating logical designs prevents vastly more expensive fixes after implementation. A day spent on careful design saves weeks of rework (Teorey et al., 2011).

This chapter provided the roadmap. Subsequent chapters provide detailed skills: Chapters 3-5 teach conceptual design using ER and EER modeling; Chapters 6-7 cover logical design and the relational model; and Chapters 8-9 address SQL implementation. Each chapter builds systematically on this foundation, ultimately equipping you with comprehensive database design capabilities (Elmasri & Navathe, 2016).

The journey from the registrar's informal description ("We need to track students and their courses") to a functioning database serving thousands of concurrent users begins with a clear understanding of the design process. The following sections examine each phase in turn.

Review Questions

Conceptual Understanding Questions

2.1. Lifecycle Phases: Describe the major phases of the database development lifecycle (DDLC) and explain the primary deliverables of each phase. How do these phases connect to the broader Software Development Life Cycle (SDLC)?

2.2. DBMS Independence vs. Dependence: Explain the distinction between DBMS-independent and DBMS-specific design phases. Why is maintaining DBMS independence during conceptual design important? At what point does the design necessarily become DBMS-specific?

2.3. Requirements Categories: Distinguish between data requirements and functional requirements, providing specific examples of each from either the University or Company case study. How do these two categories of requirements influence database design differently?

2.4. Conceptual Design Purpose: Explain the primary purposes of conceptual design. Why do we create ER diagrams rather than proceeding directly from requirements to table definitions?

2.5. Transformation Phases: Describe how the database structure evolves as it transforms from conceptual design (ER diagram) through logical design (relational schema) to physical design (actual database). What changes at each transformation step?

Application and Analysis Questions

2.6. Requirements Discovery Scenario: You're designing a database for a hospital patient management system. Identify at least four different stakeholder types you would interview, and explain what specific information each would contribute to requirements analysis.

2.7. Business Rule Identification: A company manager tells you, "Sales representatives are assigned to specific regional territories, and they handle orders from customers in those territories." Identify at least five implicit business rules or requirements hidden in this statement. What clarifying questions would you ask?

2.8. ER to Relational Transformation: Given a conceptual requirement that "students enroll in courses, and we need to track what grade each student received in each course," describe how this would be represented in: (a) an ER diagram, (b) a logical relational schema with tables and foreign keys. What transformation rules apply?

2.9. Cost of Late Discovery: Provide a specific example showing how discovering a missing requirement at different phases has dramatically different costs. Compare the effort required to fix this problem if discovered during: (a) requirements analysis, (b) conceptual design validation, (c) after implementation in production.

2.10. Stakeholder Communication: You need to explain your conceptual ER diagram to a business manager who has no technical database knowledge. Describe your approach—what terminology would you use, what would you avoid, and how would you ensure they understand?

Critical Thinking Questions

2.11. Design Process Assessment: Some developers argue that "we don't need all this design work—we can just create tables as we need them during development." Explain why this approach leads to problems. What specific issues arise from skipping conceptual and logical design?

2.12. Iteration Management: Design iteration can be either productive (improving quality) or counterproductive (preventing progress). Describe the characteristics that distinguish productive iteration from "analysis paralysis." How would you manage iteration to maintain forward progress?

2.13. Chen Notation Justification: This book uses Chen notation for ER diagrams. Research at least two other ER diagram notation systems (such as crow's foot or UML class diagrams). Compare their strengths and weaknesses relative to Chen's notation. When might you choose an alternative notation?

2.14. Requirements Validation Challenges: What makes validating requirements particularly difficult when working with non-technical stakeholders? How can you design

validation sessions that effectively catch requirements errors before they become design errors?

2.15. Technology Independence Trade-offs: Explain the trade-offs involved in maintaining DBMS independence during conceptual and logical design. What benefits does independence provide? What constraints might it impose? When might DBMS-specific considerations appropriately influence early design?

Practical Application Questions

2.16. University Database Requirements: For the University Management System case study, list at least 10 specific data items that requirements analysis would need to identify. Include entities, attributes, relationships, and business rules. Explain how you would discover each through stakeholder interviews.

2.17. Company Database Scope: For the Company/Business Database case study, identify at least 8 different stakeholder roles and explain what specific data or functional requirements each would contribute. How might requirements from different stakeholders' conflict?

2.18. Requirements Document Structure: Design the structure of a requirements specification document for a database project. What sections should it include? What information should each section contain? How would this document be used in subsequent design phases?

2.19. Interview Planning: You're about to interview a university registrar to gather requirements for a student information system. Prepare at least 10 open-ended questions designed to elicit data requirements, functional requirements, and business rules. Explain what each question is designed to discover.

2.20. Validation Session Design: Plan a validation session for presenting a conceptual ER diagram to business stakeholders. What materials would you prepare? What questions would you ask? How would you structure the session to maximize effective feedback?

Integration and Synthesis Questions

2.21. Lifecycle Integration: Explain how the database development lifecycle integrates with application development. At what points do database designers and application developers need to coordinate? What information must be exchanged between these roles?

2.22. Quality Criteria Across Phases: Identify at least three quality criteria that apply across multiple design phases (e.g., completeness, accuracy, clarity). Explain how these criteria manifest differently at the requirements, conceptual design, and logical design phases.

2.23. Case Study Comparison: Compare the University and Company case study examples. What entities, relationships, or requirements patterns appear in both domains? What differences in business rules or data structures reflect the different organizational contexts? What does this reveal about universal database design principles?

2.24. Design Decision Rationale: Throughout the design process, designers make numerous decisions: which entities to create, how to structure relationships, what attributes

to include, etc. Explain how requirements analysis, business rules, and stakeholder input should guide these decisions. Provide specific examples.

2.25. Chapter Connections: This chapter overviewed the complete design process. Based on this overview, what aspects of conceptual design (covered in Chapters 3-5) seem most challenging? What aspects of logical design and implementation (Chapters 6-9) will require the most careful attention? How will understanding the overall process help you master individual phases?

Advanced Thinking Questions

2.26. Agile Database Design: Research agile software development methodologies. How do agile principles (iterative development, responding to change, frequent delivery) apply to database design? What challenges arise when trying to be "agile" with database development? How can you balance thorough design with agile responsiveness?

2.27. Design Patterns: Just as software development has design patterns (standard solutions to common problems), database design has recurring patterns for modeling common business scenarios. Identify at least three database design patterns (e.g., temporal data, hierarchical structures, many-to-many relationships with attributes) and explain when each applies. Use examples from the University or Company case studies.

2.28. Reverse Engineering: Suppose you inherited an existing database with poor documentation. Describe the reverse-engineering process you would use to understand its design: What would you examine? How would you create conceptual ER diagrams from existing tables? What challenges would you face?

2.29. Legacy System Evolution: Many organizations have databases that have evolved over decades, with layers of modifications built on top of original designs. Explain the specific challenges of modifying such legacy databases. How would you approach adding major new functionality to a decades-old database? What risks must be managed?

2.30. Professional Practice: Based on this chapter's discussion of the complete design process, what professional skills beyond technical database knowledge does a successful database designer need? Consider communication, analysis, project management, and domain knowledge. How would you develop these complementary skills?

References and Recommended Readings

Primary References

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of Database Systems* (7th ed.). Pearson.

Connolly, T. M., & Begg, C. E. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson.

Date, C. J. (2003). *An Introduction to Database Systems* (8th ed.). Addison-Wesley.

Design Process and Methodology

Teorey, T. J., Lightstone, S. S., Nadeau, T., & Jagadish, H. V. (2011). *Database Modeling and Design: Logical Design* (5th ed.). Morgan Kaufmann.

Hoffer, J. A., Ramesh, V., & Topi, H. (2016). *Modern Database Management* (12th ed.). Pearson.

Coronel, C., & Morris, S. (2017). *Database Systems: Design, Implementation, & Management* (12th ed.). Cengage Learning.

Requirements Engineering and Analysis

Wiegers, K., & Beatty, J. (2013). *Software Requirements* (3rd ed.). Microsoft Press.

Sommerville, I. (2015). *Software Engineering* (10th ed.). Pearson.

Robertson, S., & Robertson, J. (2012). *Mastering the Requirements Process: Getting Requirements Right* (3rd ed.). Addison-Wesley.

Database Implementation

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database System Concepts* (7th ed.). McGraw-Hill.

Ramakrishnan, R., & Gehrke, J. (2003). *Database Management Systems* (3rd ed.). McGraw-Hill.

Project Management and Stakeholder Engagement

Project Management Institute. (2017). *A Guide to the Project Management Body of Knowledge (PMBOK Guide)* (6th ed.). PMI.

Schwalbe, K. (2015). *Information Technology Project Management* (8th ed.). Cengage Learning.

Complementary References

Hristoski, I., & Dimovski, T. (2022). Dizajn na relaciski bazi na podatoci, Tom 1 [Design of Relational Databases, Volume 1]. University "St. Kliment Ohridski", Bitola. (In Macedonian)

Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database Systems: The Complete Book* (2nd ed.). Pearson.

Part 2

Conceptual Design

Entity-Relationship (ER)

Enhanced Entity-Relationship (EER) Modeling

Learning Objectives:

By the end of Part 2, you will be able to:

- Understand the role and importance of conceptual design in the database development lifecycle
- Apply systematic methodologies for analyzing requirements and creating conceptual models
- Master Entity-Relationship (ER) modeling techniques using Chen notation
- Handle complex modeling scenarios using Enhanced Entity-Relationship (EER) concepts
- Validate and refine conceptual models to ensure they accurately represent business requirements
- Create comprehensive conceptual designs that serve as blueprints for successful database implementations

***Chapter Overview*:**

Chapter 3: Conceptual Design Fundamentals Introduces the principles, processes, and methodologies of conceptual database design. Covers requirement analysis, stakeholder communication, design validation, and the systematic approach to creating technology-independent data models that accurately represent organizational needs.

Chapter 4: Entity-Relationship (ER) Modeling Provides comprehensive coverage of ER modeling using Chen notation, including entities (strong and weak), attributes (simple, composite, multivalued, derived), relationships (binary, recursive, ternary), cardinality constraints, and participation constraints. Features complete case studies demonstrating step-by-step ER diagram construction.

Chapter 5: Enhanced Entity-Relationship (EER) Modeling Extends basic ER concepts with advanced modeling constructs including specialization/generalization hierarchies, inheritance relationships, category (union) types, and complex constraint specifications. Demonstrates how EER modeling handles sophisticated real-world scenarios that exceed basic ER capabilities.

Our Comprehensive Case Study Approach: Throughout Part 2, we employ **Chen notation**, the original and most precise ER diagramming method, as our primary modeling language. All concepts are illustrated through two comprehensive, real-world case studies that demonstrate practical application: the **University Management System** (modeling students, courses, professors, faculties, and academic relationships) and the **Company/Business Database** (handling employees, departments, customers, orders, products, and organizational structures). These examples show how abstract modeling principles address real business challenges, ensuring that each concept learned can be directly applied to real database design projects.

Chapter 3

3. Conceptual Design

3.1 Introduction: The Art of Understanding Before Building

Imagine an architect sitting down with a family who dreams of building their perfect home. The family describes their vision: "We want an open kitchen that flows into the living room, a quiet home office, bedrooms that get morning light, and outdoor spaces where we can entertain friends." The architect does not immediately start drawing blueprints or calculating load-bearing requirements. Instead, she asks thoughtful questions, listens carefully, sketches rough ideas, and gradually develops a clear understanding of what the family truly needs versus what they initially thought they wanted.

This careful discovery process exemplifies the process of **conceptual database design**. Before we can create efficient data storage systems or implement security measures, we must first understand the essence of what we are building. **Conceptual design** serves as a critical bridge between the messy, complex reality of organizational needs and the structured, precise world of database implementation (Batini et al., 1992).

> **Why This Chapter Matters**
>
> The success or failure of an entire database projects often hinges on conceptual design. When done well, it provides a solid foundation for all subsequent development. When rushed or poorly executed, problems cascade through every phase, becoming exponentially more expensive to fix as development progresses.

This chapter explores the fundamental principles and processes that transform business requirements into clear, technology-independent models. You will discover why conceptual design often determines project success, and you will learn a systematic approach for creating models that accurately capture organizational reality while serving as reliable blueprints for the **ER and EER diagrams** you'll create in Chapters 4 and 5.

What You'll Learn

By the end of this chapter, you'll understand:

- **What conceptual design is** and why it's the most critical phase in database development
- **The three-phase process** for transforming business requirements into structured models

- **How to work effectively** with business stakeholders who lack technical database knowledge
- **Quality standards** that distinguish good conceptual models from poor ones
- **The direct connection** between conceptual design principles and the ER/EER modeling techniques you'll apply in the next chapters

Most importantly, you'll see exactly **how the abstract principles** in this chapter translate into the concrete ER diagrams you'll start creating in Chapter 4.

3.2 Understanding Conceptual Database Design

Conceptual database design is the process of creating an abstract, high-level representation of information requirements that captures the essential structure, relationships, and business rules of an organization's data needs without concern for how the information will be physically stored or accessed (Chen, 1976; Elmasri & Navathe, 2016).

> **Definition: Conceptual Database Design**
>
> **Conceptual database design** creates a technology-independent model that shows **what** data needs to be stored and **how** different data elements relate to each other, without specifying **how** the data will be implemented in any particular database system.

Think of conceptual design as creating a **map of the information landscape**. Just as a geographical map shows cities, rivers, and mountains without depicting every tree or building, a conceptual data model reveals the major information structures and their connections without specifying implementation details such as *file structures* or *indexing strategies*.

3.2.1 Why Abstraction Matters

Conceptual design operates at the highest level of abstraction in database development, serving three crucial purposes:

Technology Independence: Conceptual models don't commit to specific database management systems, preserving flexibility for later optimization. A well-crafted conceptual model remains valuable even as underlying technologies evolve, because it captures **enduring business logic** rather than transient technical details.

Business Focus: Abstraction keeps attention centered on organizational requirements rather than technical implementation challenges. This enables *meaningful participation* by business users with domain expertise but without technical database knowledge.

Stability: Business concepts and relationships typically remain stable even when technology platforms change. The relationship between *customers and orders* or *students and courses* persists regardless of whether data is stored in relational databases, document stores, or cloud systems.

3.2.2 From Business Requirements to ER Elements

Here's how conceptual design connects to the ER diagrams you'll create in Chapter 4:

From Requirements to ER: A Preview

Business Statement: "Students enroll in courses and receive grades."
Conceptual Analysis:

- Important objects: Students, Courses
- Relationship: Enrollment
- Additional information: Grades belong to the enrollment relationship

ER Diagram Elements (Chapter 4):

- STUDENT entity, COURSE entity
- ENROLLS relationship
- Grade attribute on the relationship

This preview shows exactly how the abstract thinking you will learn in this chapter translates into the concrete ER diagrams you'll draw in Chapter 4. Every business requirement undergoes this same transformation process from conceptual understanding to a visual model.

3.2.3 Characteristics of Effective Conceptual Models

Quality conceptual models share four essential characteristics:

- **Comprehensive Scope**: Captures all major information structures and business rules essential to operations
- **Business Rule Integration**: Encodes the policies and constraints that govern how organizations operate
- **Stakeholder Validation Capability**: Allows business users to verify requirements using familiar language
- **Implementation Independence**: Preserves options for technical optimization in later design phases

These characteristics ensure that conceptual models serve as reliable bridges between business needs and technical implementation—exactly what you need to create effective ER and EER diagrams in the chapters ahead.

3.3 Why Conceptual Design Determines Project Success

Conceptual design is often the single most important phase in database development because decisions made here echo throughout the entire system lifecycle, influencing everything from *query performance* to *maintenance costs* to *user satisfaction* (Mancas, 2015; Teorey et al., 2011).

3.3.1 The Foundation Effect

Conceptual design lays the foundation for all subsequent development. Every table structure, every relationship, and every application feature ultimately traces back to decisions made during conceptual design. When this foundation is **solid and well-**

conceived, development proceeds smoothly. When it's *flawed or incomplete*, problems compound through every phase.

Consider this real example: A university registration system where the conceptual design failed to properly model the relationship between course sections and time conflicts. During conceptual design, they overlooked that *students shouldn't be able to register for two courses that meet at the same time* (like Intro to Programming on Tuesdays 2-4 PM and Database Systems also on Tuesdays 2-4 PM). This seemingly *minor oversight* eventually required:

- **Complex application logic** to prevent scheduling conflicts (programming code to check every possible time overlap)
- **Elaborate database constraints** to maintain data integrity (complicated rules to prevent invalid enrollments)
- **Frustrated users** encountering arbitrary system limitations (students getting error messages without understanding why)
- **Expensive system modifications** after deployment (restructuring the entire course scheduling system)

The **cumulative effect** becomes more apparent as systems mature. *Well-designed conceptual models* gracefully accommodate new requirements and organizational growth. *Poorly designed foundations* require extensive modifications, often necessitating the **complete redevelopment** of major components.

3.3.2 The Cost of Change Curve

One of the most compelling reasons to invest in thorough conceptual design is understanding how the **cost of fixing design errors** increases dramatically as development progresses. A *missing information structure* discovered during conceptual design might require only updating a few diagrams and having additional conversations with stakeholders. The same discovery during application development requires *database schema changes, code modifications*, and *user interface adjustments*. During production, that same oversight could necessitate **data migration**, **system downtime**, and **extensive regression testing**.

This creates an **exponential cost progression** where fixing the same problem becomes 10 times more expensive in the development phase and 100 times more expensive in production. For example, if correcting a missing requirement costs 1,000 EUR during conceptual design, the same correction would cost 10,000 EUR during development and 100,000 EUR in production.

> **The Cost Curve Reality**
>
> - **Conceptual Design Phase**: Missing requirement = Update diagrams, have stakeholder conversations
> - **Development Phase**: Same missing requirement = Database changes, code modifications, UI adjustments
> - **Production Phase**: Same missing requirement = Data migration, system downtime, extensive testing

Cost multiplier: 1x → 10x → 100x
(A 1,000 EUR fix becomes 10,000 EUR, then 100,000 EUR)

Understanding this **exponential cost progression** explains why experienced database professionals invest heavily in getting conceptual design right the first time. The investment in thorough requirements analysis and stakeholder validation pays tremendous dividends by preventing much more expensive corrections later.

Conceptual design is also the last point at which business stakeholders can meaningfully influence the fundamental system architecture. Once development moves to technical phases, models become too complex for business users to evaluate effectively.

3.3.3 A Failure Story: When Conceptual Design is Skipped

> **Real-World Consequence: The Rushed E-commerce System**
>
> **Situation**: A startup skipped conceptual design to launch quickly, building directly from basic requirements: "Customers buy products."
> **What They Missed**:
> - Guest checkout vs. registered customers
> - Product variants (sizes, colors)
> - Inventory tracking across multiple warehouses
> - Return and exchange processes
>
> **Results After 6 Months**:
> - Database redesign required: 150,000 EUR
> - Lost sales during system downtime: 50,000 EUR
> - Customer data migration problems: 2 weeks offline
> - **Total cost**: 5x more than a proper conceptual design would have cost
>
> **Lesson**: Three weeks of conceptual design would have prevented six months of expensive fixes.

This failure story illustrates why **conceptual design** isn't an academic exercise. It's a practical necessity that directly impacts business success. When you create ER diagrams in Chapter 4, you are not just drawing pictures; you are making decisions that will affect the entire system's future.

Another Failure Story: The Medical Clinic System

Situation: A medical clinic built a patient management system without proper conceptual design, working from the simple requirement: "Track patient appointments with doctors."
What They Missed During Conceptual Design:
- Different appointment types (consultation, follow-up, procedure) have different time requirements
- Doctors have different specializations and availability patterns
- Patients need medical history tracking across multiple visits
- Insurance verification and billing rule variations
- Emergency appointment handling procedures

Results After 4 Months:

- Double-booking conflicts: Lost patient trust
- Insurance claim rejections: 25,000 EUR in unbilled services
- Manual workarounds: 2 additional staff hired
- System replacement required: 75,000 EUR

Impact: Poor conceptual design created a system that hindered rather than helped patient care, ultimately costing more than starting over.

3.4 The Three-Phase Conceptual Design Process

Successful conceptual design follows a systematic three-phase process that transforms business requirements into structured information models. This proven methodology ensures comprehensive coverage of requirements while maintaining focus on business value and stakeholder needs (Batini et al., 1992; Teorey et al., 2011).

> **The Three-Phase Framework**
>
> **Phase 1: Requirements Discovery** - Understanding what the system needs to accomplish
> **Phase 2: Data Analysis and Modeling** - Transforming requirements into structured models
> **Phase 3: Validation and Refinement** - Ensuring models accurately represent business needs

This systematic approach prevents the chaos that results from jumping directly into technical design without a proper understanding of business requirements.

3.4.1 Phase 1: Requirements Discovery

Requirements discovery develops a deep understanding of what the system needs to accomplish. This phase employs multiple techniques to reveal both explicit requirements stated directly by stakeholders and *implicit requirements* that emerge from understanding business processes.

Key Activities:

- **Stakeholder identification**: Map everyone whose needs affect system success (*direct users, managers, auditors, customers*)
- **Information gathering**: Use *structured interviews, document analysis*, and *process observation*
- **Business rule discovery**: Uncover both formal policies and informal practices

The **gap between documented procedures and actual practices** often provides the most valuable insights. When stakeholders say "we don't really use that form," they're revealing **hidden requirements** that formal documentation misses.

3.4.2 Phase 2: Data Analysis and Modeling

Data analysis transforms requirements into structured information models that can be evaluated and refined. This transformation requires both *analytical skills* to identify patterns and *synthesis skills* to organize complex requirements coherently.

Key Activities:

- **Concept identification**: Analyze requirements for entities (organizational objects like students, courses, orders)
- **Relationship discovery**: Understand how concepts connect (students enroll in courses, customers place orders)
- **Business rule specification**: Translate organizational policies into formal constraints

> **From Analysis to ER Elements**
>
> **Requirement**: "Students enroll in courses. Each student can enroll in multiple courses, and each course can have multiple students. We track the enrollment date and final grade."
>
> **Analysis Results**:
>
> - **Entities**: STUDENT, COURSE
> - **Relationship**: ENROLLS (many-to-many)
> - **Attributes**: Enrollment_Date, Grade (belong to the relationship)
>
> **Next**: Chapter 4 shows exactly how to draw this as an ER diagram

The **modeling process itself** serves as a discovery tool, revealing gaps in requirements and inconsistencies between stakeholder perspectives that weren't apparent during initial analysis.

3.4.3 Phase 3: Validation and Refinement

Validation and refinement ensure models accurately represent business requirements and will support successful implementation. This phase involves multiple stakeholders in a systematic evaluation using realistic business scenarios.

Key Activities:

- **Stakeholder validation**: Present models using business language and realistic scenarios
- **Scenario testing**: Walk through typical business situations to verify model adequacy
- **Quality assurance**: Check for *clarity, completeness, correctness*, and *consistency*

Validation sessions should focus on business concepts rather than technical notation. The goal is to verify that models capture business reality, not teach stakeholders ER diagramming techniques.

3.4.4 How This Connects to ER/EER Modeling

This three-phase process directly feeds into the **ER and EER modeling techniques** you'll learn in Chapters 4 and 5:

- **Phase 1 discoveries** become the business requirements you'll analyze in Chapter 4
- **Phase 2 analysis** produces the entities, relationships, and constraints you'll draw in ER diagrams
- **Phase 3 validation** ensures your ER/EER models accurately represent stakeholder needs

Every **Chen notation diagram** you create in Chapter 4 will be the direct result of systematically applying this three-phase process to real business requirements.

3.5 Working with Business Stakeholders

One of the most critical aspects of conceptual design involves **effective communication** with stakeholders who possess domain expertise but lack technical database knowledge. The success of database projects often depends more on how well you communicate with business users than on the **technical elegance** of your ER diagrams.

3.5.1 The Communication Challenge

Database designers and **business stakeholders** operate in different worlds. *Business users* think in terms of processes, workflows, and practical problems they need to solve. *Technical designers* think in terms of data structures, relationships, and constraints. Bridging this gap requires specific techniques that respect both perspectives.

> **Common Communication Pitfalls**
>
> - **Technical Jargon**: "We need to normalize the customer entity and establish foreign key relationships."
> - **Implementation Focus**: Discussing database tables instead of business objects
> - **Assumption Making**: Assuming business requirements are obvious
> - **Poor Validation**: Failing to verify that stakeholders understand your models

Solution Strategies:

- Use **business language**: Talk about "customers and orders," not "tables and foreign keys."
- Focus on **what** the system accomplishes, not **how** it works technically
- Ask for **concrete examples** rather than abstract descriptions
- **Validate understanding** by having stakeholders explain back what they heard

3.5.2 Practical Communication Techniques

Visual Communication: Different stakeholders need different views of your models. *Executives* need high-level overviews, *process managers* need workflow-focused views, and *end users* need to see how the system supports their daily tasks.

Iterative Presentation: Build understanding gradually. Start with the **most important concepts** (like major entities) before adding complexity (like relationship constraints). This prevents stakeholders from being overwhelmed by complete, complex models all at once.

Interactive Modeling: Involve stakeholders in the model creation process. Ask questions like: "If a customer places an order, what information do we need to track?" This participatory approach reveals requirements that traditional presentations miss.

3.5.3 Managing Conflicting Requirements

Different stakeholders often have **conflicting requirements** that must be reconciled. The Sales department might want customer information organized differently from the Marketing department. These conflicts are natural; different roles create different perspectives.

Resolution Strategies:

- **Facilitated workshops**: Bring conflicting stakeholders together to discuss differences
- **Requirements prioritization**: Rank requirements by business value and feasibility
- **Alternative analysis**: Show stakeholders the trade-offs of different design approaches
- **Phased implementation**: Address critical needs first, plan future enhancements

> **Communication Success Example**
>
> **Wrong Approach**: "We need a many-to-many relationship between the customer and product entities with an associative entity for order line items."
> **Right Approach**: "Let me understand this: A customer can buy many different products in a single order, and we need to track how many of each product they bought and at what price. Is that correct?"
> The second approach uses business language and asks for confirmation, making it much more likely to uncover the real requirements.

Remember: When you create ER diagrams in Chapter 4, you are not just drawing for other database designers—you are creating models that business stakeholders must validate and approve. The communication skills you develop here are essential for creating ER diagrams that truly serve business needs.

3.6 Business Rules: The Hidden Foundation of Database Design

Business rules govern how organizations function and include policies, regulations, and operational constraints. Their identification and proper modeling are crucial for creating

systems that accurately reflect organizational reality (Robertson & Robertson, 2012; Teorey et al., 2011).

> **Why Business Rules Matter**
>
> Business rules eventually become the *constraints, validations*, and *business logic* that ensure data integrity in your implemented database. Miss them during conceptual design, and you'll spend months adding complex workarounds later.

3.6.1 Four Categories of Business Rules

Understanding different types of business rules helps you identify them systematically during requirements analysis:

Structural Rules: Define what information concepts exist and how they relate

- *"Every employee must be assigned to exactly one department."*
- *"Courses can have multiple prerequisites."*
- *"Orders contain one or more line items."*

Operational Rules: Govern what actions are allowed, required, or forbidden

- *"Orders cannot be shipped until payment is confirmed."*
- *"Students cannot enroll in courses without meeting prerequisites."*
- *"Inventory levels must be checked before confirming orders."*

Derivation Rules: Specify how calculated information is determined

- *"Student average grade is calculated as the weighted average of all completed courses."*
- *"Order total equals line items plus taxes and shipping."*
- *"Employee tenure is calculated from hire date to current date."*

Temporal Rules: Specify timing constraints and relationships

- *"Course enrollment begins 30 days before semester starts."*
- *"Employee performance reviews must be completed annually."*
- *"Financial reports must be generated within 5 business days of month-end."*

3.6.2 Discovering Hidden Rules

Explicit rules are formally documented in policy manuals and procedures. These are easy to find but represent only a portion of the actual business rules.

Implicit rules require detective work to uncover. Look for:

- **Observation gaps**: What people actually do vs. what procedures say
- **Exception handling**: Rules that emerge when things go wrong
- **Tribal knowledge**: Informal practices experienced employees use

- **Historical decisions**: Why certain procedures exist

> **Hidden Rule Discovery Example**
>
> **Documented Rule**: "Students register for courses online."
> **Hidden Rules Discovered Through Investigation**:
> - Students can't register for more than 30 credits per semester without approval from the Head of Academic Affairs
> - Prerequisites must be completed with a grade of 6 or better (passing grade)
> - Each course is worth 6 credits, with 5 courses per semester being the standard load
> - International students need visa status verification
> - Some courses require instructor permission regardless of prerequisites

3.6.3 From Business Rules to ER Constraints

Business rules translate directly into the **constraints and cardinalities** you'll specify in ER diagrams:

Information-level rules → *Attribute constraints* (domain values, format requirements)
Relationship-level rules → *Cardinality and participation constraints* (1:1, 1:N, M:N, mandatory/optional)
System-level rules → *Complex constraints* spanning multiple entities

> **Business Rule → ER Constraint Example**
>
> **Business Rule**: "Each student can enroll in multiple courses, but each enrollment must have exactly one grade."
> **ER Implementation** (Chapter 4):
> - STUDENT entity connected to COURSE entity
> - ENROLLS relationship (M:N cardinality)
> - Grade attribute attached to the ENROLLS relationship
> - Participation: Students must enroll in at least one course (total participation)

This preview shows exactly how the business rules you identify during conceptual design become the constraints you'll draw in your ER diagrams in Chapter 4.

3.6.4 Why This Matters for ER Modeling

When you create ER diagrams in Chapter 4, every **cardinality constraint** (1:1, 1:N, M:N) and every **participation constraint** (mandatory or optional) you draw will be based on business rules you identify during conceptual design. Missing business rules during conceptual analysis means missing constraints in your ER diagrams, which leads to databases that can't enforce organizational policies.

3.7 Quality Assurance: Getting Your Models Right

Creating a conceptual model represents only half the challenge; ensuring that your model accurately reflects business requirements and supports effective implementation requires **systematic quality assurance** (Batini et al., 1992).

3.7.1 The Four Quality Standards

Quality conceptual models must meet four essential criteria that distinguish professional work from amateur efforts:

Clarity: Can stakeholders understand your models without extensive technical explanation? *Clear models use meaningful business names, appropriate detail levels*, and *minimal technical jargon*.

Completeness: Do your models address all identified requirements? *Complete models represent all important business concepts, capture all significant relationships*, and *incorporate all business rules*.

Correctness: Do your models accurately reflect business reality? *Correct models use accurate cardinality constraints, realistic participation constraints*, and *valid business rule representations*.

Consistency: Are your models internally coherent? *Consistent models apply uniform naming conventions, use consistent approaches for similar situations*, and *avoid contradictory constraints*.

3.7.2 Validation Techniques

Stakeholder Walkthrough Sessions provide structured opportunities for business users to evaluate your models. These sessions require **careful preparation** to ensure technical concepts are explained in *business language* and realistic scenarios test model adequacy.

> **Effective Walkthrough Structure**
>
> 1. **Start Simple**: Present major entities and relationships first
> 2. **Use Business Language**: "Customers place orders," not "Customer entity relates to Order entity."
> 3. **Test Scenarios**: Walk through realistic business situations
> 4. **Ask Questions**: "Can this model handle when a customer changes their address?"
> 5. **Document Gaps**: Record what's missing or incorrect

Scenario Testing involves walking through realistic business situations to verify your models can handle *required operations*, accommodate *exception cases*, and support *reporting requirements*. Each scenario should trace through your conceptual model to verify that all necessary information is captured.

Traceability Analysis ensures every business requirement connects to specific model elements, while every model element serves a clear business purpose. *Requirements without model elements* indicate gaps; *model elements without business justification* suggest unnecessary complexity.

3.7.3 Common Quality Problems and Solutions

Over-Modeling: Including details that belong in later design phases

- *Symptoms*: Stakeholders find models confusing, excessive technical detail
- *Solution*: Focus on essential business concepts, defer optimization decisions

Under-Modeling: Missing essential business requirements

- *Symptoms*: Models seem too simple, stakeholders say "but what about..."
- *Solution*: Systematic requirements review, diverse stakeholder input

Inconsistent Modeling: Using different approaches for similar situations

- *Symptoms*: Mixed naming patterns, contradictory constraints
- *Solution*: Establish modeling standards, systematic consistency review

> **Quality Check Questions**
>
> **Before presenting to stakeholders, ask yourself:**
> - Can a business user understand this without my explanation?
> - Have I captured all the business rules stakeholders mentioned?
> - Do my cardinality constraints match how the business actually works?
> - Am I using consistent names and approaches throughout?
> - Can this model support the reports and queries the business needs?

3.7.4 Preparing for ER Diagram Validation

When you create ER diagrams in Chapter 4, you'll need to validate them with stakeholders using these same quality standards. The **Chen notation diagrams** you'll draw must pass the same tests:

- **Clarity**: Can stakeholders understand your entity and relationship names?
- **Completeness**: Do your ER diagrams capture all the business concepts and rules?
- **Correctness**: Do your cardinality constraints (1:1, 1:N, M:N) match business reality?
- **Consistency**: Are you using Chen notation symbols correctly and consistently?

Quality assurance isn't just about catching errors. It's about ensuring your ER diagrams will actually serve the business needs they're designed to support.

3.8 Tools for Conceptual Design

Effective conceptual design requires appropriate tools that support the modeling process while facilitating **stakeholder communication**. The right tool selection significantly influences both design efficiency and the quality of stakeholder engagement.

3.8.1 Choosing Your Modeling Notation

Chen Notation represents the **most precise approach** to ER diagramming and is the primary notation you'll learn in Chapter 4. Chen notation uses explicit symbols—rectangles for entities, diamonds for relationships, and ovals for attributes—that create a clear visual distinction among modeling constructs.

Tool Integration with Chen Notation: Throughout this textbook, we will demonstrate Chen notation using practical tools you can access. **We primarily use Microsoft Visio** to create professional ER diagrams because it offers excellent Chen notation templates and precise symbol alignment. **We also use Draw.io** (now called diagrams.net) for simpler examples and when demonstrating concepts that benefit from freely accessible tools. Both tools effectively support Chen notation's explicit symbols—rectangles for entities, diamonds for relationships, and ovals for attributes—while maintaining the *semantic precision* that makes Chen notation valuable for learning.

> **Why Chen Notation for Learning**
>
> - **Semantic Precision**: Each symbol has an exact meaning, reducing ambiguity
> - **Business Rule Clarity**: Excellent for capturing cardinality and participation constraints
> - **Academic Standard**: Widely taught and recognized in database education
> - **Direct Translation**: Clear path from ER diagrams to relational database design

Alternative notations, such as Crow's Foot or UML Class Diagrams, offer distinct advantages but sacrifice some semantic precision. You will focus on Chen notation throughout Chapters 4 and 5 because it provides the clearest foundation for understanding ER concepts.

Enhanced ER (EER) Extensions handle *inheritance relationships*, *specialization hierarchies*, and *category types* that basic ER cannot represent. You'll learn when and how to use EER extensions in Chapter 5 for complex organizational hierarchies.

3.8.2 Software Tool Categories

Selecting the right software tool for conceptual design can significantly impact both your efficiency and the quality of stakeholder communication. Understanding the different categories of available tools helps you make informed decisions based on your specific needs and project requirements.

Specialized Database Design Tools represent professional-grade software specifically built for database modeling and development. Tools such as ERwin, PowerDesigner, and MySQL Workbench offer advanced features that support the entire database development lifecycle. These tools offer *built-in validation* to automatically catch modeling errors, constraint checking to ensure your business rules are properly implemented, and automatic code generation to translate your conceptual models into database schemas.

The primary advantage of specialized tools lies in their comprehensive feature sets designed specifically for database professionals. They understand database concepts natively, enabling them to provide intelligent assistance during the modeling process. However, these tools typically require significant investment in both cost and learning time. For students and beginners, the complexity can sometimes overshadow the learning of fundamental concepts.

General Diagramming Tools offer a more accessible approach to conceptual modeling, focusing on visual representation rather than database-specific features. Tools like Microsoft Visio, Lucidchart, and Draw.io provide flexible diagramming capabilities that can effectively

support ER modeling needs. **In this textbook, we primarily use Microsoft Visio to create ER diagrams,** as it offers an excellent balance of functionality and ease of use. We also use Draw.io for some examples, particularly when demonstrating concepts that benefit from simpler, more accessible tools that users can use freely.

The strength of general diagramming tools lies in their *lower cost and learning curves*, making them ideal for educational environments. They typically offer *greater flexibility in diagram layout and formatting*, which can be valuable when presenting models to different stakeholder groups. These tools also provide *better integration with standard business documentation*, allowing you to embed your conceptual models directly into project documents and presentations. Perhaps most importantly, they facilitate *easier sharing with non-technical stakeholders* who may be intimidated by specialized database software.

Collaborative Platforms have emerged as increasingly important tools for conceptual design, especially in today's distributed work environments. Platforms such as Miro, Figma, and various online whiteboarding tools support the inherently collaborative nature of conceptual design by enabling multiple stakeholders to participate simultaneously in model development and refinement.

These platforms excel at supporting real-time collaborative editing, where multiple team members can work on the same model simultaneously and see each other's changes instantly. They provide *version control and change tracking* capabilities that help manage the iterative nature of conceptual design. Most collaborative platforms also offer *integration with project management systems*, connecting your conceptual models to broader project workflows. Their *cross-platform accessibility* ensures that stakeholders can participate regardless of their device or location.

> **Tool Selection**
>
> **For Learning ER/EER Concepts** (Chapters 4-5): Start with simple diagramming tools like Draw.io or Microsoft Visio to focus on concepts rather than software complexity
> **For Professional Projects**: Graduate to specialized database tools once you master the fundamental concepts
> **For Stakeholder Communication**: Use whatever tool makes sharing and collaboration easiest

3.8.3 Getting Ready for Chapter 4

As you prepare to create your first ER diagrams in Chapter 4, remember that **the tool is only a means to an end**. Whether you use sophisticated database software or simple drawing tools, success depends on:

1. **Understanding the business requirements** (conceptual design principles from this chapter)
2. **Applying Chen notation correctly** (technical skills from Chapter 4)
3. **Validating with stakeholders** (communication skills from this chapter)
4. **Maintaining quality standards** (clarity, completeness, correctness, consistency)

The conceptual design process you have learned in this chapter provides the foundation. Chapter 4 will teach you the technical skills needed to transform that foundation into precise ER diagrams using Chen notation.

3.9 Summary: Your Foundation for ER and EER Modeling

Conceptual design represents the **cornerstone of successful database development**, transforming organizational requirements into structured models that guide all subsequent implementation activities. This chapter has provided you with the fundamental principles, systematic processes, and practical techniques that make conceptual design effective.

3.9.1 What You've Learned

You now understand the **three-phase conceptual design process** that professional database designers use:

Phase 1: Requirements Discovery taught you how to systematically gather information from stakeholders, identify both explicit and implicit requirements, and uncover the business rules that will become database constraints.

Phase 2: Data Analysis and Modeling showed you how to transform business requirements into structured information models, identifying entities, relationships, and constraints that accurately represent organizational reality.

Phase 3: Validation and Refinement equipped you with techniques to ensure your models meet quality standards (clarity, completeness, correctness, and consistency) and truly serve business needs.

Beyond the process, you have learned **essential skills** for working with business stakeholders, communicating technical concepts effectively, and managing conflicting requirements that emerge in real projects.

3.9.2 The Bridge to ER and EER Modeling

Everything you have learned in this chapter directly prepares you for the **practical modeling work** in Chapters 4 and 5. The conceptual design principles you have mastered provide the methodology; ER and EER modeling provide the technical implementation approach.

> **From Conceptual Design to ER Diagrams**
>
> **Chapter 3 Foundation → Chapter 4 Application**
> - Requirements discovery → Sentence-by-sentence analysis of business scenarios
> - Business rule identification → Cardinality and participation constraints
> - Stakeholder validation → ER diagram review sessions
> - Quality standards → Professional Chen notation diagrams

Chapter 4: Entity-Relationship Modeling will show you exactly how to apply these conceptual design principles using Chen notation. You'll learn to create precise ER diagrams that capture the business requirements and rules you've learned to identify in this chapter.

Chapter 5: Enhanced ER Modeling will extend these capabilities to handle complex inheritance relationships and organizational hierarchies that basic ER modeling cannot address effectively.

3.9.3 Your Next Steps

As you move into Chapter 4, remember that **ER diagramming is not just a technical drawing exercise**. It's the visual representation of the careful analysis and stakeholder collaboration you have learned in this chapter. Every entity rectangle, relationship diamond, and constraint specification you draw will be grounded in the systematic approach you've developed here.

The **sentence-by-sentence analysis method** you will use in Chapter 4 directly applies the requirements analysis techniques from this chapter. The **business scenarios** you'll model reflect the stakeholder communication skills you've developed. The **validation sessions** you'll conduct use the quality assurance approaches you've learned.

> **Success Formula for ER Modeling**
>
> **Solid Conceptual Foundation** (Chapter 3) + **Technical ER Skills** (Chapter 4) + **Advanced EER Techniques** (Chapter 5) = **Professional Database Design Capability**

The investment you have made in understanding conceptual design principles will pay dividends throughout your database design career. **Quality conceptual design** creates systems that feel natural to users, adapt gracefully to changing requirements, and provide a reliable foundation for organizational growth.

You are now ready to see these abstract principles come alive in the concrete, visual world of ER diagrams. Chapter 4 awaits, where business requirements are translated into precise, implementable database designs using the power of Chen notation and systematic ER modeling techniques.

■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■

Review Questions

Conceptual Understanding Questions

3.1. Definition and Importance: Define conceptual database design and explain why it's considered the most critical phase in database development. How does it differ from the logical and physical design phases you'll learn about later?

3.2. The Three-Phase Process: Explain the three phases of conceptual design (Requirements Discovery, Data Analysis and Modeling, Validation and Refinement). What are the key activities in each phase, and how do they connect to ER modeling?

3.3. Business Rules Categories: Describe the four categories of business rules with specific examples. How do structural rules differ from operational rules, and why does this distinction matter for ER modeling?

3.4. Quality Standards: Explain the four quality standards for conceptual models (clarity, completeness, correctness, consistency). Provide a specific example of what each standard means in practice.

Application and Analysis Questions

3.5. Requirements Discovery: You're designing a library management system. Identify four different stakeholder types you would interview and explain what specific information each stakeholder group would contribute to your requirements analysis.

3.6. Business Rule Identification: A restaurant manager tells you, "Customers make reservations for tables." Identify at least five business rules that might be hidden in this simple statement and explain how you would discover them.

3.7. Cost of Change Analysis: Provide a specific example of how a missing business requirement discovered at different phases would have different costs. Use concrete examples (like the university scheduling conflict example in the chapter).

3.8. Stakeholder Communication: You need to explain your conceptual model to a business manager who has no technical database knowledge. Describe your approach, what language you would use, and what you would avoid.

Critical Thinking Questions

3.9. Quality Problem Diagnosis: You receive a conceptual model that contains 25 different entities, uses inconsistent naming (some plural, some singular), and includes detailed file storage specifications. Identify the quality problems and explain how you would fix them.

3.10. Validation Planning: Design a validation session for a hospital patient management system. What scenarios would you test? What stakeholders would you involve? How would you structure the session to be most effective?

3.11. Tool Selection: You are working on a student project vs. a professional database system for a large corporation. Compare and contrast your tool selection decisions for these two scenarios, explaining your reasoning.

3.12. Failure Analysis: A software company skipped conceptual design and built a customer management system directly from basic requirements. Six months later, they need an expensive database restructuring. What problems might they have encountered, and how could proper conceptual design have prevented them?

Practical Application Questions

3.13. Three-Phase Application: Apply the three-phase conceptual design process to this scenario: "Design a database for a small medical clinic that schedules patient appointments with doctors." Walk through each phase with specific activities you would perform.

3.14. Business Rule Translation: Given this business rule: "Students can enroll in multiple courses, but cannot enroll in more than 30 credit hours per semester without approval," explain how this would translate into ER diagram constraints in Chapter 4.

3.15. Stakeholder Conflict Resolution: The Sales department wants customer data organized by territory, while Customer Service wants it organized by support tier. How would you approach resolving this conflict during conceptual design?

3.16. Connection to ER Modeling: Explain how the conceptual design process you learned in this chapter directly prepares you for creating ER diagrams in Chapter 4. What specific connections exist between conceptual design activities and ER modeling techniques?

Integration Questions

3.17. End-to-End Process: Trace how a business requirement moves through the complete process: from stakeholder interview → requirements analysis → conceptual modeling → ER diagram. Use a specific example, such as "professors teach courses."

3.18. Quality Assurance Application: You've created your first ER diagram for a course registration system. Describe how you would apply the four quality standards to evaluate your diagram before presenting it to stakeholders.

3.19. Professional Preparation: Based on what you learned in this chapter, what skills and knowledge do you now have that prepare you for professional database design work? What do you still need to learn in the upcoming chapters?

3.20. Chapter Integration: This chapter focused on conceptual design principles. How do you expect these principles to guide your work in Chapter 4 (ER Modeling) and Chapter 5 (Enhanced ER Modeling)? What connections do you anticipate?

References and Recommended Readings

Foundational Sources

Batini, C., Ceri, S., & Navathe, S. B. (1992). *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings Publishing Company.

Chen, P. P. (1976). The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1), 9-36.

Contemporary Textbooks

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of Database Systems* (7th ed.). Pearson.

Mancas, C. (2015). *Conceptual Data Modeling and Database Design: A Fully Algorithmic Approach*. Apple Academic Press.

Teorey, T. J., Lightstone, S. S., Nadeau, T., & Jagadish, H. V. (2011). *Database Modeling and Design: Logical Design* (5th ed.). Morgan Kaufmann.

Requirements Analysis and Communication

Robertson, S., & Robertson, J. (2012). *Mastering the Requirements Process: Getting Requirements Right* (3rd ed.). Addison-Wesley.

Alexander, I., & Stevens, R. (2002). *Writing Better Requirements*. Addison-Wesley.

Business Process and Stakeholder Management

Dennis, A., Wixom, B. H., & Tegarden, D. (2020). *Systems Analysis and Design: An Object-Oriented Approach with UML* (6th ed.). Wiley.

Sommerville, I. (2015). *Software Engineering* (10th ed.). Pearson.

Quality Assurance and Validation

Wiegers, K., & Beatty, J. (2013). *Software Requirements* (3rd ed.). Microsoft Press.

Hull, E., Jackson, K., & Dick, J. (2017). *Requirements Engineering* (4th ed.). Springer.

Database Design Methodology

Simsion, G. C., & Witt, G. C. (2005). *Data Modeling Essentials* (3rd ed.). Morgan Kaufmann.

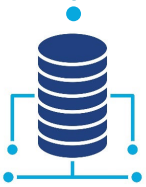
Hernandez, M. J. (2013). *Database Design for Mere Mortals: A Hands-On Guide to Relational Database Design* (3rd ed.). Addison-Wesley.

Professional Standards and Best Practices

DAMA International. (2017). *DAMA-DMBOK: Data Management Body of Knowledge* (2nd ed.). Technics Publications.

Date, C. J. (2019). *Database Design and Relational Theory: Normal Forms and All That Jazz* (2nd ed.). O'Reilly Media.

Chapter 4



4. Entity-Relationship (ER) Modeling

4.1 The Art of Seeing Data Relationships

On a university campus, you see students heading to classes, professors carrying papers, and busy department offices. Each person, place, and interaction shows how the university operates. As a database designer, your job is to capture these activities and organize them into a properly structured format that a computer can process and manage efficiently.

The **Entity-Relationship (ER) model** is a tool for representing these scenarios. Developed by **Dr. Peter Chen in 1976** (Chen, 1976), the ER model provides a visual language for describing the essential elements of any business domain and their relationships. Instead of mapping roads and landmarks, it maps data and the relationships between different pieces of information.

For example, when a student named Sarah registers for a Database Systems course taught by Professor Koneski in the Computer Science department, multiple relationships are involved. Sarah is an **entity** with attributes such as her student ID, address, and birth date. The course has its own **attributes**, including the course code, credit hours, and description. Sarah's enrollment in the course creates **relationships** with properties such as the enrollment date and grade.

ER modeling accurately represents real-life situations and remains easy to understand for both technical and non-technical stakeholders. For example, when we say, "students enroll in courses, and we need to track their grades," we can visualize students and courses as rectangles, connected by a diamond to show the enrollment relationship.

4.2 Introduction to ER Modeling

The **Entity-Relationship (ER)** model is fundamental to **conceptual database design**. It provides a systematic approach to modeling real-life data requirements (Elmasri & Navathe, 2017). Introduced by Dr. Peter Chen in 1976, the ER model translates business requirements into a visual blueprint for database development.

ER modeling bridges the gap between human understanding of information and how computers store and process data. For example, when we say, "students enroll in courses and receive grades," we need a formal method to accurately represent this relationship and its details for proper database implementation.

> **The ER Model Foundation**
>
> The ER model accomplishes this through **three fundamental constructs**:
> - **Entities** represent objects or concepts of interest.
> - **Attributes** describe the properties of entities.
> - **Relationships** are associations between entities.

This chapter focuses on **Chen notation**, the original and most detailed approach to ER modeling. Chen notation provides the required accuracy for complex database design while remaining understandable to business stakeholders.

Learning Approach: In this chapter, we use the **University Management System** and **Company/Business Database** models as examples to demonstrate each ER modeling concept. The diagrams include all attribute types, relationship patterns, cardinality constraints, and different entity types to ensure complete coverage. Each concept is illustrated with specific examples from both case studies, showing how abstract ideas apply to real-life situations. The complete examples of these two case studies are provided in **Chapter 11: Integrated Case Studies**.

4.3 The Role of ER Diagrams in Database Design

ER diagrams serve **multiple roles** in the database design process and are *valuable tools* for database professionals (Teorey et al., 2011). The case studies presented in this chapter demonstrate these roles in practice.

Communication and Notation: ER diagrams provide a **visual language** that enables effective communication between database designers, business analysts, and stakeholders (Batini et al., 1992). For example, when discussing student enrollment requirements, referencing the relationship between STUDENT and COURSE entities *removes uncertainty* about which data to collect and how different pieces of information connect.

Design Validation: Before implementing, ER diagrams help us **test our data model** against real-life situations. Can our university model represent a student withdrawing from a course? How do we handle a professor teaching multiple sections of the same course? Can our company model include an employee managing multiple orders while working in a specific warehouse and being supervised by a department? These validations help us catch design flaws early in the development process.

Implementation Blueprint: ER diagrams offer a solid starting point **for relational database implementation**. Each entity is mapped to a table, each attribute becomes a column, and each relationship determines the foreign key connections necessary to ensure data integrity.

System Evolution: As business requirements change, ER diagrams serve as the **foundation for evaluating and implementing modifications**. The visual representation

makes it easier to *assess the impact of changes* and ensure that modifications preserve system consistency.

4.4 Chen Notation: The Language of Precise Design

Dr. Peter Chen introduced his notation in 1976 to solve a critical issue in database design: creating a visual language that is both *accurate* for technical purposes and *intuitive* for business users. The solution he developed became the standard notation known **as Chen notation**.

4.4.1 Historical Foundation and Design Principles

The idea behind Chen's original paper, "The Entity-Relationship Model - Toward a Unified View of Data" (Chen, 1976), came from his observation that existing data modeling techniques were either *too abstract* for business users or *too implementation-specific* for conceptual design. He needed a notation that could bridge **this gap**, providing mathematical precision while remaining visually accessible.

The **strength of Chen notation** comes from its use of simple geometric shapes, each representing a *specific meaning*. A rectangle isn't just a box. It represents an **entity**, a distinct object or concept that we want to store information about. A diamond isn't just a decorative element; it symbolizes a **relationship**, the meaningful connection between entities. Chen carefully chose shapes that would be easy to draw, recognize, and reproduce across various media and cultures.

> **Chen Notation Core Elements**
>
> Dr. Peter Chen introduced his notation to address the need for a modeling approach that is both **mathematically accurate** and **visually intuitive**. Chen's notation uses different geometric shapes. Each shape carries a specific semantic meaning:
>
> - **Rectangles** represent entities
> - **Diamonds** represent relationships
> - **Ovals** represent attributes
> - **Lines** connect these parts and indicate constraints
>
> This geometric vocabulary creates a **modeling language** that directly translates into database implementation constructs.

4.4.2 Advantages of Chen Notation

Chen's notation offers a **few advantages** over other available modeling methods:

Explicit Weak Entity Representation: Chen’s notation uses **double rectangles** to identify weak entities. Weak entities are entities that depend on other entities for identification. Alternative notations usually don’t handle this explicitly; they do that *implicitly*, making it easy to overlook essential dependencies.

Comprehensive Attribute Types: The notation provides **different symbols** for simple, composite, multivalued, and derived attributes. This makes sure that implementation teams understand how to handle each piece of information and data.

Clear Relationship Semantics: Diamonds explicitly represent relationships as **first-class modeling constructs**. That makes it easier to identify relationship attributes and understand complex relationship patterns.

Direct Relational Mapping: Each Chen notation element corresponds **directly to relational database constructs**, reducing translation errors during implementation.

4.4.3 Comparison with Alternative Notations

As noted earlier, Chen notation is our main modeling method. To explain why we chose this approach, we review the alternative modeling techniques:

Crow's Foot Notation: This notation highlights *implementation details* and cardinality, and is widely used in industry (Connolly & Begg, 2015). It does not explicitly represent weak entities or multivalued attributes.

UML Class Diagrams: UML supports inheritance hierarchies for object-oriented systems but requires additional notation to represent database-specific concepts, such as weak entities.

IDEF1X: This systematic and detailed notation is effective for large enterprises, though its complexity may overwhelm stakeholders.

Chen notation offers an effective **balance** of correctness and approachability for database design.

4.5 ER Diagram Symbols and Components

4.5.1 Symbol Set

Chen notation uses a set of **geometric shapes**, each carrying a *specific meaning* (Chen, 1976).

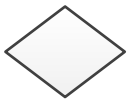
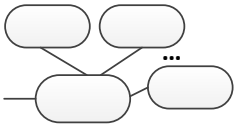
Table 4.1 summarizes the notation for ER diagrams.

4.5.2 Connection Types and Constraints

Lines connecting symbols carry important information about business rules:

- **Single lines** specify *partial participation* (optional relationship participation)
- **Double lines** specify *total participation* (mandatory relationship participation)
- **Cardinality labels** (1, N, M) specify **relationships** between entities
- **Composite attribute trees** show attribute hierarchies

Table 4.1: Summary of the Notation for ER Diagrams

Symbol	Meaning	Description
	Entity	*Represents an object or concept*
	Weak Entity	*An entity dependent on another for identification*
	Relationship	*Association between entities*
	Identifying Relationship	*A relationship that helps identify a weak entity*
	Attribute	*Property or characteristic*
	Multivalued Attribute	*An attribute that can have multiple values*
	Derived Attribute	*Calculated from other attributes*
	Key Attribute	*Uniquely identifies entity instances*
	Composite Attribute	*Show attribute hierarchies*

4.5.3 Naming Conventions in ER Modeling

Uniform naming conventions assure clarity, prevent SQL implementation issues, and maintain *professional standards* (Connolly & Begg, 2015). Our case studies show these conventions in practice.

Entity Naming Conventions

1. Use UPPERCASE for entity names

- University: STUDENT, COURSE, PROFESSOR, DEPARTMENT, CLASS_SESSION
- Company: EMPLOYEE, CUSTOMER, PRODUCT, ORDER, ORDER_LINE

2. Use singular nouns

- University: STUDENT *(not STUDENTS)*, COURSE *(not COURSES)*
- Company: EMPLOYEE *(not EMPLOYEES)*, CUSTOMER *(not CUSTOMERS)*

3. Multi-word entities use underscores

- University: CLASS_SESSION *(not CLASSSESSION)*
- Company: ORDER_LINE *(not ORDERLINE)*

Attribute Naming Conventions

1. Use First letter capitalized with underscores

- University: Student_id, First_name, Birth_date, Enrollment_date
- Company: Employee_id, Order_date, Unit_price, Quantity_on_hand

2. Be descriptive while concise

- **Good**: Birth_date, Phone_number, Email_address, Order_date
- **Poor**: bd, ph_no, email, ord_dt

3. Avoid abbreviations unless standard

- **Good**: Student_id, SSN *(Social Security Number)*, Employee_id
- **Poor**: std_id, fn *(first name)*, emp_id

Relationship Naming Conventions

1. Use lowercase with underscores

- University: enrolls_in, works_for, teaches, heads
- Company: works_for, processes, places, supplies, stored_in

2. Use verb phrases that read naturally

- University: STUDENT enrolls_in COURSE, PROFESSOR teaches COURSE, PROFESSOR heads DEPARTMENT
- Company: EMPLOYEE works_for DEPARTMENT, CUSTOMER places ORDER, SUPPLIER supplies PRODUCT

Alternative Naming Styles (Not Used in This Book)

While we use **snake_case**, you should be aware of other common conventions:

- **PascalCase**: StudentId, FirstName, OrderDate, ProductName
- **camelCase**: studentId, firstName, orderDate, productName
- **kebab-case**: student-id, first-name *(not recommended for databases)*

We chose the **First_letter_capitalized** approach because:

- Clear visual difference between words
- No SQL quoting issues

- Professional appearance in ER diagrams
- Consistent with entity naming patterns

Consistency is important

After you select a naming convention, use it **consistently in your ER model**. Our case studies illustrate consistent application of these conventions across all entities, attributes, and relationships.

General Naming Rules

1. Examples:

- **Entities**: STUDENT, ACADEMIC_PROGRAM, COURSE_ENROLLMENT, EMPLOYEE, CUSTOMER_ORDER, PRODUCT_SUPPLIER
- **Attributes**: First_name, Last_name, Birth_date, Phone_number, Employee_ID, Order_date, Unit_price
- **Relationships**: enrolls_in, works_for, belongs_to, processes, supplies

2. Don't use spaces in names, as they generate problems during SQL implementation

Note: If you use spaces in names, SQL requires them to be enclosed in quotes; otherwise, they can cause maintenance issues.

4.6 Entities: The Foundation of Data Models

Entities are the **basic building blocks** of any Entity-Relationship (ER) model. They represent objects, concepts, and other elements that need information stored. Understanding entities is crucial because they form the foundation for all relationships, attributes, and constraints in a database (Elmasri & Navathe, 2017). Comprehensive University and Company case studies demonstrate each type of entity in practical, real-world contexts.

4.6.1 What is an Entity?

Definition: Entity

An **entity** represents an abstraction of a real-life object of a particular kind. It can be a living being, a physical object, an event, a location, or an abstract concept that exists independently and can be uniquely identified within the analyzed system. It is an object of special importance to the analyzed system that users want to store data in.

Entities represent, describe, and classify the **objects of interest** in the analyzed system. An entity is the fundamental building block of an ER diagram/model.

Key Characteristics of Entities:

1. **Independence**: Can exist on its own *(for strong entities)*
2. **Uniqueness**: Each instance can be distinctly identified
3. **Business Relevance**: Important to the system's purpose
4. **Data Storage**: Requires multiple pieces of information to be stored

4.6.2 Categories of Entities

Depending on the nature of the analyzed system (the real-world environment being modeled), entities can take **many forms**:

Living Beings:

- People: employee, student, patient, doctor, worker, professor, salesperson, engineer
- Animals: livestock, pets, wildlife specimens
- Plants: crops, trees, specimens

Physical Objects:

- Tangible items: machine, book, car, computer, document, product, medicine, person, house
- Documents: invoice, receipt, contract, report, certificate
- Infrastructure: equipment, furniture, vehicles, tools

Events and Activities:

- Business processes: sale, invoicing, teaching session, examination, medical checkup
- Temporal events: meeting, conference, competition, project milestone
- Transactions: purchase, payment, transfer, enrollment, sale

Locations and Places:

- Geographic entities: country, city, village, region
- Physical spaces: warehouse zone, office, classroom
- Address components: postal code areas, regions

Abstract Concepts:

- Organizational units: department, company, branch
- Financial concepts: account, budget, exchange rate, loan
- Services: consultation, warranty, subscription, membership
- Academic concepts: course, degree program, research project

4.6.3 Entity Types and Instances

An **entity type** is a group of objects sharing the same characteristics. Each individual occurrence within this group is called an **instance**.

> **Definition: Entity Type vs. Entity Instance**
>
> An **entity type** represents a collection of objects that share the same characteristics; it is the template or blueprint that defines the structure.
>
> An **entity instance** (or simply *instance*) is each specific occurrence; it is a concrete example that follows the template defined by its entity type.

University System Example

The **STUDENT entity type** represents all students (objects with shared characteristics). When we mention the **STUDENT** entity, we're referring to the entire group of students, both named and generalized, using a single term. However, when we mention the student "Sarah Smith, Student ID: S001," we're referring to a specific student, a precisely defined member of the student group, which is an instance (concrete occurrence) of the **STUDENT** entity.

STUDENT (entity type) vs. "Sarah Smith, Student_ID: S001" (instance)

Company System Example

Similarly, **CUSTOMER** represents an entity type that includes all customers in the business system. "Ana Ivanovska, Customer_ID: C1" would be a specific example of the **CUSTOMER** entity type.

CUSTOMER (entity type) vs. "Ana Ivanovska, Customer_ID: C1" (instance)

4.6.4 Unique Identification of Entity Instances

Each individual instance of a given entity must be **uniquely identified,** so it differs from all other instances in the entity set.

University System Example

Although two or more students at the same university may share the same first and last names, each is a separate instance of the STUDENT entity and differs from others based on a unique characteristic, such as Student_ID or a national identification number.

Company System Example

Two employees might share the same name, but they are distinguished by a unique **Employee_ID** or email address. Likewise, two customers might have the same name but are differentiated by **Customer_ID**.

Each instance of an entity type has specific values for its characteristics. Individual instances differ based on the values of one or more characteristics, while for the other characteristics, they may have either different or the same values.

Key Concepts for Entity Identification

To understand how entities are uniquely identified, we need to introduce the concept of **keys** - special attributes used as identifiers.

> **Definition: Key Attributes**
>
> **Key attributes** are special attributes whose values uniquely identify each entity instance within an entity set, ensuring that no two instances share the same key value.

Types of Keys in ER Modeling:

1. Key Attribute (or simply "Key")

- An attribute whose values can uniquely identify each entity instance within an entity set

- No two instances of the same entity type can have the same key value
- They are represented by underlining the attribute name in ER diagrams
- Examples:
    - University case study: **Student_ID** for STUDENT entity, **Course_ID** for COURSE entity
    - Company case study: **Employee_ID** for EMPLOYEE entity, **Customer_ID** for CUSTOMER entity

2. Partial Key

- An attribute that can identify entity instances only within the context of their owner entity
- Used in *weak entities*
- Cannot uniquely identify instances by itself across the entire database
- Must be combined with the owner entity's key for complete identification
- Examples:
    - University case study: **Session_number** in CLASS_SESSION (only unique within a specific course)
    - Company case study: **Line_number** in ORDER_LINE (only unique within a specific order)

3. Composite Key

- Multiple attributes combined to form a unique identifier
- Used when no single attribute can serve as a unique identifier
- Common in weak entities: owner's key + partial key = composite key
- Examples:
    - University case study: **Course_ID + Session_number + Year** uniquely identifies CLASS_SESSION
    - Company case study: **Order_ID + Line_number** uniquely identifies ORDER_LINE

Key Properties:

- **Uniqueness**: No two entity instances can have the same key value
- **Stability**: Key values should not change frequently over time
- **Minimality**: Use the smallest set of attributes necessary for unique identification
- **Non-null**: Key attributes must always have values (cannot be empty)

> **Important Note**
>
> While we identify key attributes in ER modeling, the specific choice of primary key occurs during the logical design phase when mapping to relational schemas (covered in Chapter 7).

4.6.5 Recognizing Entities in Requirements

To identify entities when analyzing requirements, we look for nouns that represent objects of interest to the system. We can ask these questions:

Entity Recognition Questions

1. **Is it important to the system?** Does the system need to store information about this object?
2. **Does it exist independently?** Can it exist without depending on another entity for its identity?
3. **Do we need multiple instances?** Will there be more than one occurrence of this type of object?
4. **Can instances be uniquely identified?** Is there a way to distinguish one instance from another?

University System - Entity Recognition Process

Requirements Statement: *"A university consists of multiple faculties. Each faculty offers multiple courses. Students enroll in particular courses and receive grades. Each course is taught by a professor from the affiliated faculty."*

Identified Entities:

- **FACULTY** - organizational units that offer courses
- **COURSE** - academic subjects offered by faculties
- **STUDENT** - individuals who enroll in courses
- **PROFESSOR** - faculty members who teach courses

Why are these entities:

- Each represents objects of importance to the university system
- Each exists independently (a student exists whether or not enrolled in courses)
- Multiple instances exist (many students, many courses, etc.)
- Each instance can be uniquely identified (student ID, course code, etc.)

Company System - Entity Recognition Process

Requirements Statement: *"Our company has multiple departments with employees. Employees process customer orders. We sell products to customers who place orders. Each order contains multiple products supplied by various suppliers and stored in warehouses."*

Identified Entities:

- **EMPLOYEE** - people who work for the company
- **DEPARTMENT** - organizational divisions within the company
- **CUSTOMER** - individuals or organizations that purchase products
- **PRODUCT** - items offered for sale
- **ORDER** - purchase requests from customers
- **SUPPLIER** - organizations that provide products

- **WAREHOUSE** - facilities where products are stored

4.6.6 Entity Classification: Strong vs. Weak Entities

In ER modeling, entities are classified into two major categories according to their identification requirements and dependency relationships. For effective database design and implementation, it is important to distinguish between the two.

Entity Relationship Dependencies

Analyzing the relationships among entities is crucial for determining their classification.

- **Independent entities** can exist alone and serve as the foundation of the model
- **Dependent entities** require other entities to have a meaningful existence

The entity can be classified as strong or weak, based on these relationships.

This classification is important for implementing entities in relational databases and dictates the referential integrity constraints required to ensure data validity.

> **Definition: Strong vs. Weak Entities**
>
> **Strong entities** exist independently and have their own unique identification mechanism based on key attributes.
>
> **Weak entities** depend on other entities for both existence and identification. They require a combination of their own partial key and their owner entity's key to be fully identified.

4.6.7 Strong Entities

Strong entities form the backbone of our ER models. They have the following characteristics:

> **Characteristics of Strong Entities:**
>
> 1. **Independence**: Can exist on its own without depending on other entities
> 2. **Self-Identification**: Have their own unique key that can identify each instance
> 3. **Foundation Role**: Serve as the foundation that other entities can reference
> 4. **Business Importance**: Represent primary business concepts or objects

Examples from Our Case Studies:

University System Strong Entities:

- **STUDENT**: Individual students can exist independently in the university system
- **PROFESSOR**: Faculty members exist as independent individuals
- **COURSE**: Academic courses are independent educational offerings
- **FACULTY**: Organizational educational units that exist independently
- **TEXTBOOK**: Physical or digital books that exist as independent resources

🏢 ***Company System Strong Entities:***

- **EMPLOYEE**: Workers exist as independent individuals in the company
- **CUSTOMER**: Clients exist independently, whether they place orders or not
- **PRODUCT**: Items exist independently in the company's catalog
- **SUPPLIER**: Partner companies exist independently
- **WAREHOUSE**: Storage facilities exist as independent locations
- **ORDER**: Business transactions that exist as independent events
- **DEPARTMENT**: Organizational divisions that exist independently

4.6.8 Weak Entities

Weak entities depend on strong entities for their existence and identification. They have the following characteristics:

> **Characteristics of Weak Entities:**
>
> 1. **Dependency**: Cannot exist without its owner entity (strong entity)
> 2. **Partial Identification**: Have partial keys that are only unique within their owner's context
> 3. **Identifying Relationship**: Always connected to their owner through a special relationship
> 4. **Combined Identification**: Require both their owner's key and their own partial key for complete identification

Examples from Our Case Studies:

🎓 ***University System Weak Entity:***

CLASS_SESSION: Cannot exist without a COURSE. A class session only makes sense in the context of a specific course. Session_number "1" and Year "2025" have meaning only when we know they belong to a particular Course_ID "CS101" or "MA301".

🏢 ***Company System Weak Entity:***

ORDER_LINE: Cannot exist without an ORDER. A line item only makes sense as part of a specific order. Line_number "Line1" has meaning only when we know it belongs to a particular **Order_ID** "Order1" or "Order3".

Why These Are Weak Entities:

CLASS_SESSION Dependency:

- If a COURSE is deleted, all its CLASS_SESSIONs should also be deleted;
- A Session_number like "1" is only meaningful within a specific course (**Course_ID)** and Year;
- You cannot have a class session without knowing which course and year it belongs to.

ORDER_LINE Dependency:

- If an ORDER is deleted, all its ORDER_LINEs should also be deleted;
- A Line_number like "Order1" is only meaningful within a specific order (Order_ID);
- You cannot have an order line without knowing which order it belongs to.

4.6.9 Graphical Representation

In Chen's notation, entities are represented differently based on their type, making it easy to distinguish between strong and weak entities.

Strong Entity Representation

Strong entities are represented by a **single rectangle** containing the entity name in **CAPITAL LETTERS** (see Figure 4.1 a) and b))

Figure 4.1 a) University System Strong Entities Examples

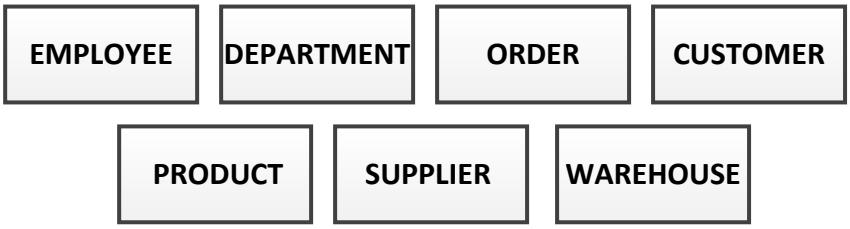


Figure 4.1 b) Company System Strong Entities Examples

Weak Entity Representation

Weak entities are represented by a **double rectangle** containing the entity name in **CAPITAL LETTERS**. The double border immediately signals that this entity depends on another entity for its existence and identification (see Figure 4.2 a) and b)).

Figure 4.2 a) University System Weak Entity Examples

Figure 4.2 b) Company System Weak Entity Examples

Visual Difference

The visual difference between single and double rectangles serves an important purpose:

Strong vs. Weak Entity Visual Cues

- **Single Rectangle**: Immediately identifies independent entities that form the foundation of the model
- **Double Rectangle**: Immediately identifies dependent entities that require special handling in database implementation

Strong vs. Weak Entity Visual Comparison is displayed in Figure 4.3.

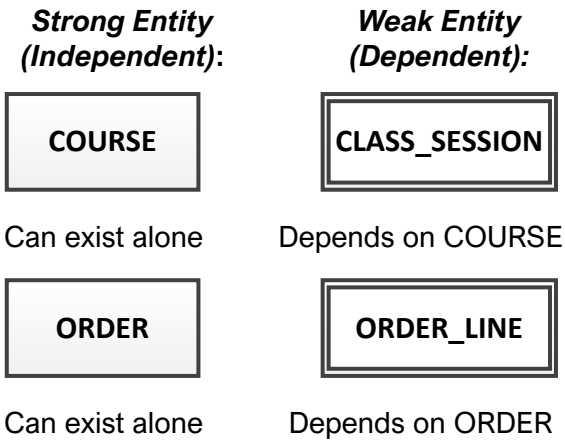


Figure 4.3: Strong vs. Weak Entity Visual Comparison

4.6.10 Naming Best Practices

Entity names should be sufficiently **generic** to easily adapt to future changes in the observed system.

Generic Naming Examples:

Poor Example: If a bookstore currently sells only physical books, it might seem logical to create an entity called PHYSICAL_BOOK in the ER diagram.

Better Example: However, considering that the bookstore might expand to sell e-books, audiobooks, magazines, and other reading materials in the future, it's better to name the entity PUBLICATION or PRODUCT.

University System Examples:

- Use COURSE rather than MATH_COURSE (allows for courses from all faculties)
- Use ACADEMIC_PROGRAM rather than BACHELOR_PROGRAM (accommodates also master's, and doctoral programs)

Company System Examples:

- Use PRODUCT rather than SOFTWARE_PRODUCT (allows for hardware, services)
- Use CUSTOMER rather than INDIVIDUAL_CUSTOMER (accommodates corporate customers)

Naming Conventions

- Entity names should be **singular nouns**
- Use **UPPERCASE** for entity names
- Use **descriptive** names, but keep them concise
- Avoid **abbreviations** unless they are standard
- Being **generic enough** to enable future growth

4.6.11 Common Mistakes When Identifying Entities

Knowing common pitfalls helps avoid design errors:

Mistake 1: Confusing Entities with Attributes

- **Wrong**: Creating separate entities for concepts that should be attributes
- **Example**: Creating FIRST_NAME and LAST_NAME entities instead of making them attributes of PERSON

Mistake 2: Creating Entities for Every Noun

- Not every noun appearing in the requirements represents an entity
- **How to test**: Ask, "Do we need to store multiple pieces of information about this object?"

Mistake 3: Overspecification

- **Wrong**: Creating separate entities for similar concepts in the early stage of the design
- **Example**: Creating UNDERGRADUATE_STUDENT and GRADUATE_STUDENT entities initially
- **Better**: Start with STUDENT entity; add specialization later if needed

Mistake 4: Missing Important Entities

- Failing to identify important objects that are implicitly mentioned in requirements
- **Solution**: Ask "What other objects are necessary for this system to function?"

Mistake 5: Incorrect Weak Entity Identification

- **Wrong**: Making every dependent object a weak entity
- **How to test**: Ask "Does this entity need the owner entity's key for identification?"

Foundation Principle

Understanding entities is very important because they form the foundation of the entire ER model. Every relationship, attribute, and constraint in the diagram will connect back to these fundamental building blocks.

4.7 Attributes: The Characteristics of Entities

Attributes are the fundamental components of an ER diagram that describe entities. They represent the characteristics, properties, and facts about each entity instance. Understanding attributes is important because they determine what information our database will hold and how that information will be organized and constrained.

4.7.1 What is an Attribute?

An **attribute** represents a property or characteristic of an entity or relationship that is important to the system. Attributes describe the details of entities; they provide meaning and context to our database model.

Attributes are the descriptive details of our data model. When we say "Student Sarah Smith has ID S001, was born on August 25, 1995, and lives in Ohrid," we're describing the **STUDENT** entity through its attributes: **Student_ID**, **First_name**, **Last_name**, **Birth_date**, and **City**.

They translate abstract concepts of entities into practical information. For example, a STUDENT entity without attributes is just a placeholder. Attributes add useful details about entities, capturing real-world specifics such as a unique identifier (Student_ID), the student's name, and other personal and administrative information.

Key Characteristics of Attributes

- **Descriptive Nature**: Attributes describe properties of entities. They don't exist independently
- **Value Assignment**: Each attribute has a specific value for each entity instance
- **Domain Constraint**: Attributes are restricted to specific types of valid values
- **Business Relevance**: Attributes capture information important to the purpose of the system

4.7.2 Attributes in Our Case Studies

To illustrate the attributes, we will provide examples from our University and Company databases using real case-study data.

When we review a university student's record, we see how attributes work together to create a complete picture. Consider **Anica Stojnasen**, a computer science student in our university system with **Student_ID S006**. Her **STUDENT** entity instance has multiple attributes that describe her: her unique **Student_ID** connects her to all university systems, her **Birth_date** (15.05.2003) establishes her age (22, calculated in 2025 when this chapter is written) and academic eligibility, her composite **Name** allows both formal and informal communication, and her multivalued **Email** attribute includes both her university email (anica.stojansen@university.edu) and personal contact (anica.s@gmail.com).

Similarly, in our company system, when we review an employee like **Zoran Pejovski** (**Employee_ID: E1**), we see how attributes capture both administrative requirements and

practical business needs. For example, we store information about his salary (60000), his hire date (15.01.2020), or his phone numbers so the company can reach him.

Understanding Attribute Categories

Our case studies demonstrate that different entities require different attribute types. Some attributes, like **Student_ID** or **Product_ID**, are used as simple identifiers that remain unchanged throughout the entity's lifetime. Others, like addresses, need to be broken down into components for practical use. For example, a university needs to know the complete addresses of its students to contact them by mail, if needed, or to identify international students. A company needs city information for local tax calculations. Some attributes, like phone numbers, can support multiple values because real people and businesses maintain multiple contact methods for different purposes.

The following examples show how attributes support business operations in both systems:

University System Attributes Examples:

- **STUDENT**: Student_ID (S001, S002), Name (composite for formal/informal use), Birth_date (for age verification), Email (multivalued; support multiple phone numbers)
- **COURSE**: Course_ID (CS101, MA301), Title ("Intro to Programming"), Credits (5-9 for degree planning), Description (detailed course information)
- **PROFESSOR**: Professor_ID (P001, P002), Office (A201, B305 for location), Hire_date (for tenure tracking)

Company System Attributes Examples:

- **EMPLOYEE**: Employee_ID (E1, E2), Salary (60000, 55000 for payroll), Hire_date (for benefits), Phone (multivalued for business contact)
- **CUSTOMER**: Customer_ID (C1, C2), Name (composite for communication), Address (composite for shipping)
- **PRODUCT**: Product_ID (1, 2, 3), Product_name ("Ajvar", "Rakija"), Price (372, 682 for sales), Category ("Food", "Beverage" for classification)

4.7.3 Attribute Domains

Every attribute has an associated **domain** (also called a value set) that defines the set of legal values that the attribute can contain. Domains preserve data integrity by constraining attribute values to relevant and valid ranges.

Understanding Domains. A domain specifies:

- **Data Type**: What kind of values are allowed (numbers, text, dates, etc.)
- **Value Range**: The specific set or range of acceptable values
- **Formatting Rules**: How the domain values are structured
- **Constraints**: Additional constraints on values based on the real-world requirements

For a well-designed database, it is important to understand domains. They translate business rules into technical specifications and constraints. If a university specifies that a course's credit range is 5 to 9, this rule becomes a domain constraint. That way, we prevent invalid data entry. Similarly, when a company requires employee IDs to follow the format "E" followed by digits, this becomes a domain specification that establishes consistent authentication across all systems.

Domain Examples in Practice

Let's examine how domains function in our hypothetical university system. According to the administrative regulations, the **Student_ID** domain requires a capital letter 'S' followed by three digits (S001, S002, S003). The **Credits** domain has integer values between 5 and 9. The **Grade** domain uses values from 5 to 10, implementing the Macedonian educational grading system.

Our company system shows different domain requirements. **Employee_ID** starts with a capital letter 'E' followed by digits (E1, E2, E3), making employee identification simple. **Salary** values range from 25,000 to 100,000. **Product_Category** uses enumerated values like "Food", "Beverage", and "Dairy".

Table 4.2 illustrates domain specifications for selected attributes in the university system, showing how academic business rules translate into specific data constraints and validation requirements.

Table 4.2: University System Domain Examples

Attribute	*Type*	*Format/Range*	*Valid Values*	*Invalid Values*	*Business Rule*
Student_ID	Character string	'S' + 3 digits	S001, S002, S003, S004, S005	001, ST01, S12345	Unique student identification
Credits	Integer	5 to 9	5, 6, 7, 8, 9	0, 4, 10, 15, 3.5	Academic credit limits
Grade	Numeric	5 to 10	6, 7, 8, 9, 10	4, 11, 15, 2.5	Macedonian grading system
Birth_date	Date	DD.MM.YYYY	15.05.2003, 20.11.2002, 10.03.2004	2030-12-25, 1850-01-01, 15/05/03	Valid birth dates
Professor_Office	Character string	Letter + digits	A201, B305, C102, D404, E203	Room 301, Building A, 205	Campus office format

Table 4.3 illustrates the corresponding domain examples from the Company system, demonstrating how business needs drive different data formats and validation requirements.

Table 4.3: Company System Domain Examples

Attribute	*Type*	*Format/Range*	*Valid Values*	*Invalid Values*	*Business Rule*
Employee_ID	Character string	'E' + digits	E1, E2, E3, E4, E5, E6, E7, E8	EMP1, 1, E, Employee1	Staff identification format

Salary	Integer	25,000 to 100,000	25000, 30000, 35000, 40000, 45000, 50000, 55000, 60000	-1000, 1000000, 15000	Company salary range
Product_Category	Enumerated list	Predefined list	Food, Beverage, Dairy, Accessories, Apparel, Souvenir	Electronics, Software, Unknown	Product classification
Order_Status	Enumerated list	Predefined list	Shipped, Pending, Processing, Canceled	Delivered, Draft, Unknown	Order lifecycle states
Phone_Number	Character string	+389XX/XXX-XXX	+38970/212-789, +38975/812-547, +38977/612-348	070-212-789, +1-555-1234, phone	Macedonian phone format

These tables show how domains maintain data consistency. The University system uses structured academic formats like student IDs and grade ranges, while the Company system focuses on business needs such as salary ranges, order statuses, and communication formats. Each domain reflects the specific requirements and limitations of its business environment.

These domain specifications serve as the foundation for choosing data types and implementing constraints during relational database design, which we will examine in detail in Chapter 7: Relational Model.

Domain Implementation Aspects

Domains are usually defined during logical database design and are generally not shown directly in ER diagrams. However, it is important to understand domain constraints during conceptual modeling to detect potential data quality issues and conflicts with business rules. For example, if student birthdates indicate ages over 80, this could point to data entry errors.

The enforcement of domain constraints ultimately occurs through database implementation, but considering domains during ER modeling helps avoid common data quality issues and ensures that the conceptual model accurately represents business requirements.

Understanding domains is important for ensuring data quality and maintaining compliance with business rules throughout the database lifecycle.

4.7.4 Categories of Attributes

Based on the structure of the attributes, characteristics of the values, and derivation methods, we can classify them into a few categories. For proper ER modeling and database implementation, it is important to understand these categories.

Attribute Classification Framework

Attributes are organized based on their:

- **Structure**: How the attribute is organized (simple vs. composite)
- **Characteristics of the values**: How many values the attribute can hold (single vs. multivalued)
- **Derivation methods**: How the attribute value is obtained (stored vs. derived)

4.7.4.1 Simple (Atomic) Attributes

Simple attributes are indivisible values that cannot be broken down into smaller components within their business context. These attributes have specific, well-defined purposes and meanings as complete units.

The concept of "atomic" depends on the business context and the way the data will be used. The attribute **Birth_date**, with the value "15.05.2003", is a simple attribute in most contexts. We treat it as a single piece of information for age calculations, which can be used for different purposes, e.g., eligibility checks. While the date could theoretically be decomposed into day, month, and year components, doing so would not serve a useful business purpose in our database systems.

Simple Attributes in Our Case Studies

We will explore and explain simple attributes in more detail using our case studies. They show how simple attributes support various business operations. In the University system, we use the simple attribute **Birth_date** to calculate current age and verify minimum age requirements for certain programs. These operations all work with the complete date value, making decomposition unnecessary. Course **Credits** is a single numeric value to represent the academic value of a course. The **Enrollment_date** is a temporal reference point for administrative purposes.

The Company system shows similar patterns. The single attribute **Salary** represents employ's compensation as a single monetary value. The **Weight** attribute captures a product's physical mass. **Order Status** specifies the current state of a customer order.

The **COURSE** entity effectively employs multiple simple attributes. The **Title** is a simple attribute that stores course names as descriptive text, for example, "Intro to Programming" or "Database Systems". While these titles contain multiple words, they function as single, atomic units because we treat them as indivisible course identifiers. The **Credits** attribute stores simple numbers representing each course's academic weight, such as 5, 6, or 7.

Design of the Simple Attributes

Simple attributes require minimal storage, enable straightforward comparisons and calculations, and translate directly to database columns. However, deciding what values and information should remain simple needs careful consideration of current and future data usage patterns.

Simple Attribute Design Questions

When designing simple attributes, ask these types of questions:

- Will we ever need to search or sort by components of this data?
- Do business rules apply to parts of this information?
- Will reporting requirements need to access sub-components?
- If the answer to any of these questions is yes, consider making the attribute composite rather than simple.

Graphical Representation

Simple attributes are represented by **single ovals** connected to their entity with straight lines.

Figure 4.4 demonstrates the Chen notation representation of simple attributes across both our case study domains.

Examples of simple attributes in our University System (Figure 4.4 a):

- **Student_ID:** Simple attribute that contains values like "S001", "S002", "S003". They represent identifiers that cannot be divided within the university context
- **Birth_date:** Stores complete dates like "15.05.2003" as single, indivisible values
- **Credits:** Contains numeric values like 5, 6, or 7 representing the academic weight of courses

Examples of simple attributes in our Company System (Figure 4.4 b):

- **Employee_ID:** Contains values like "E001", "E002", "E003". They represent the business identifiers of employees
- **Salary:** Stores monetary values like 35000, 45000, 55000 as single amounts
- **Order_date:** Simple attribute that contains complete dates like "23.08.2024" representing when orders were placed

Business Setting Dependency:

Figure 4.4 emphasizes that **atomicity depends entirely on business requirements**. The same type of information, such as dates, appears as simple attributes in both systems because both universities and companies treat dates as indivisible units for their business needs. For their primary business processes, neither system requires breaking dates into separate day, month, and year components.

This graphical illustration guides database designers in making **appropriate granularity decisions.** If the business processes we model need to search, sort, or analyze components of these values separately, we have to reconsider the attribute structure.

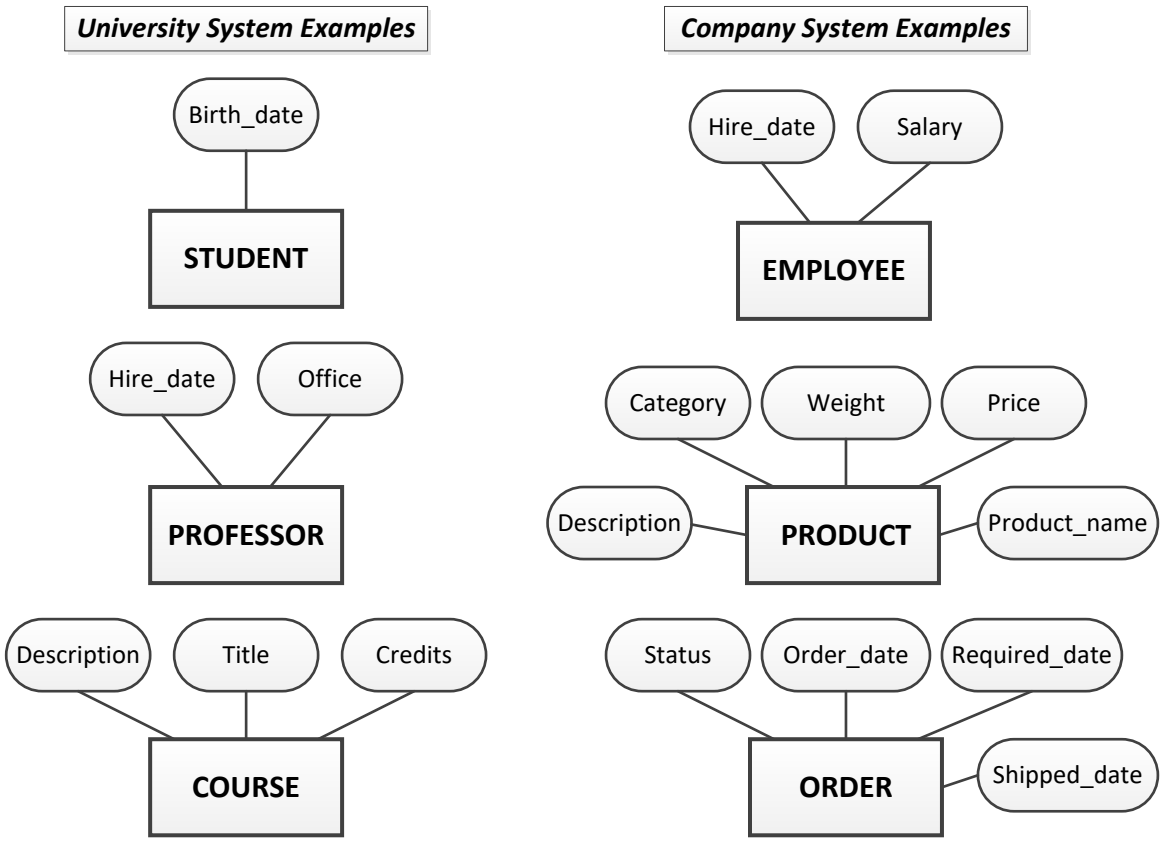


Figure 4.4: Examples of Simple Attributes: a) University system example, b) Company system example

4.7.4.2 Composite (Complex) Attributes

Composite attributes represent the sophisticated side of data modeling. They contain multiple related pieces of information, each meaningful on its own, but more meaningful when combined. Composite attributes reflect the reality that many real-world characteristics are naturally hierarchical or consist of multiple parts (Elmasri & Navathe, 2016).

When someone identifies themselves as "Sarah Smith", we understand this as containing two distinct but related parts of information. The *first name* is personal and informal, and the *last name* is familial and formal. In database terms, this creates a composite attribute named **Name** that contains the sub-attributes First_name and Last_name.

Composite attributes offer **flexibility in how we use the data**. For formal documents, we need the complete name. Sometimes we need just the first name for personal communication, and sometimes just the last name for alphabetical sorting. A well-designed composite attribute provides all these options while preserving the logical relationship between components.

> **Definition of the Composite Attribute**
>
> A **composite attribute** is an attribute that can be divided into smaller sub-attributes, each with independent meaning. These sub-attributes can be accessed individually or used together as a complete unit.

Composite Attributes in Our Case Studies

To illustrate the composite attributes, we will use the example of the Address, a widely understood concept. Let’s consider Elena Mitan's address in our university system: "Turisticka 22, Ohrid, 6000, North Macedonia". This information contains four components:

- **Street**: "Turisticka 22"
- **City**: "Ohrid"
- **Postal_code**: "6000"
- **Country**: "North Macedonia"

Each component of this attribute serves different purposes. For some local event notifications, the university administration only needs the city information. For billing, the financial department requires the full address, while the postal code can be used for demographic analysis. Country information can be used to filter international students.

In the company's system, we record the addresses of employees, customers, and suppliers, which follow the same structure as in the university example but serve different business purposes.

Business Settings and Design Decisions

The decision on when and how to implement composite attributes depends on **how the system will use that information**. Names can benefit from being composite because, for some business purposes, it is necessary to access first and last names separately. Alternatively, you can store the first and last names as simple attributes, for example, "Sarah Smith." However, every time we need to separate first and last names, the system is forced to use string parsing, which is very inefficient.

The address attribute, as previously explained, consists of multiple components. Whether to make it simple or composite depends on how the system will use that information. If the system requires geographical analysis or uses the data for shipping optimization, a composite attribute structure will be more beneficial. However, if the system only uses the information to print the complete address on envelopes or labels, a simple address attribute will suffice. Nonetheless, this approach will limit future flexibility.

Composite Attribute Design Questions

When deciding whether to make an attribute composite, consider:

- **Will different business processes need different components?** (Names for sorting vs. display)
- **Do reporting requirements analyze sub-components separately?** (Sales by city, students by country)
- **Will search functionality need to filter by components?** (Find all customers in specific postal codes)
- **Do business rules apply differently to different parts?** (International shipping vs. domestic)

Implementation of the composite attributes

Composite attributes in ER modeling represent the logical structure without specifying specific implementation methods. During logical design, they often become multiple columns in relational tables. However, when used as composite attributes in the ER diagram during conceptual design, they are visualized as a single piece of information composed of multiple components. This helps database designers accurately capture business requirements, so that technical constraints are better understood and implemented.

For example, the composite Name attribute will become separate First_name and Last_name columns, while Address will become Street, City, Postal_code, and Country columns.

Graphical Representation:

Composite attributes are shown with **ovals linked to their sub-attributes in a tree structure**. This layout demonstrates the hierarchical connection between the main attribute and its parts.

Figure 4.5 illustrates the Chen notation representation of composite attributes from both case study systems.

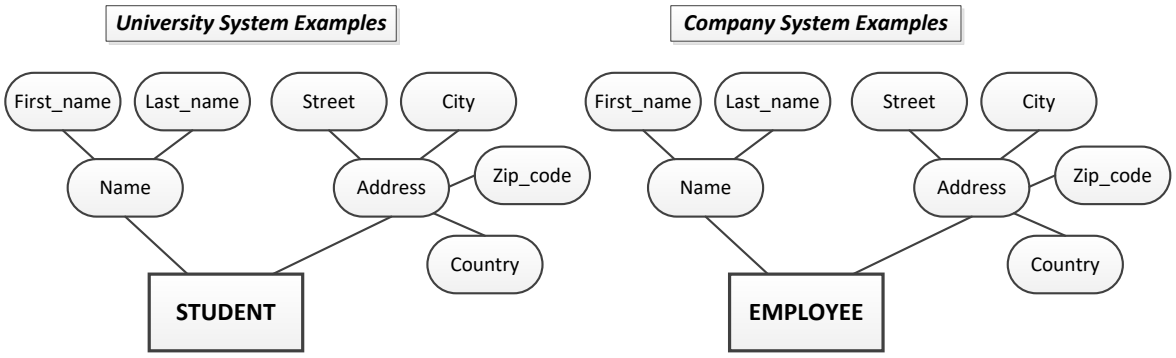


Figure 4.5: Composite Attributes Examples

The **tree structure** uses a parent-child relationship to connect a composite attribute to its sub-attributes. This helps database designers understand the composite attribute and all its individual components.

The same type of information appears in both use cases and appears identical in both systems. This illustrates that the same information can be used in different database systems, but the sub-attributes have different business purposes in each scenario.

4.7.4.3 Multivalued Attributes

The need for **multivalued attributes** comes from the reality that people and businesses communicate in multiple ways and have different interests or qualifications. (Elmasri & Navathe, 2016). The purpose of the multivalued attribute is to visualize and illustrate this complex information pattern.

Using the university case study, we can consider the student Elena Mitan and her two email addresses: one official, from the university domain (elena.mitan@university.edu), and another personal Gmail account (elena.mitan@gmail.com). The university needs both of them, the first for official communication, and the second to reach her if she doesn't check her university email regularly, especially during breaks. So, both emails serve different purposes in her academic life.

The question is how to store this information in a database. As separate information? To store just the official email address? This choice will prevent us from reaching her in an emergency if she is not responding to official email. This is where multivalued attributes come into play and solve this dilemma. They are used to store multiple values in a single entity.

Multivalued Attribute Definition

A **multivalued attribute** is an attribute that can hold multiple values for a single entity instance. The total number of values reflects the variety of information patterns, so it can vary across instances of the entities.

Multivalued Attributes in the Company Case Study

In the company, employees can have multiple phone numbers. Consider Zoran Pejovski, our sales manager. He needs multiple contact numbers for different tasks. His mobile phone handles client calls or is used when he travels or works remotely. His office phone is used for routine communication. So, the company needs both of them to ensure business continuity. The multivalued attribute structure supports these requirements.

Constraints in the Multivalued Attributes

Multivalued attributes follow **set semantics**, which means that their order doesn't matter and duplicate values are not allowed (Connolly & Begg, 2015). In our university case study, the student Elena Mitan has two email addresses. The number of values in the multivalued attribute depends on the business needs and reflects real-world situations. Some students have only a university email address; others have both a university and a personal email address; and some even have multiple personal email accounts. The database system should resolve all these options, without requiring each student to have the same number of email addresses.

Multivalued Attribute Design Guidelines

When considering multivalued attributes, evaluate:

- **Natural multiplicity**: Does the business naturally produce multiple values? (Contact methods, skills, languages, functions)
- **Manageable cardinality**: Will the number of values remain reasonable? (Typically, 2-10 values per entity)
- **Similar purpose**: Do all values have the same general function? (All communication methods, all certifications)
- **Implementation impact**: Can the system effectively handle the multiple values during processing?

Multivalued Attributes Implementation

Multivalued attributes need special treatment during database implementation. They cannot be stored directly in relational tables and must be normalized into separate tables (Date, 2003). For illustration, the multivalued Email attribute in the university database system is implemented as a separate EMAIL table with columns Student_ID and Email. In the company database system, we implement the EMPLOYEE_PHONE table with columns Employee_ID and Phone_number to store the multivalued phone attribute.

This implementation detail influences ER design choices. Database designers must ensure that this multivalued structure addresses business needs and does not introduce unnecessary complexity during implementation. Chapter 7 (Relational Model) explains in detail how to manage multivalued attributes.

Graphical Representation

Multivalued attributes are represented by **double ovals.** This representation distinguishes them from simple attributes (represented by a single oval) and composite attributes (represented by a hierarchical oval). This helps database designers normalize these attributes during logical design and plan any additional tables and relations needed. The **double oval notation** indicates that these attributes require special treatment during database implementation.

Figure 4.6 shows the Chen notation for multivalued attributes in both case studies.

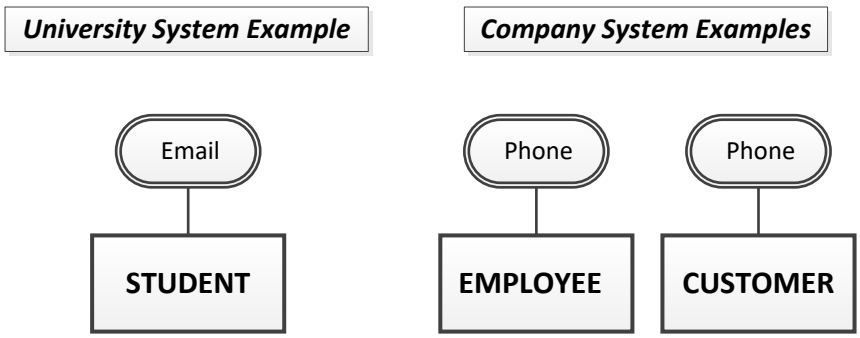


Figure 4.6: Multivalued Attributes Examples

4.7.4.4 Derived Attributes

Derived attributes address an important design principle: **avoid storing information in databases that can be calculated on demand from existing data**. Derived attributes are not stored in database tables; they are calculated from existing attributes and represent logical information (Elmasri & Navathe, 2016).

An example from the university case study is a student's age, which can be calculated and derived from their Birth_date attribute and the current date. Age is information that the university needs for various administrative purposes. However, age changes daily, so the value stored in a database becomes invalid within 24 hours. This makes it an ideal candidate for a derived attribute. Whenever we need the student's age, we can calculate it from the Birth_date attribute, which is constant and unchanging.

The database designer, while developing the company's conceptual database design, may consider adding a derived Total_amount attribute to the ORDER entity. It can be calculated from the existing Unit_price and Quantity attributes using the simple formula: Unit_price × Quantity. The rationale behind this is that the total amount changes whenever order lines are added, modified, or removed, making it a good candidate for a derived attribute.

> **Derived Attribute Definition**
>
> A **derived attribute** is an attribute whose value is calculated from other attributes, entities, or relationships in the database. These attributes are not stored directly in a database. They are calculated on demand, when needed, ensuring accuracy and eliminating data redundancy.

University System Examples:

STUDENT Entity - Age Attribute:

- **Source**: Birth_date attribute and current date (2025)
- **Calculation**: Current_year - Birth_year (with month/day adjustments)
- **Examples**:
  - Anica Stojnasen with Birth_date "15.05.2003" → Age = 22 (in 2025)
  - David Boskovski with Birth_date "25.01.2001" → Age = 24 (in 2025)
  - Sara Smith with Birth_date "25.08.1995" → Age = 30 (in 2025)

STUDENT Entity - GPA Attribute (Grade Point Average):

- **Source**: All course grades and credit hours from enrollment records
- **Calculation**: Σ(Grade_points × Credits) / Σ(Credits) for all completed courses
- **Business Value**: Important graduation eligibility, scholarship determination

Company System Examples:

ORDER Entity - Total_amount Attribute:

- **Source**: ORDER_LINE records associated with the order
- **Calculation**: SUM(Unit_price × Quantity) for all order lines
- **Examples**:
  - Order 1: Line 1 (372 × 2) = 744
  - Order 3: Line 1 (930 × 3) = 2790
  - Order 6: Line 1 (9300 × 6) = 55800

EMPLOYEE Entity - Years_of_service Attribute:

- **Source**: Hire_date attribute and current date
- **Calculation**: Current_date - Hire_date (converted to years)

- **Business Value**: Supports benefits calculation, promotion eligibility, and retirement planning

Benefits of Derived Attributes

Derived attributes provide many advantages in database systems (Date, 2003). The first is data consistency, meaning they always display the correct calculated value based on the current source data. They also **eliminate redundancy**, which reduces storage requirements. With derived attributes, there is **no need to manually update calculated values**; they are calculated as needed and presented to the user accurately.

Let's exemplify this using our university system. To determine the age of the student Anica Stojnasen, we can derive it from her birth date. This information is stored in the Birth_date attribute of the STUDENT entity, and for this particular student has a value of '15.05.2003'. The calculated result is Age = 22 (in 2025 when this chapter is written). This calculation is performed each time the information is needed to ensure accuracy, so the administrative staff never needs to remember to update age values.

In the company database system, we can find even more complex derived attributes. Examples include total order amounts, discount calculations, and tax implications, all calculated using appropriate formulas based on the existing attributes. By deriving these values rather than storing them, the system maintains accuracy even when business rules change or calculation methods are updated.

Design of Derived Attributes

Derived attributes are not stored directly in the database tables. But including them in the conceptual design and ER diagram, and documenting them in the early stages of database development, is very important from multiple perspectives. They represent business information requirements and provide and document the calculation methods and formulas for future implementation. They can also identify performance-critical calculations that require optimization strategies such as indexing or caching.

When multiple entities or complex rules are involved, calculations can affect query performance. If we deal with calculated values that we need to access frequently, we can cache them so we don't recalculate them each time. These situations require balancing the accuracy of derived values with performance requirements.

Database designers decide whether to include derived attributes in the ER diagram based on their significance and the **complexity of their calculations**. Simple derived attributes, like age, can be omitted because they are obvious. But complex calculations, such as total order amounts or financial summaries, need to be documented and included in ER diagrams to obtain accuracy during implementation.

Implementation of Derived Attributes

There are several approaches for derived attributes implementation (Connolly & Begg, 2015):

- **Real-time calculation**: The values are calculated during the execution of the query to obtain the maximum accuracy
- **Materialized views**: Values are pre-calculated and stored, with periodic refresh

- **Triggers**: The values are updated automatically when source data changes
- **Application logic**: The calculations are done at the application layer

Graphical Representation

Derived attributes are represented by **dashed ovals**.

Figure 4.7 illustrates the Chen notation representation of derived attributes from both case study systems.

Derived attributes often depend on data from **multiple sources. For example,** the Age attribute requires the Birth_date attribute and the current date, while the Total_amount attribute requires aggregating multiple ORDER_LINE records. This characteristic of derived attributes affects both conceptual design decisions and physical implementation strategies.

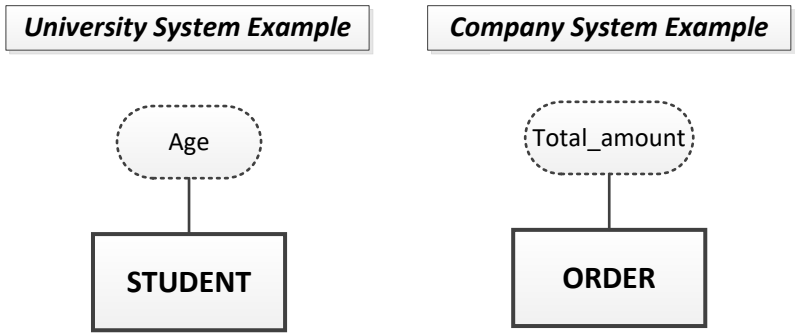


Figure 4.7: Derived Attributes Examples

4.7.5 Key Attributes

Key attributes are used to identify entities and provide a foundation for database relationships and integrity constraints (Elmasri & Navathe, 2016). There are different types of key attributes, each serving a different purpose: providing **unique entity identification**, establishing **referential integrity**, preventing duplicate entity instances, and optimizing data retrieval and storage operations.

When Sarah Smith enrolls at our university, she is assigned a permanent identifier to use throughout her entire education. In our university case study, the Student_ID key attribute is the identifier, and the value assigned to Sara is "S001". She uses it for various purposes in her academic life, including registering for courses, recording grades, and administrative interactions. Unlike attributes such as her name, address, or telephone number, which can change over time, this identifier remains stable.

The university database might handle a million tuples. But Student_ID "S001" will always refer to the student Sarah Smith, providing a stable foundation for the database structure. Regardless of whether Sarah changes her first or last name, address, or phone number, this key attribute will always point to the same person. This way, it is possible to support changes to specific student data naturally without compromising the database's integrity.

> **Key Attribute Definition**
>
> Key attributes are special attributes in an ER diagram that uniquely identify all instances of a particular entity. Thus, no two instances have the same key value. They are the primary mechanism for distinguishing one instance of an entity from all others.

Types of Keys in ER Modeling

1. Simple Keys

Simple keys consist of a single attribute that uniquely identifies an entity instance. For example, the university database for the STUDENT entity has the key attribute Student_ID with values "S001", "S002", "S003", and so on (as described in Chapter 11: Integrated Case Studies). The database for our company case study for the EMPLOYEE entity has the key attribute Employee_ID with values "E1", "E2", "E3", and so on (as described in Chapter 11: Integrated Case Studies), which uniquely identify the company's employees. These simple keys provide an efficient identification mechanism implemented directly in the database as primary keys (explained in detail in Chapter 7: Relational Model).

2. Composite Keys

When a single attribute is not sufficient to uniquely identify an entity instance, multiple attributes are combined into a composite key. This is most common in weak entities, where identification requires both the owner's key and the dependent entity's partial key. As an example, consider the CLASS_SESSION entity, which is uniquely identified by the attributes Course_ID, Session_number, and Year. The value "1" of the Session attribute, by itself, has no meaning and must be associated with a specific course and year to provide unique identification.

3. Natural Keys

Natural keys are concepts that have real-world meaning beyond identifying entities in a database. For example, the ISBN attribute of the TEXTBOOK entity in a university database is a unique, internationally standardized identifier for books, not just in our database. (Date, 2003). Natural keys are the best unique identifiers if they exist and remain unchanged over time.

4. Surrogate Keys

When there is no natural identifier for an entity, a **Surrogate key** is added to the database. These keys are generated by the system and hold no meaning outside the database. The Student_ID and Employee_ID attributes in both case studies are Surrogate keys. They do not represent any real-world characteristics of the entities they refer to, but they allow their efficient and consistent identification. These keys improve database performance and are unaffected by changes to business rules.

While this chapter focuses on conceptual modeling, it's helpful to understand that key attributes in ER models eventually become primary keys in relational databases. These keys enable relationships between entities through foreign key references. This implementation process is explored in detail in Chapter 7: Relational Model.

Identifying Weak Entities Using Partial Keys

Weak entities have **Partial key attributes** that uniquely identify their instances (Connolly & Begg, 2015). These keys exist only in weak entities and do not serve as unique identification at the database level. To fully identify a weak entity, multiple attributes must be combined, as previously explained for composite keys.

For example, let's consider the CLASS_SESSION entity in our university database system. This entity represents individual class meetings for courses. The **Session_number** attribute is a partial key with values 1, 2, 3, and so on. These values represent the particular class meetings within each course. It's obvious that Session "1" alone doesn't uniquely identify the CLASS_SESSION entity across the database. This session appears in multiple courses, such as CS101, CS201, and MA301. So, to uniquely identify the CLASS_SESSION, it must be combined with other attributes. In our university database, the complete identification of a CLASS_SESSION combines the owner entity's key (Course_ID) with the partial key (Session_number) and Year. This creates a composite key "CS101-1-2025" or "MA301-2-2025" that provides unique identification across the entire university database system. The **partial key handles identification within the owner's scope**, while the **composite key handles identification across the entire database**.

ORDER_LINE is an example of a weak entity in our company database system. The **Line_number** attribute is a partial key with values 1, 2, 3, and so on. These values represent individual line items within orders. Line "1" appears in multiple orders, such as Order 1, Order 2, and Order 3. To provide a complete identification for this weak entity, we combine attributes Order_ID and Line_number to create a unique composite identifier, such as "Order1-Line1" or "Order5-Line3".

Key Selection Principles

When choosing the key attributes, several factors should be considered:

Stability is the most critical factor**.** Key values must remain stable during the entity's lifetime. The key attribute, Student_ID, remains constant throughout a student's education. Other attributes, such as email addresses, might change frequently as students graduate or switch providers.

Meaningfulness involves choosing between natural and surrogate keys. Natural keys like ISBN have real-world meaning. They are useful for human users and cross-system integration. However, natural keys can change or become unavailable, forcing system modifications. Surrogate keys lack business meaning but provide stable, consistent entity identification that is resistant to external changes.

Size and Performance are related to key selection in large-scale systems. When selecting a key for the EMPLOYEE entity, we evaluate multiple options. The surrogate key Employee_ID is an integer and requires minimal storage, and provides optimal indexing performance. The natural key Social Security Numbers require more space and process more slowly. Large-scale systems process millions of transactions, so these factors and considerations are critical.

Business Rules and Policies can also influence key selection decisions. Privacy regulations discourage the use of Social Security Numbers as primary identifiers. For

databases intended for international use, we are selecting keys that work across different countries.

Choosing the correct key for each entity involves balancing all of the factors listed above.

Important Distinction: Key Attributes vs. Primary Keys

To prevent confusion during design and implementation, it is important to understand the concepts and relationships between ER modeling keys and relational database primary keys.

The **key attributes** just described **are included in ER diagrams** as attributes that can uniquely identify entities, without requiring their implementation. For the same entity, multiple key attributes may exist, allowing for different forms of identification.

Primary keys are selected during the logical design of a database for actual implementation. They are explained in detail in Chapter 7: Relational Model.

The ER modeling process involves identifying all possible key attributes and understanding their characteristics. This serves as the basis for selecting primary keys during the logical design. For example, a STUDENT entity might have key attributes such as Student_ID, Social_Security_Number, and Email (if unique). In the logical design phase, we can decide which one to use. After careful analysis of stability, privacy, and performance considerations, we can choose Student_ID as the primary key. We can recognize other attributes as alternative unique identifiers.

Graphical Representation:

Key attributes are represented by **underlined text** within their ovals. Partial key attributes use **dashed underlining**.

Figure 4.8 illustrates the Chen notation representation of key attributes from both case study systems.

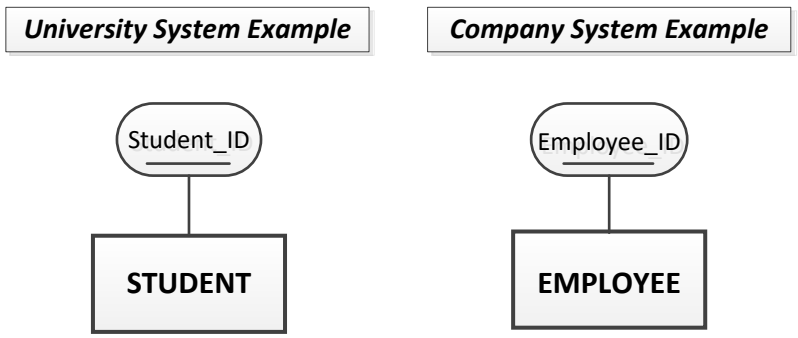


Figure 4.8: Key Attributes Examples

Figure 4.9 illustrates the Chen notation representation of partial key attributes, showing ovals with dashed underlined text connected to weak entities.

This visual distinction between key attributes helps database designers identify which keys will become primary or foreign keys during implementation. (Chapters 6 and 7)

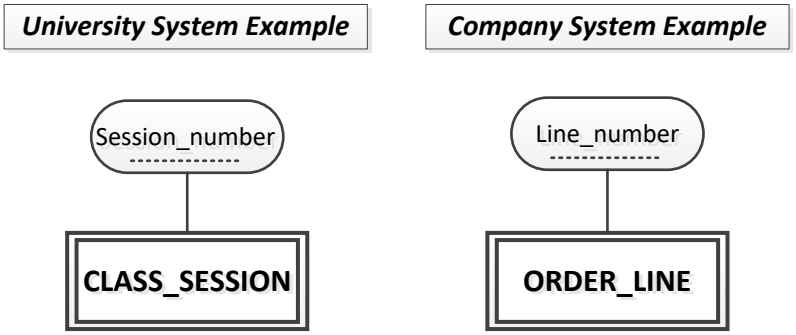


Figure 4.9: Partial Key Attributes Examples

4.7.6 Integrated Attribute Examples

To understand how different attribute types work together in practice, let's examine complete entity examples from our case studies. These examples demonstrate how entities combine **simple, composite, multivalued, derived, and key attributes** to create data models that support real business operations (Elmasri & Navathe, 2016).

Real databases use different types of attributes to represent the complexity of their entities. The goal is to create entity structures that naturally support all required business operations while sustaining data integrity and system performance. Our case studies illustrate this idea.

University System Integration

Consider how a STUDENT entity in our university system (see Figure 4.10) demonstrates attribute integration. The Student_ID is a simple key attribute that uniquely identifies students. It provides a foundation that links all student information across the university's systems. The Name attribute has a more complex, composite structure, consisting of two sub-attributes: First_name and Last_name. This design approach enables both formal documentation ("Smith, Sarah") and informal communication ("Hi Sarah"). Email is modeled as a multivalued attribute to accommodate both university and personal addresses. Age is a derived attribute calculated from the stored Birth_date attribute and the current date to provide up-to-date information without redundant storage. The Address is another composite attribute, consisting of Street, City, Country, and Zip_code components.

This approach reflects how actual universities function. The administrative office needs **consistent identifiers** to track students. They also prefer separating students' first and last names to allow both informal and formal communication. For reliable communication, multiple contact methods are available. All these attributes together define and model the STUDENT entity.

The **COURSE entity** combines various patterns and attributes. A Course_ID as a simple key provides consistent, stable identification of courses. Title and Description attributes use simple text storage and serve as complete units in academic communication. Credits are a simple attribute that represents a unit of academic value.

Company System Integration

The Company system shows that business entities require different combinations of attributes. An **EMPLOYEE entity** (see Figure 4.10) combines employee identification (Employee_ID), contact information through composite addresses and multivalued phone numbers, employment details (Salary, Hire_date), and organizational relationships. This structure supports **payroll processing, emergency contact procedures, organizational reporting, and human resources management**.

The **ORDER entity** has a unique identifier called Order_ID. Status tracking uses enumerated values such as Shipped, Pending, and Canceled. Date management is handled through three attributes: Order_date, Required_date, and Shipped_date, which are used for workflow control and potential total calculations.

ORDER_LINE entities (see Figure 4.11) illustrate how weak entities depend on their owners. while maintaining descriptive attributes for specific business functions. The partial key Line_number provides identification within each order, while product references, quantities, and pricing information capture transaction details essential for fulfillment and accounting.

ORDER_LINE entities (see Figure 4.11) demonstrate how weak entities rely on their owners. The partial key Line_number offers identification within each order, while product references, quantities, pricing, and discount information record transaction details.

Integration Patterns by Business Domain

Different business domains require different attribute integration patterns:

- **Academic systems**: Emphasize stable identification, flexible communication, and temporal tracking for student lifecycle management
- **Business systems**: Focus on transaction processing, financial tracking, and operational efficiency for commercial operations
- **Healthcare systems**: Prioritize patient safety, regulatory compliance, and comprehensive medical history tracking
- **Government systems**: Emphasize citizen identification, service delivery, and regulatory compliance requirements

Attribute Synergy Examples

***University System - STUDENT Entity Integration*:**

- **Key + Composite + Multivalued**: Student_ID (S001) + Name (Sarah, Smith) + Email {sarah.smith@university.edu, sarah@gmail.com} creates comprehensive identity management
- **Simple + Derived**: Birth_date (25.08.1995) + Age (30 in 2025) provides both stable storage and current information

- **Composite + Simple**: Address (Street: "Jane Sandanski 32", City: "Ohrid", Country: "Macedonia") + Status ("Active") supports both location services and administrative tracking

Company System - EMPLOYEE Entity Integration:

- **Key + Simple + Multivalued**: Employee_ID (E1) + Salary (60000) + Phone ("070-212-789") enables payroll processing and comprehensive communication
- **Composite + Simple**: Name (Zoran, Pejovski) + Hire_date (15.01.2020) supports both formal documentation and tenure tracking
- **Simple + Derived**: Hire_date (15.01.2020) + Years_of_service (5 in 2025) provides both historical record and current employment metrics

Graphical Representation

Complete entity examples, including all their attributes, provide comprehensive information about the entity.

Figure 4.10 illustrates complete entity structures from both case study systems. It shows how different attribute types integrate together to create comprehensive data models.

Figure 4.11 illustrates examples of weak entities. It shows how partial keys combine with other attribute types to create complete dependent entity structures.

> **Understanding Figure 4.10: Complete Entity Integration**
>
> The figure shows complete entity structures with **all attribute types working together**: solid ovals (simple), underlined ovals (keys), hierarchical structures (composite), double ovals (multivalued), and dashed ovals (derived).

University System Examples (4.10 a):

STUDENT Entity contains the following attributes:

- **Student_ID** (key, underlined), used for unique identification
- **Name** (composite): First_name + Last_name hierarchy
- **Birth_date** (simple): Stable temporal reference point
- **Age** (derived, dashed): Current information without storage redundancy
- **Email** (multivalued, double oval): Comprehensive communication coverage
- **Address** (composite): Street + City + Zip_code + Country for geographic services

Company System Examples (Figure 4.10 b):

EMPLOYEE Entity shows business-optimized integration:

- **Employee_ID** (key, underlined): Stable business identifier
- **Name** (composite): Professional naming structure
- **Address** (composite): Street + City + Zip_code + Country for geographic services
- **Hire_date** (simple): Employment history information

- **Salary** (simple): Payroll processing requirement
- **Phone** (multivalued, double oval): Multiple contact capabilities
- **Email** (simple): Communication coverage

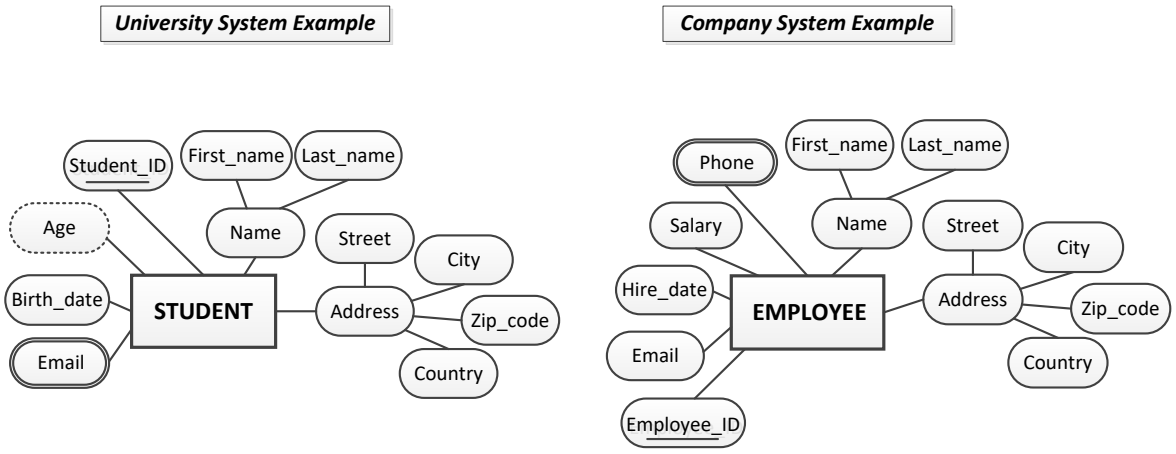


Figure 4.10: Entities and Their Attributes Examples; a) University System Example; b) Company System Example

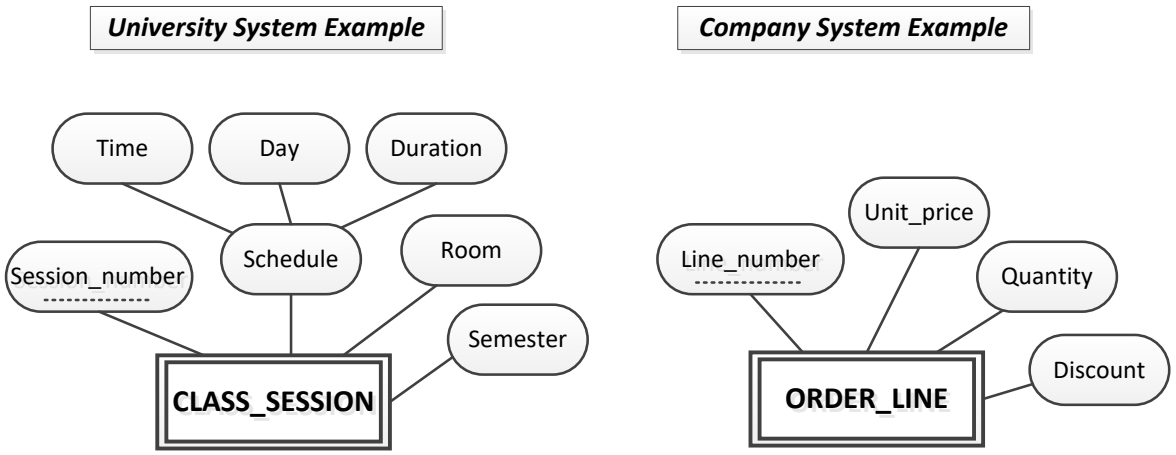


Figure 4.11: Weak Entities and Their Attributes Examples

> **Understanding Figure 4.11: Weak Entity Integration**
>
> The figure shows how weak entities combine partial keys with other attributes. Note that entities are represented as double rectangles, and partial keys are highlighted with dashed underlines.

4.7.7 Attribute Design Principles

The design of attributes depends on how the data will be used throughout the system's lifetime. It is often necessary to strike a balance between current needs and the system's future flexibility.

Attribute Design Strategies

1. Selecting a Key Attribute

When selecting a key attribute, consider factors that influence the current implementation and long-term maintenance of the system (Connolly & Begg, 2015):

Stability: Choose keys whose values do not change frequently over the lifetime of the entity.

Performance and Efficiency: When selecting key attributes, consider storage requirements and indexing efficiency.

Business Policy: The selected key attributes should support regulatory requirements.

2. Composite Attributes

The difference between simple and composite attributes lies in how the system uses the information. If the system needs to access and use information that consists of multiple components, such as a student's first and last name, the information can be modeled as a composite attribute rather than a simple attribute.

3. Multivalued Attributes

If the information we are modeling naturally contains multiple values, we can model it as a Multivalued Attribute. A student or employee may have multiple phone numbers and multiple email addresses. An employee may perform multiple functions.

4. Derived Attributes

Data that changes frequently should not be stored in the database; instead, it is calculated as needed from existing data. Derived Attributes, although not implemented in the database, are shown in the ER diagram and documented to indicate their need for the system's future existence. If they are simple and straightforward, they can be excluded from the diagram. An example is the Age attribute, which is obviously calculated from the existing Birth_date attribute. If the attribute's calculation requires complex logic, it is advisable to document it.

Common Design Mistakes

During the conceptual design and creation of the ER diagram, certain errors also occur (Elmasri & Navathe, 2016):

Over-decomposition means the information is unnecessarily broken into too many components. The decomposition of the address into the individual words of which it is composed is an example of this.

Under-decomposition is the opposite of the previous problem, in which information is not decomposed into the necessary and useful parts. For example, modeling the address as a simple attribute limits the possibility of its further geographical analysis, shipping optimization, etc.

Inconsistent domain specification refers to different approaches used to model the same data type across different entities. For example, in our case, the attribute name should be modeled as a composite attribute for both entities, STUDENT and PROFESSOR.

Choosing the right key attributes is especially important because it can have long-term effects on system performance. If a complex key selected changes often, it needs frequent updates. Key choices have lasting impacts on system design and maintenance costs.

The Goal of Attribute Design

To achieve a successful design of attributes, a thorough understanding of the problem being modeled, the users' needs, and the system's limitations is required. The ultimate goal is to design attributes for each entity that provide a complete and valid description, while ensuring data integrity, good system performance, and adaptability to future system requirements.

4.8 Relationships: The Connections That Give Meaning to Data

Relationships represent the most essential aspect of ER modeling. They are the connections and dependencies that exist between entities in the real world (Chen, 1976). While entities represent objects of interest and attributes describe their properties, **relationships explain how entities interact**.

Relationships are expressed as verbs in problem descriptions and represent the interdependencies among two or more entities. When we say "Sarah enrolls in Database Systems course," "Professor Kliment Ohridski teaches Computer Science courses," or "Order 5 contains multiple product line items," we are describing relationships as fundamental interconnections among entities in the system. They are not abstract concepts, but real processes, rules, and interdependencies through which companies and institutions function.

Without relationships, entities with their attributes would be just collections of data. Relationships link these pieces of data into a connected system. For example, when isolated from the rest of the system, the STUDENT entity, with its attributes, represents only a collection of personal data. But if we connect it to the COURSE, CLASS_SESSION, and FACULTY entities, it becomes part of the academic ecosystem, enabling the system to function as a whole.

4.8.1 What is a Relationship?

A **relationship** is an association, connection, or meaningful interaction between two or more entities in the real world that is important to the system being modeled (Elmasri & Navathe, 2016).

Relationships represent the fundamental business logic that governs how organizations operate. The "enrolls_in" relationship between STUDENT and COURSE in our university database is not just a connection between two tables. It offers information about a complex process through which students gain knowledge, and the system tracks their progress. This relationship includes, for example, information about the enrollment process (when it occurs), prerequisites, grades achieved, and status (current enrollment or withdrawal).

Key Characteristics of Relationships

- **Business Meaning**: Relationships represent real-world connections that have significance to the institution or business
- **Structural Dependencies**: They define how entities depend on or interact with each other
- **Rule Enforcement**: Relationships include business rules and operational constraints
- **Information Flow**: They enable data to move meaningfully across entity boundaries
- **Process Support**: Relationships support organizational workflows and business processes

4.8.2 Relationships in Our Case Studies

To demonstrate how relationships function in practice, we will examine examples from our University and Company database systems.

University System Relationship Examples

In our university system, relationships create the academic infrastructure that connects students, courses, faculty, and administrative structures into an organized educational environment. The **"enrolls_in" relationship** between STUDENT and COURSE entities represents the process by which students select and enroll in courses, and the university tracks their progress.

The **"teaches" relationship** between PROFESSOR and COURSE entities enables course assignment management, workload distribution, and academic quality oversight by connecting faculty expertise with curricular requirements.

Company System Relationship Examples

The Company system demonstrates how relationships support complex business operations. The **"places" relationship** between CUSTOMER and ORDER entities connects customer needs with business processes.

The **"contains" relationship** between the ORDER and ORDER_LINE entities shows a structure in which complex transactions decompose into detailed components. Order 1 contains multiple line items, each with specific product references, quantities, and pricing.

Relationship Attributes

Relationships might have their own attributes that provide additional description of the process involving all the entities that participate in that relationship. The "enrolls_in" relationship includes **Grade** and **Enrollment_date attributes**, representing the grades students earn for courses they are enrolled in, and the temporal aspect of the connection, the date of enrollment. The "stored_in" relationship between PRODUCT and WAREHOUSE includes **Quantity_on_hand and Reorder_level attributes**, supporting inventory management and supply chain operations.

It is important to note that some information doesn't belong to an entity. For example, a grade doesn't belong to the student entity (which course is related to?), or to the course

entity (who is the student who earned that grade?), but arises from the relationship between them. Similarly, inventory quantities don't belong to products or warehouses. They result from their relationship.

4.8.3 Graphical Representation

In Chen notation, relationships are represented by **diamonds** containing the relationship name. They are connected to participating entities via lines that include cardinality and participation constraints (Date, 2003).

Dr. Chen chose the diamond shape to represent the **dynamic nature of relationships.** Relationships exist only in the context of their participating entities, and the diamond symbol communicates this dependency through its connecting structure.

> **Relationship Naming Convention**
>
> **In this book, relationship names will be placed next to the diamond symbol** rather than inside it. Note that both conventions are acceptable: relationship names can appear either inside the diamond or next to it, depending on design preferences and space constraints.

Basic Relationship Representation

Figure 4.12 illustrates the Chen notation for relationships. It shows diamonds connected to entity rectangles with labeled lines indicating cardinality constraints.

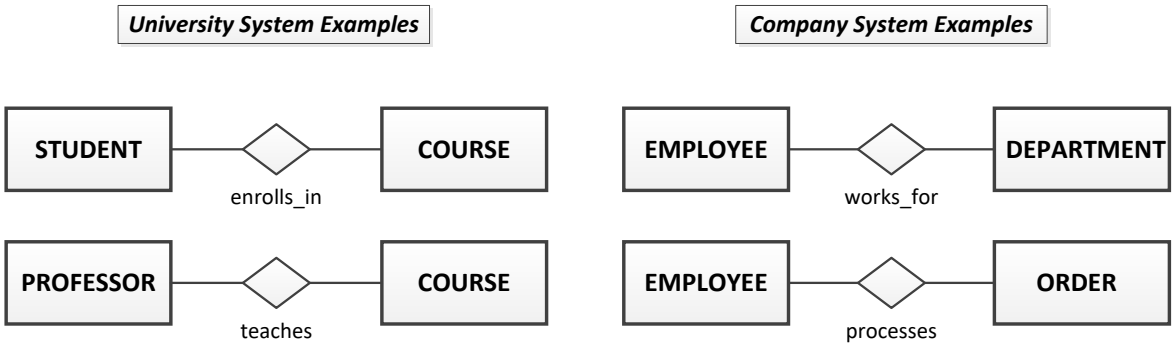


Figure 4.12: Relationships Examples

Relationship with Attributes

When relationships have their own attributes, and they are connected to the relationship diamond just as attributes are connected to entities. This demonstrates that relationships can have descriptive characteristics independent of the entities that participate in them.

Figure 4.13 shows relationship attributes connected to relationship diamonds, illustrating how associations can carry their own descriptive information.

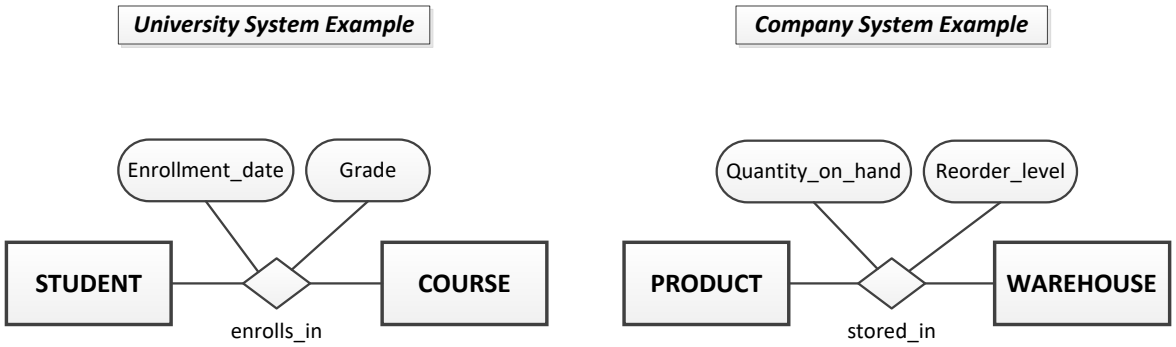


Figure 4.13: Relationships Attributes Examples

Understanding relationship representation and meaning provides a basis for exploring more complex relationship concepts, including degree, cardinality, participation constraints, and specialized relationship types. The visual notation system creates a common language for communicating business logic and system requirements across technical and non-technical stakeholders.

4.8.4 Degree of Relationships

The **degree** of a relationship refers to the number of entities that participate in the relationship (Elmasri & Navathe, 2016). Understanding relationship degree is important for proper modeling because it affects both the conceptual and logical designs and the actual database implementation.

Binary relationships involving two entities are the most common in database systems. **Ternary relationships**, involving three entities, capture more complex business scenarios that require specialized implementation approaches. **Higher-degree relationships**, while possible, are rare in practice. If they exist, they are often decomposed into multiple binary relationships to improve clarity and simplify implementation.

Relationship Degree Classification

- **Unary (Degree 1)**: Also called recursive relationships, involve a single entity type relating to itself
- **Binary (Degree 2)**: Most common type, involving exactly two entity types
- **Ternary (Degree 3)**: Complex relationships involving three entity types
- **Higher-degree (Degree 4+)**: Rare relationships involving four or more entity types

The choice of relationship degree directly affects the complexity of database implementation and query design. We'll explore the relational implementation of these concepts in **Chapter 7: Relational Model**.

4.8.4.1 Unary (Recursive) Relationships

Unary relationships, also called **recursive relationships**, involve a single entity type that relates to itself in different roles.

Recursive relationships naturally occur in many business situations where objects of the same kind have meaningful connections with each other. Examples include course prerequisites, product dependencies, and social networks, all of which involve entities relating to similar types.

University System Recursive Example

The **"prerequisite" relationship** in our university system demonstrates a classic recursive relationship. The COURSE entity relates to itself through prerequisites, where some courses serve as prerequisites for others. CS101 (Intro to Programming) is a prerequisite for CS201 (Advanced Programming), which, in turn, is a prerequisite for CS303 (Database Systems).

This recursive structure is used to ensure students have the necessary background knowledge before enrolling in advanced courses. It is important for degree planning, enrollment validation, and curriculum development, as it helps uphold the educational dependencies that define proper academic progression.

Recursive Relationship Roles

In recursive relationships, entities have different **roles** (Connolly & Begg, 2015). In the prerequisite relationship, one course serves as the "prerequisite," while the other acts as the "dependent." These role labels are essential for understanding the relationship's meaning and applying correct business logic.

Recursive Relationship Implementation

Recursive relationships demand special attention during database implementation. They establish **self-referencing foreign key structures**. This setup allows complex queries to traverse prerequisite chains and identify all courses that depend on a specific prerequisite. The translation of these ER constructs into database tables is explained in detail in Chapter 7: Relational Model.

4.8.4.2 Binary Relationships

Binary relationships represent associations between exactly two entity types, making them the most common and fundamental relationship pattern in ER modeling. They promote **conceptual clarity and efficient implementation**. Binary relationships directly translate into relational database structures, bridging the gap from conceptual design to physical implementation.

Binary relationships mirror the **natural associations** seen in real business activities. It comes from how we naturally think in terms of binary relationships: who works in which department, which student enrolled in which course, or which customer placed which order. Examples from our use cases include relationships between customers and orders, employees and departments, students and courses, and products and suppliers.

University System Binary Examples

Our university system demonstrates various binary relationship patterns. Each one supports different aspects of academic operations:

STUDENT enrolls_in COURSE: Students selecting and registering for academic courses (see Figure 4.12).

PROFESSOR teaches COURSE: Each professor is assigned to teach courses (see Figure 4.12).

STUDENT belongs_to FACULTY: Administrative structure: students are part of the existing faculties in the university (see Figure 4.14).

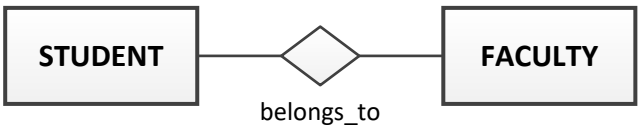


Figure 4.14: "belongs_to" Relationship

Company System Binary Examples

The Company system shows how binary relationships support business operations:

EMPLOYEE works_for DEPARTMENT: Organizational structure: each employee belongs to a department (see Figure 4.12).

EMPLOYEE processes ORDER: Order fulfillment and customer service operations (see Figure 4.12).

CUSTOMER places ORDER: Commercial transactions and sales processing (see Figure 4.15).

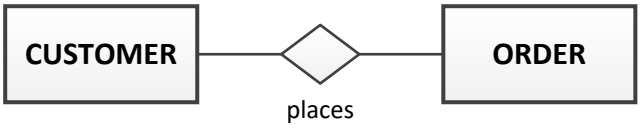


Figure 4.15: "places" Relationship

SUPPLIER supplies PRODUCT: Vendor relationships (see Figure 4.16).

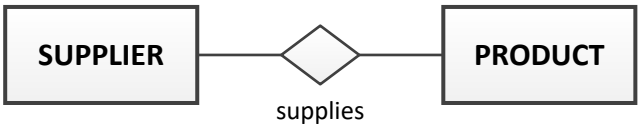


Figure 4.16: "supplies" Relationship

4.8.4.3 Ternary Relationships

Ternary relationships involve exactly three entity types that interact to create meaningful associations between them. These relationships capture complex business logic that cannot be adequately represented through binary relationships alone.

The example that demonstrates the necessity of a ternary relationship involves textbook assignments in our university case study. Consider the **"assigns" relationship** connecting

PROFESSOR, COURSE, and TEXTBOOK entities. This relationship captures the scenario in which professors select textbooks for the courses they teach.

This relationship **can't be decomposed** into binary relationships without losing important meaning. We can't just connect professors to textbooks (which professor uses which book?) or textbooks to courses (which book is used in which course?) without losing the details about who makes the assignment decisions and in what academic context.

Here is an example: Professor P001 (Kliment Ohridski) assigns Textbook ISBN-978-0134093413 (Introduction to Programming by Examples) for Course CS101 (Intro to Programming). This ternary relationship supports **textbook ordering, academic resource management, and course material coordination** in ways that binary relationships cannot adequately represent. Therefore, it reflects the academic reality that selecting course materials requires coordination between the professor's expertise, the course requirements, and available resources.

4.8.4.4 Higher-Degree Relationships

Higher-degree relationships involve four or more entities and represent complex business scenarios that are **rare in practice**. Most database scenarios that initially appear to require higher-degree relationships can be decomposed into combinations of binary and ternary relationships.

When higher-degree relationships occur, they usually involve complex approval processes or resource allocation scenarios with many participants. However, with detailed analysis, these scenarios can be modeled using **intermediate entities** that connect multiple binary relationships rather than single higher-degree relationships.

> **Design Guidelines for Relationship Degree**
>
> - **Start with binary relationships**: Most real relationships are naturally binary
> - **Consider ternary carefully**: Ensure all three entities are genuinely required for meaningful logic
> - **Avoid higher-degree**: Decompose complex scenarios into simpler relationship combinations
> - **Test decomposition**: Verify that binary alternatives don't lose essential meaning
> - **Consider implementation**: Higher-degree relationships create complex database structures

This systematic approach often shows that seemingly higher-degree relationships are better modeled with combinations of lower-degree ones, leading to clearer conceptual models and simpler database designs.

4.8.5 Cardinality of Relationships

Cardinality specifies the number of relationship instances in which an entity can participate (Elmasri & Navathe, 2016). It is important to understand cardinality because it directly affects database implementation.

When we specify that each employee works for exactly one department (1:N relationship), we're implementing an organizational procedure that prevents employees from being assigned to multiple departments simultaneously. These ensure that the system enforces business rules at the data level.

The notation for cardinality in Chen diagrams uses simple labels (1, M, N) placed on the lines connecting entities to relationships. These labels specify the maximum number of relationship instances in which entities can participate. We'll examine the relational implementation of these concepts in Chapter 7: Relational Model.

Cardinality Types Overview

- **One-to-One (1:1)**: Each entity instance relates to at most one instance on the other side
- **One-to-Many (1:N)**: "One" side entities can relate to multiple "many" side instances
- **Many-to-Many (M:N)**: Entity instances on both sides can relate to multiple instances on the other side

4.8.5.1 One-to-One (1:1) Relationships

One-to-one relationships are associations in which each entity instance on one side relates to at most one entity instance on the other side, and vice versa. These relationships are relatively rare because they often imply that the associated entities could be combined into a single entity.

One-to-one relationships typically occur in three scenarios: **security separation** (where sensitive information is stored separately from general information), **specialized subtype relationships** (where some entities have additional specialized attributes), or **legal/organizational requirements** (where certain information must be maintained separately for compliance reasons).

University System 1:1 Example

The "heads" relationship between PROFESSOR and FACULTY (see Figure 4.14) demonstrates a one-to-one (1:1) cardinality. Each faculty is led by exactly one head (the dean of that faculty), and each professor can be the head of at most one faculty. This one-to-one constraint clearly defines the leadership structure in the university.

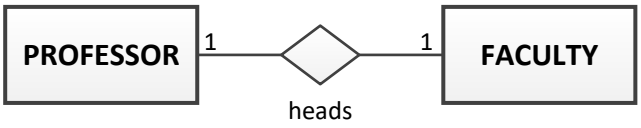


Figure 4.17: "heads" one-to-one (1:1) Relationship

Company System 1:1 Example

The **"manages" relationship** between EMPLOYEE and DEPARTMENT (see Figure 4.18) could represent one-to-one cardinality if company policy ensures that each department has exactly one manager and each manager can manage at most one department.

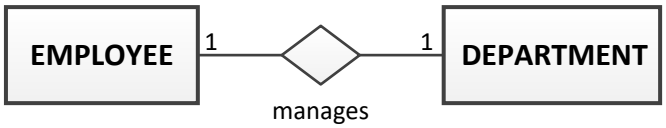


Figure 4.18: "manages" one-to-one (1:1) Relationship

4.8.5.2 One-to-Many (1:N) Relationships

One-to-many (1:N) relationships represent associations where entity instances on the "one" side can relate to multiple entity instances on the "many" side, but each entity instance on the "many" side relates to at most one entity instance on the "one" side.

One-to-many relationships represent natural associations in relational database scenarios, making them the **most common** type in database systems models. For example, departments have multiple employees, orders contain multiple line items, courses include multiple class sessions, and customers place multiple orders.

University System 1:N Examples

"offers" (FACULTY-COURSE) (see Figure 4.19): Each faculty can offer multiple courses, but each course is offered by exactly one faculty.

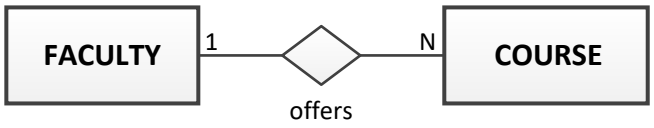


Figure 4.19: "offers" one-to-many (1:N) Relationship

"teaches" (PROFESSOR-COURSE) (see Figure 4.20): Each professor can teach multiple courses, but each course is taught by exactly one professor (as specified in the requirements). Please note that in some universities, a course can be taught by many professors. By analyzing the requirements, we specify the cardinality; in our case, a professor can teach one course.

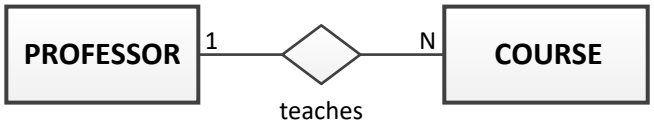


Figure 4.20: "teaches" one-to-many (1:N) Relationship

"conducts" (COURSE-CLASS_SESSION) (see Figure 4.21): Each course includes multiple class sessions, but each class session belongs to exactly one course.

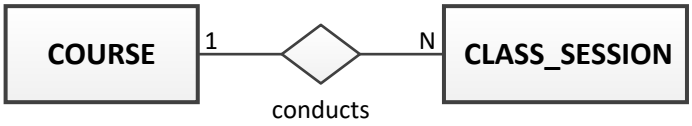


Figure 4.21: "conducts" one-to-many (1:N) Relationship

Company System 1:N Examples

"works_for" (DEPARTMENT-EMPLOYEE) (see Figure 4.22): Each department employs multiple workers, but each employee works for exactly one department.

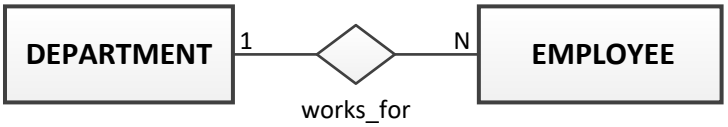


Figure 4.22: "works_for" one-to-many (1:N) Relationship

"places" (CUSTOMER-ORDER) (see Figure 4.23): Each customer can place multiple orders, but each order is placed by exactly one customer.

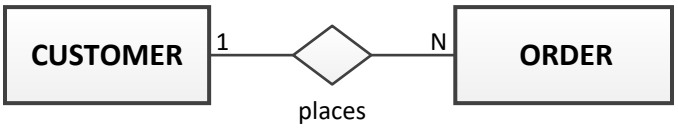


Figure 4.23: "places" one-to-many (1:N) Relationship

"processes" (EMPLOYEE-ORDER) (see Figure 4.24): Each employee can process multiple orders, but each order is processed by exactly one employee.

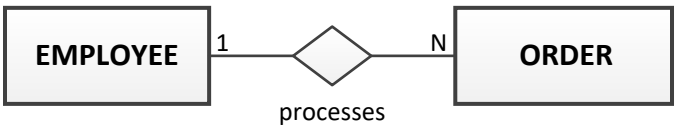


Figure 4.24: "processes" one-to-many (1:N) Relationship

4.8.5.3 Many-to-Many (M:N) Relationships

Many-to-many (M:N) relationships represent associations where entity instances on both sides can relate to multiple entity instances on the other side.

Many-to-many relationships often illustrate some of the most **interesting and complex business scenarios** in database systems. For example, students enroll in multiple courses, and each course has many students. Products are supplied by multiple suppliers, and each supplier supplies multiple products. Employees work on multiple projects, and projects involve multiple employees. These relationships reflect the collaborative and interconnected nature of modern relational database systems.

University System M:N Examples

"enrolls_in" (STUDENT-COURSE) (see Figure 4.25): Students can enroll in multiple courses, and courses can have multiple enrolled students. M:N relationships can include their own attributes as additional descriptive information. In our university case study, this includes **Grade** and **Enrollment_date** attributes, as explained in Relationship Attributes in subsection 4.8.2.

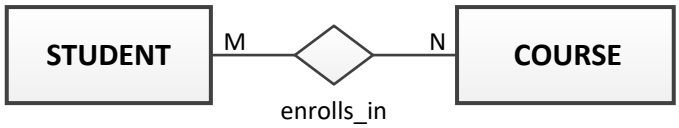


Figure 4.25: "enrolls_in" many-to-many (M:N) Relationship

"prerequisite" (COURSE-COURSE) (see Figure 4.26): Courses can have multiple prerequisites, and courses can serve as prerequisites for multiple other courses. This recursive many-to-many relationship creates the academic dependency network that ensures proper educational sequencing.

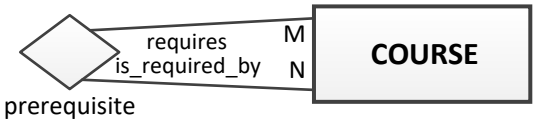


Figure 4.26: "prerequisite" many-to-many (M:N) Relationship

"attends" (STUDENT-CLASS_SESSION) (see Figure 4.27): Students can attend multiple class sessions, and class sessions can be attended by multiple students.

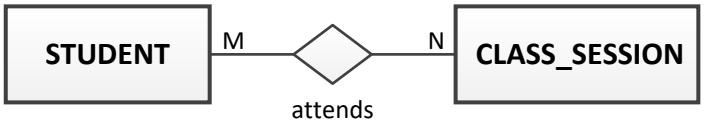


Figure 4.27: "attends" many-to-many (M:N) Relationship

Company System M:N Examples

"supplies" (SUPPLIER-PRODUCT) (see Figure 4.28): Suppliers can provide multiple products, and products can be supplied by multiple vendors. The relationship includes **Unit_cost** and **Supply_date** attributes for supplier tracking.

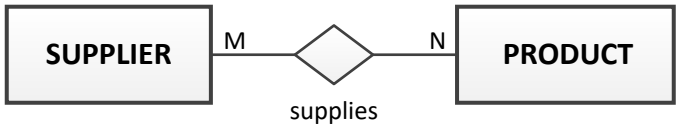


Figure 4.28: "supplies" many-to-many (M:N) Relationship

"stored_in" (PRODUCT-WAREHOUSE) (see Figure 4.29): Products can be stored in multiple warehouses, and warehouses can store multiple products. The relationship includes **Quantity_on_hand** and **Reorder_level** attributes for inventory management.

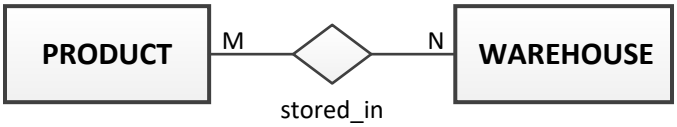


Figure 4.29: "stored_in" many-to-many (M:N) Relationship

4.8.5.4 Ternary Relationship Cardinality (M:N:K)

Ternary relationship cardinality involves three entities and specifies how each entity relates to combinations of the other two (Elmasri & Navathe, 2016). The notation **M:N:K** indicates that each entity can participate in multiple relationship instances, but specific combinations may be constrained by business rules.

In ternary relationships, constraints can apply to any combination of the three participating entities. Some ternary relationships allow unlimited participation by all entities (M:N:K with no additional constraints), while others include rules that limit specific combinations. For proper database implementation, it is important to understand these constraints.

> **Ternary Cardinality**
>
> **Ternary relationships require careful analysis** of constraints that apply to any three entities. The notation M:N:K represents the general case, but specific business rules often create functional dependencies that restrict certain combinations of entities.

University System Ternary Example

The **"assignment" relationship** connecting PROFESSOR, COURSE, and TEXTBOOK (see Figure 4.30) demonstrates a ternary (M:N:K) cardinality. A professor can assign multiple textbooks to multiple courses; textbooks can be assigned by multiple professors to multiple courses; and courses can have multiple textbooks assigned by multiple professors.

However, this flexibility can sometimes be limited. For example, each course might allow only one professor to assign textbooks, creating a restriction in which each (Course, Textbook) pair is linked to exactly one professor. This kind of restriction affects how the ternary relationship is implemented.

Cardinality Constraint Analysis

Analyzing ternary relationship cardinality requires understanding **which entity combinations uniquely identify the relationship** and which business rules constrain participation (Date, 2003). In the assignment example, if the requirements ensure that each course has one primary professor responsible for textbook selection, then the (Course, Textbook) combination uniquely determines the Professor, creating a functional dependency within the ternary relationship.

Consider these common constraint patterns in ternary relationships:

Type 1: No Constraints (Full M:N:K)

- All entities can participate freely with any combination of the other two entities
- Example: Authors can write multiple books for multiple publishers with no restrictions

Type 2: Pairwise Constraints

- Specific pairs of entities have functional dependencies
- Example: Each (Course, Textbook) combination determines exactly one Professor
- Notation: The constrained entity gets "1" while others get "M" or "N"

Type 3: Complex Business Rules

- Multiple constraints apply simultaneously
- Example: Each project uses one primary supplier per product, but can have backup suppliers
- Requires careful analysis of business requirements and constraint documentation

Alternative Modeling Considerations

For complex ternary constraints, we can consider **refining the model**. If the (Course, Textbook) combination uniquely determines the Professor, the relationship might be better modeled as:

- Binary relationship: PROFESSOR assigns TEXTBOOK
- Binary relationship: TEXTBOOK used_in COURSE
- Business rule: Assignment authority is course-specific

This requires careful analysis to ensure no business meaning is lost.

Graphical Representation

Figure 4.30 shows the ternary relationship with M:N:K cardinality, demonstrating how three entities connect through a single relationship diamond with proper cardinality labels on each connecting line. Additional constraints, like functional dependencies, are usually documented separately rather than shown graphically.

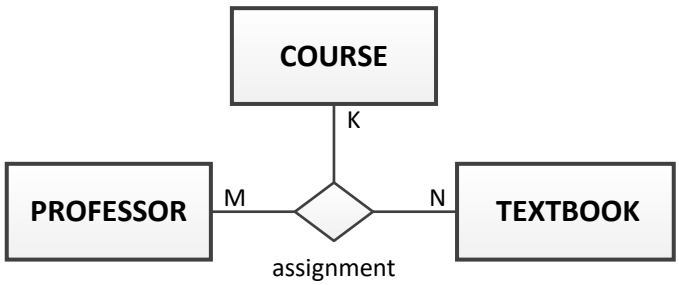


Figure 4.30: "assignment" ternary (M:N:K) Relationship

University System **(Figure 4.30):**

- **PROFESSOR** connects with "M" (can assign multiple textbook-course combinations)
- **COURSE** connects with "K" (can have multiple textbook-professor combinations)
- **TEXTBOOK** connects with "N" (can be assigned by multiple professor-course combinations)

Documentation additional note: "Each course-textbook combination is assigned by exactly one professor (the course professor responsible for material selection)."

Design Guidelines for Ternary Cardinality

When analyzing ternary relationship cardinality, follow these systematic steps:

1. **Identify Base Cardinality**: Determine the general M:N:K pattern based on the requirements
2. **Analyze Constraints**: Look for functional dependencies between entity pairs
3. **Document Business Rules**: Clearly specify any restrictions on entity combinations
4. **Consider Alternatives**: Evaluate whether binary relationships might be clearer
5. **Plan Implementation**: Choose an implementation strategy based on constraint complexity

Performance and Query Implications

Ternary relationships with complex cardinality constraints can make SQL query scenarios **more challenging**. Designers need to consider how applications will access the three-way data and whether the ternary structure supports efficient querying patterns. Proper analysis of ternary cardinality ensures these complex relationships accurately reflect requirements while remaining feasible and maintainable in database systems.

4.8.5.5 Cardinality Summary and Selection Guidelines

Cardinality patterns are essential for accurately modeling business relationships and meeting requirements. The choice of cardinality directly affects both the model's clarity and the final database implementation strategy.

Table 4.4 provides a summary of cardinality types, their characteristics, and business applications:

Table 4.4: Relationship Cardinality Summary

Cardinality Type	***Notation***	***Description***	***Examples***	***Implementation Notes***
One-to-One (1:1)	1:1	Each entity instance relates to at most one instance on the other side	Professor heads Faculty; Employee manages Department; Person has Social Security Number;	Relatively rare; Often suggests entity combination; Used for security separation;
One-to-Many (1:N)	1:N	"One" side entities can relate to multiple "many" side instances	Faculty offers Courses; Customer places Orders; Department employs Workers;	Most common cardinality; Represents hierarchical patterns; Natural organizational structures
Many-to-Many (M:N)	M:N	Entity instances on both sides can relate to	Students enroll in Courses;	Requires junction tables;

	multiple instances	Products supplied by Suppliers; Employees work on Projects;	Often has relationship attributes; Reflects collaborative processes;

4.8.6 Participation Constraints

Participation constraints concern whether relationships are mandatory or optional and specify whether entity instances must participate in relationship instances (Elmasri & Navathe, 2016).

There are two types of participation constraints: **total** and **partial**.

When we specify that every employee must belong to a department (total participation), we model a database that prevents unassigned employees. When we specify that not all professors are heads or deans of a faculty (partial participation), we recognize that most professors focus on teaching and research rather than administrative roles.

Participation Constraint Types

- **Total Participation (Existence Dependency)**: Every entity instance must participate in at least one relationship instance
- **Partial Participation**: Entity instances may or may not participate in relationship instances
- **Mixed Participation**: Different entities in the same relationship have different participation constraints

4.8.6.1 Total Participation (Existence Dependency)

Total participation means that every entity instance must participate in at least one relationship instance. This creates an **existence dependency**, in which entities cannot exist meaningfully without their required relationships (Date, 2003).

These constraints often arise from legal requirements, operational policies, or logical dependencies that make entity instances meaningless without their associated relationships.

Graphical Representation

Total participation is represented by **double lines** connecting the entity to the relationship.

University System Total Participation Examples

STUDENT "belongs_to" FACULTY (see Figure 4.31): Every student must belong to one faculty, in our example, Computer Science, Electrical Engineering, Economy & Tourism, Bioengineering, or Humanities. This total participation ensures that there are no unassigned students.

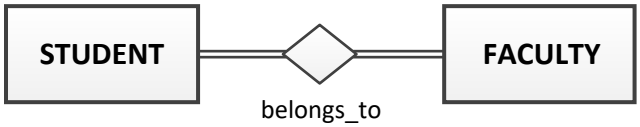


Figure 4.31: "belongs_to" relationship - total participation example

PROFESSOR "affiliated_with" FACULTY (see Figure 4.32): Every professor must be affiliated with a faculty. This constraint ensures that faculty members have appropriate institutional connections and administrative support.

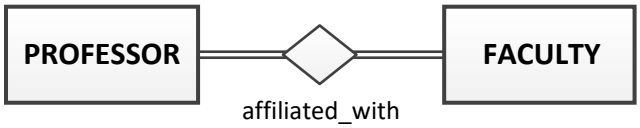


Figure 4.32: "affiliated_with" relationship - total participation example

CLASS_SESSION "conducts" COURSE (see Figure 4.33): Every class session must be associated with a specific course. Sessions exist only as components of course delivery.

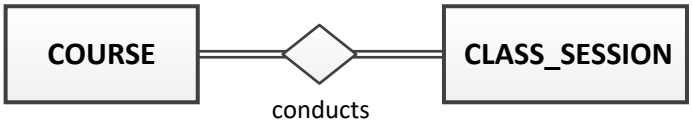


Figure 4.33: "conducts" relationship - total participation example

Company System Total Participation Examples

EMPLOYEE "works_for" DEPARTMENT (see Figure 4.34): Every employee must work for a department. This constraint prevents unassigned employees, as they would cause administrative issues and unclear responsibilities.

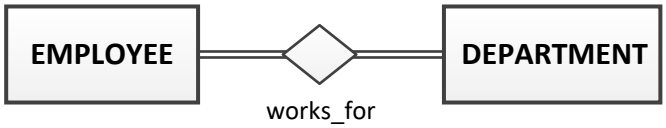


Figure 4.34: "works_for" relationship - total participation example

ORDER_LINE "contains" ORDER (see Figure 4.35): Every order line must belong to a specific order. Line items only exist as part of customer transactions. This total participation guarantees that all transaction details are properly associated with customer orders.

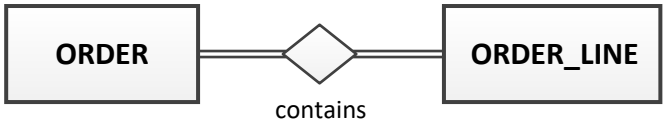


Figure 4.35: "contains" relationship - total participation example

Implications of Total Participation

Total participation constraints enforce essential business rules that uphold organizational integrity and operational efficiency.

- **Administrative Clarity**: Prevents orphaned records that don't belong to appropriate organizational units
- **Process Integrity**: Ensures that dependent entities maintain proper relationships to their parent entities
- **Regulatory Compliance**: Supports requirements where certain associations are legally mandated
- **System Consistency**: Eliminates ambiguous entity states that could cause processing errors

4.8.6.2 Partial Participation

Partial participation means that entity instances may participate in relationship instances or not. This indicates **optional associations** in which entities can exist independently of the relationship (Connolly & Begg, 2015).

Partial participation recognizes a reality in which not all entities require specific relationships to be meaningful.

Graphical Representation

Partial participation is represented by **single lines** connecting the entity to the relationship.

University System Partial Participation Examples

PROFESSOR "heads" FACULTY (see Figure 4.17): Not every professor acts as a dean of the faculty. Most professors focus on teaching and research instead of administrative leadership. This partial participation helps the system accurately represent both regular professors and administrative leaders.

COURSE "prerequisite" COURSE (see Figure 4.26): Not all courses have prerequisites. For example, introductory courses don’t require prerequisites because they provide foundational knowledge. This partial participation allows for an accurate representation of the curriculum structure while accommodating entry-level courses**.**

STUDENT "attends" CLASS_SESSION (see Figure 4.27): Students may not attend every class session due to illness or other legitimate reasons. This partial participation reflects both students' engagement and the reality of academic attendance.

Company System Partial Participation Examples

EMPLOYEE "manages" DEPARTMENT (see Figure 4.18): Not every employee has a management role. Most focus on operational responsibilities rather than management duties. This partial participation allows the system to represent both regular employees and managers.

PRODUCT "stored_in" WAREHOUSE (see Figure 4.29): Not all products are currently stored in warehouses. Some may be discontinued, in development, or temporarily out of stock. This partial participation allows product management while accommodating different product lifecycle stages and conditions.

4.8.6.3 Mixed Participation Scenarios

Many relationships involve **different participation constraints** for different participating entities. There are scenarios in which some entities may have mandatory relationships (total participation), while others have optional ones (partial participation).

University System Mixed Participation

The **"enrolls_in" relationship** between STUDENT and COURSE (see Figure 4.36) demonstrates mixed participation. Students have total participation because each student must be enrolled in at least one course to maintain an active academic status. Courses have partial participation because some courses might be offered but have no current enrollments (such as elective courses or new courses).

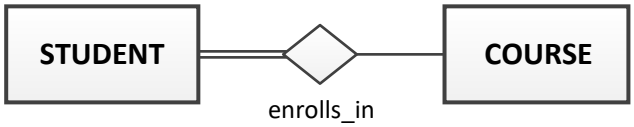


Figure 4.36: "enrolls_in" relationship - mixed participation example

Company System Mixed Participation

The "supplies" relationship between SUPPLIER and PRODUCT (see Figure 4.37) demonstrates mixed participation. Every product must have at least one supplier (total participation), while a supplier may currently supply no products (partial participation).

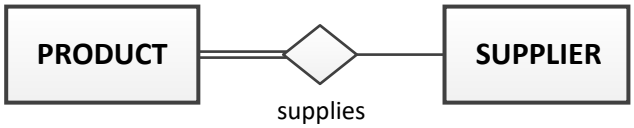


Figure 4.37: "supplies" relationship - mixed participation example

Note: Figures 4.31-4.37 depict the proper Chen notation, with double lines indicating total participation and single lines indicating partial participation, as explained in each example.

Total Participation vs Partial Participation

Choose Total Participation when

- Business rules mandate the relationship for all entity instances
- Entities become meaningless without the relationship
- Legal or regulatory requirements enforce the relationship
- Operational processes depend on the relationship existing

Choose Partial Participation when:

- The relationship represents optional roles or characteristics
- Business flexibility requires accommodating entities without the relationship
- Different entity lifecycle states need different relationship patterns
- Future business evolution might change relationship requirements

4.8.7 Weak Entities and Their Identifying Relationships

Weak entities rely on strong entities for both existence and identification, requiring special handling in related relationships (Elmasri & Navathe, 2016). The connections linking weak entities to their owner entities are known as **identifying relationships**.

Weak entity relationships represent **fundamental hierarchical dependencies** in business systems. For example, order line items cannot exist without orders, class sessions cannot exist without courses, and dependent family members cannot exist without employees.

Why Weak Entities Matter: Weak entities prevent orphaned data and ensure that dependent information cannot exist without its logical parent, maintaining **data consistency**. They reflect real-world scenarios in which certain objects have meaning only within the context of other objects.

> **Weak Entity Characteristics**
>
> - **Existence Dependency**: Cannot exist without its owner entity
> - **Identification Dependency**: Requires the owner's key plus the partial key for complete identification
> - **Total Participation**: Always have total participation in identifying relationships
> - **Cascading Operations**: Database operations on owners must cascade to weak entities

4.8.7.1 Characteristics of Identifying Relationships

Identifying relationships has several unique characteristics that differentiate them from regular relationships (Date, 2003):

Existence Dependency: Weak entities cannot exist without their owner entities. If a course is deleted, all its class sessions must be deleted as well. If an order is cancelled, all its order lines become meaningless.

Identification Dependency: Weak entities require their owner's key plus their own **partial key** for complete identification. Class Session "1" is meaningless without knowing which course it belongs to. The complete identification requires combining Course_ID, Session_number, and Year to create unique identifiers such as "CS101-1-2025" or "MA301-2-2025".

Total Participation: Weak entities **always have total participation** in their identifying relationships because they cannot exist without their owners. This is represented graphically by double lines connecting the weak entity to the identifying relationship.

Cascading Operations: Database operations on owner entities must cascade to weak entities to maintain consistency. When an ORDER is deleted, all its ORDER_LINE records must also be removed because they represent components of that specific transaction.

Composite Key Formation: The combination of the owner entity's primary key and the weak entity's partial key forms a **composite primary key** that uniquely identifies each weak entity instance across the entire database.

University System Identifying Relationships

"conducts" (COURSE-CLASS_SESSION) (see Figure 4.38): This identifying relationship connects the weak entity CLASS_SESSION to its owner COURSE. Each class session exists only as a component of a specific course and is identified by combining Course_ID, Session_number, and Year.

Class sessions are scheduled meetings within courses, so they are not standalone educational events. The session number only makes sense within the context of a specific course offering in a particular year. For illustration, when course CS101 is cancelled, all its associated class sessions (CS101-1-2025, CS101-2-2025, CS101-3-2025) must also be cancelled because they have no independent meaning or purpose.

Practical applications of this identifying relationship:

- **Course scheduling**: Sessions are scheduled as components of course offerings
- **Facility management**: Classroom assignments are made for course-session combinations
- **Attendance tracking**: Student attendance is recorded per course session
- **Academic calendar management**: Session timing aligns with course requirements

Company System Identifying Relationships

"contains" (ORDER-ORDER_LINE) (see Figure 4.39): This identifying relationship connects the weak entity ORDER_LINE to its owner ORDER. Each order line exists only as part of a specific customer order and is identified by combining Order_ID and Line_number.

Line items are not standalone business transactions; they are detailed components of customer orders. The line number only makes sense within the context of a specific customer order. For illustration, when Order 5 is cancelled, all its order lines (Order5-Line1, Order5-Line2, Order5-Line3) must also be cancelled because they represent components of that specific customer transaction.

Practical applications of this identifying relationship:

- **Order processing**: Line items are processed as components of customer orders
- **Inventory management**: Stock allocation occurs at the line-item level within orders
- **Transaction tracking**: Financial calculations aggregate line-item details
- **Customer service**: Order inquiries reference specific line items within orders

Graphical Representation of Identifying Relationships

Identifying relationships are represented by **double diamonds** connected to weak entities with double lines (Chen, 1976).

Figures 4.38 and 4.39 illustrate the Chen notation for identifying relationships, showing double diamonds connected to weak entities (double rectangles) with double lines indicating total participation and existence dependency.

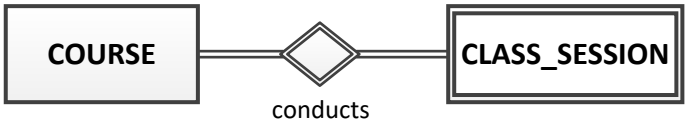


Figure 4.38: University System Identifying Relationship - "conducts" (COURSE-CLASS_SESSION)

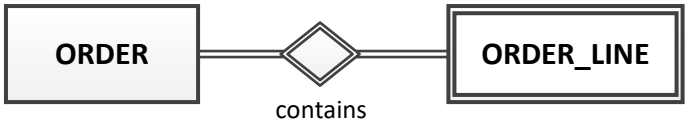


Figure 4.39: Company System Identifying Relationship - "contains" (ORDER-ORDER_LINE)

University System (Figure 4.38 explained):

- Single rectangle for strong entity COURSE
- Double diamond for identifying relationship "conducts"
- Double rectangle for weak entity CLASS_SESSION
- Double lines indicating total participation for both entities

Company System (Figure 4.39 explained):

- Single rectangle for strong entity ORDER
- Double diamond for identifying relationship "contains"
- Double rectangle for weak entity ORDER_LINE
- Double lines indicating total participation for both entities

4.8.8 Relationship Design Principles

The relationship design should **translate business processes and requirements into appropriate technical implementations**. It should reflect how actual institutions and organizations that are modeled operate. For example, when we model the relationship between students and courses, we are linking two entities or data tables. This mirrors the real educational process where knowledge is transferred, and academic progress is tracked. A thorough understanding of this process ensures that our models support both current operations and the organization's future growth.

4.8.8.1 Strategic Design Guidelines

Alignment with Business Rules

Relationships should accurately reflect the requirements. If business policy allows one employee to work in multiple departments, the relationship should be many-to-many. If policy requires an employee to work in only one department, the relationship should be many-to-one.

It's also important to consider not just current policies and practices but also the organization's evolution. For example, the relationship between PROFESSOR and COURSE might now be one-to-many (each course is taught by one professor), but future teaching initiatives could require many-to-many relationships (many professors can teach one course, and each course can be taught by many professors). Designing with this flexibility in mind helps prevent costly restructuring later.

Implementation Feasibility

Consider **how relationships translate into relational database structures** (Date, 2003). Many-to-many relationships require junction tables, which increase storage needs and query complexity. Ternary relationships need specialized implementation strategies that can impact query performance. Identifying relationships results in cascading operational requirements that must be carefully managed.

The implementation implications go beyond basic structure to include query patterns, indexing strategies, and maintenance procedures. A complex ternary relationship might offer conceptual clarity but can also create query challenges that affect system performance.

Future Flexibility

When designing relationships, consider whether current cardinality constraints are essential and accurate or simply temporary policy choices. Create a system that can adapt over time without requiring major restructuring. For example, current policy might allow employees to work in only one department (one-to-many relationship), but future organizational changes could require employees to work across multiple departments (many-to-many relationship).

Participation Constraints

Specify participation constraints that **accurately reflect business requirements** (Connolly & Begg, 2015). Avoid assuming total participation unless business rules genuinely require mandatory relationships. Don't use partial participation if organizational policy demands specific associations.

4.8.9 Common Relationship Design Mistakes

Design errors can lead to implementation problems during database development and deployment (Date, 2003). Learning from common relationship design mistakes can help us build stronger conceptual models and avoid costly redesigns. Many of these errors seem minor during conceptual modeling but cause cascading issues during implementation, testing, and operational deployment.

4.8.9.1 Structural Design Errors

Mistake 1: Redundant Relationships

Creating **multiple relationships that capture the same business association** without clear justification or different business rules. If customers place orders with line items, don't create a separate "purchases" relationship that directly connects customers to products

unless product-level customer tracking has genuinely different business rules from order-based transaction tracking.

Example Problem: Modeling both "places" (CUSTOMER-ORDER) and "purchases" (CUSTOMER-PRODUCT) when customer-product relationships are already captured through the order structure (CUSTOMER places ORDER, ORDER contains ORDER_LINE, ORDER_LINE references PRODUCT).

Solution Approach: Analyze whether seemingly different relationships actually represent the same business concept at different levels of granularity. If they do, model the more fundamental relationship and derive the other through proper hierarchical structures.

Mistake 2: Missing Intermediate Entities

Failing to recognize when **direct relationships actually involve intermediate objects**. The relationship between SUPPLIER and WAREHOUSE might seem binary, but it often requires an intermediate SHIPMENT entity to capture delivery schedules, transportation details, and shipment-specific tracking information.

Example Problem: Modeling "delivers_to" (SUPPLIER-WAREHOUSE) as a simple many-to-many relationship when business operations actually involve shipments with specific delivery dates, transportation methods, tracking numbers, and shipment-specific costs that don't belong to either the supplier or warehouse entities.

Solution Approach: When many-to-many relationships accumulate multiple attributes or complex business rules, consider whether they represent intermediate entities that deserve explicit modeling.

4.8.9.2 Cardinality and Constraint Errors

Mistake 3: Incorrect Cardinality Specification

Misspecifying cardinality constraints that don't reflect business reality. Claiming that each order has exactly one customer (1:1) when customers can place multiple orders (1:N) creates logical contradictions that prevent proper system operation.

Example Problem: Specifying "manages" (EMPLOYEE-DEPARTMENT) as 1:1 when business rules allow departments to have assistant managers or acting managers in addition to primary managers.

Solution Approach: Systematically validate cardinality decisions against real business scenarios and edge cases. Test each constraint against questions like "Can this entity participate in multiple relationship instances?" and "What happens during organizational transitions?"

Mistake 4: Inappropriate Participation Constraints

Specifying total participation when business rules allow partial relationships, or vice versa. This mismatch creates systems that either reject valid business scenarios or accept invalid data configurations.

Example Problem: Requiring total participation for PROFESSOR in "heads" relationships when most faculty focus on teaching and research rather than administrative leadership roles.

Solution Approach: Distinguish between current organizational patterns and absolute business requirements. Just because most professors don't serve as department heads doesn't mean the relationship should require total participation.

4.8.9.3 Complexity Management Errors

Mistake 5: Overuse of Ternary Relationships

Creating ternary relationships when binary relationships would be clearer and more appropriate for the business logic involved. Many apparent ternary relationships can be decomposed into binary relationships involving intermediate entities, which provide greater semantic clarity and simpler implementation.

Example Problem: Modeling PROFESSOR-COURSE-TEXTBOOK as a single ternary relationship when the business logic actually involves professors making textbook selections for their courses, which is better modeled as PROFESSOR assigns TEXTBOOK_SELECTION, and TEXTBOOK_SELECTION used_in COURSE.

Solution Approach: Before creating ternary relationships, systematically analyze whether decomposing the binary relationship into intermediate entities yields clearer semantics and easier implementation while preserving all necessary business constraints.

Mistake 6: Weak Entity Misidentification

Failing to properly model identifying relationships or incorrectly identifying strong entities as weak entities. Order line items that aren't properly connected to orders through identifying relationships can become orphaned data, complicating business operations and violating referential integrity.

Example Problem: Modeling CLASS_SESSION as a strong entity when sessions exist only as components of specific courses and have no independent business meaning.

Solution Approach: Apply the existence dependency test: "If the parent entity is deleted, should all related entities also be deleted?" If yes, consider modeling the weak entity with identifying relationships.

The goal of relationship design is to create models that naturally support business operations while maintaining data integrity and ensuring efficient implementation. This requires understanding not just technical modeling rules, but also the business context in which relationships operate and evolve over time.

Understanding relationships thoroughly is vital because they determine how information flows within organizational systems and how business rules are applied at the data level. Well-designed relationships lay the foundation for efficient database operations and dependable support for business processes.

4.9 Creating ER Diagrams from Scratch: A Step-by-Step Approach

In this section, we will demonstrate the complete process of creating Entity-Relationship diagrams from scratch using real-world examples. This practical approach will consolidate all the concepts explained in this chapter and provide you with a systematic methodology

for database design. We will use two case studies to illustrate this process: a University Management System and a Company Management System.

4.9.1 The Step-by-Step Methodology

Creating an ER diagram follows a systematic, sentence-by-sentence analysis that applies all the concepts we've covered in this chapter. As we analyze each requirement statement, we'll identify:

1. **Entities** (Section 4.6) - both strong and weak entities
2. **Attributes** (Section 4.7) - simple, composite, multivalued, derived, and key attributes
3. **Relationships** (Section 4.8) - with proper degree, cardinality, and participation constraints

Throughout this analysis, we use **Chen notation** (covered in Section 4.4).

4.9.2 Case Study 1: University Database System

Requirements Analysis

We need to design an ER diagram for a university database system with the following requirements:

There are students in the university. For each student, the system stores a student ID, birth date, email addresses, name, and address. A student can have multiple email addresses. The student's name consists of a first name and a last name. The student's address consists of a street, city, zip code, and country. The student's age can be calculated from their birth date.

The university offers courses. For each course, the system stores a course ID, title, description, and credit hours. Some courses have prerequisite courses. A course can have multiple prerequisites, and a course can serve as a prerequisite for multiple other courses.

Students enroll in courses. A student can enroll in multiple courses, and a course can have multiple enrolled students. For each enrollment, the system tracks the enrollment date and the student's course grade. Every student must be enrolled in at least one course.

There are professors in the university. For each professor, the system stores a professor ID, name, hire date, email address, office location, and academic title. The professor's name consists of a first name and a last name. The academic title includes positions such as assistant professor, associate professor, full professor, or lecturer.

Professors teach courses. A professor can teach multiple courses, but each course has exactly one main professor assigned to teach it. Every course must have a professor assigned to teach it.

There are faculties in the university. For each faculty, the system stores a faculty ID, name, budget, and location. Students belong to faculties. Each student belongs to exactly one faculty, and each faculty has multiple students. Every student must belong to a faculty, and every faculty must have students.

Professors are affiliated with faculties. Each professor is affiliated with exactly one faculty, and each faculty has multiple professors. Every professor must be affiliated with a faculty, and every faculty must have professors. One professor from each faculty serves as the faculty head (dean). Each faculty has exactly one professor as its head.

Courses are conducted through class sessions. Each course consists of multiple class sessions, and each class session belongs to exactly one course. Every course must have class sessions, and every class session must belong to a course. For each class session, the system stores a session number, room, semester, year, and schedule. The schedule consists of the day, time, and duration.

Students attend class sessions. A student can attend multiple class sessions, and a class session can be attended by multiple students. The system tracks attendance for each student in each class session.

Textbooks are used at the university. For each textbook, the system stores an ISBN, title, author, edition, publisher, and number of available copies. Professors assign textbooks to courses. A professor can assign multiple textbooks to a course, and a textbook can be assigned by multiple professors to different courses.

Faculties offer courses. Each faculty can offer multiple courses, and each course must be offered by exactly one faculty.

4.9.3 Sentence-by-Sentence Analysis

Now let's analyze each sentence systematically to identify the components of our ER diagram:

***Sentence 1*: "*There are students in the university*."**

Analysis: This identifies our first entity - STUDENT, representing individual students enrolled at the university.

Note: Since we are designing a database for a single university, we don't create a separate entity for the university itself. If we were designing for multiple universities, we would need a UNIVERSITY entity.

Identified Components:

- **Entity:** STUDENT (Strong Entity)

Graphical Representation:

Figure 4.40: First Entity Identification - STUDENT Entity

Figure 4.40 Description: Single rectangle labeled STUDENT representing a strong entity in Chen notation.

Sentences 2-6*: "*For each student, the system stores a student ID, birth date, email addresses, name, and address. A student can have multiple email addresses. The

***student's name consists of a first name and a last name. The student's address consists of a street, city, zip code, and country. The student's age can be calculated from their birth date*."**

Analysis: These sentences collectively describe all attributes of the STUDENT entity, covering key, simple, multivalued, composite, and derived attributes.

Identified Components:

- **Key attribute**: Student_ID
- **Simple attribute**: Birth_date
- **Multivalued attribute**: Email (multiple email addresses)
- **Composite attribute**: Name (First_name, Last_name)
- **Composite attribute**: Address (Street, City, Zip_code, Country)
- **Derived attribute**: Age (calculated from Birth_date)

Graphical Representation:

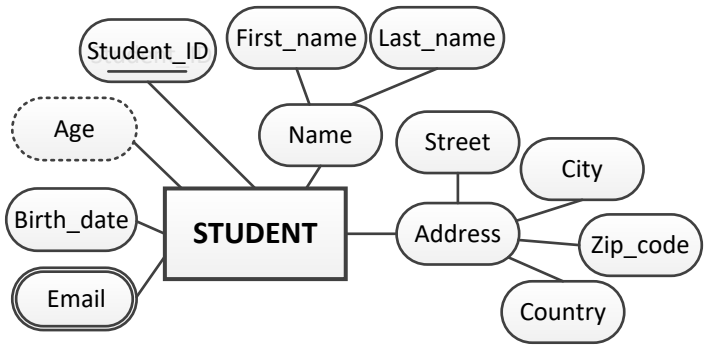


Figure 4.41: STUDENT Entity with Complete Attribute Structure

Figure 4.41 Description: STUDENT entity with complete attribute structure - Student_ID underlined as key attribute, Birth_date as simple attribute, Email as double oval for multivalued, Name and Address as hierarchical composite attributes, and Age as dashed oval for derived attribute.

Sentence 7: **"*The university offers courses*."**

Analysis: This identifies our second entity - COURSE.

Identified Components:

- **Entity: COURSE (Strong Entity)**

Graphical Representation:

Figure 4.42: Second Entity Identification - COURSE Entity

Figure 4.42 Description: COURSE entity shown as a single rectangle.

Sentence 8: **"*For each course, the system stores a course ID, title, description, and credit hours*."**

Analysis: This describes the attributes of the COURSE entity.

Identified Components:

- **Key attribute**: Course_ID
- **Simple attributes**: Title, Description, Credit_hours

Graphical Representation

Figure 4.43 Description: COURSE entity with four attributes connected - Course_ID underlined as key attribute, and three simple attributes as single ovals.

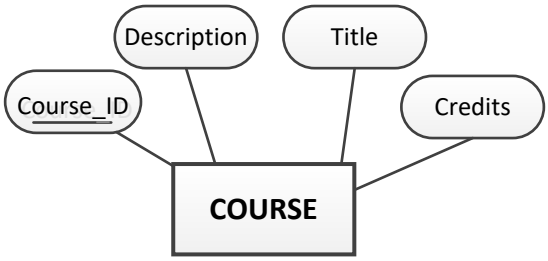


Figure 4.43: COURSE Entity with Complete Attributes

***Sentences 9-10*: "*Some courses have prerequisite courses. A course can have multiple prerequisites, and a course can serve as a prerequisite for multiple other courses*."**

Analysis: These sentences establish a recursive many-to-many relationship within the COURSE entity.

Identified Components:

- **Recursive relationship**: prerequisite (COURSE to COURSE)
- **Cardinality**: M:N

Graphical Representation

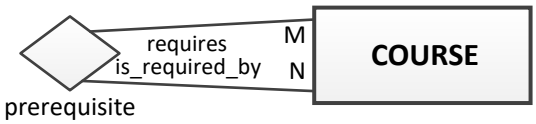


Figure 4.44: Recursive Many-to-Many Prerequisite Relationship

Figure 4.44 Description: COURSE entity with recursive relationship diamond labeled "prerequisite" connected back to itself with curved lines, showing M:N cardinality with role labels.

***Sentences 11-13*: "*Students enroll in courses. A student can enroll in multiple courses, and a course can have multiple enrolled students. For each enrollment, the system tracks the enrollment date and the student's grade in the course*."**

Analysis: These sentences establish the relationship between entities, specify its cardinality as many-to-many, and identify attributes that belong to the relationship itself.

Identified Components:

- **Relationship**: enrolls_in (between STUDENT and COURSE)
- **Cardinality**: M:N (many-to-many)
- **Relationship attributes**: Enrollment_date, Grade

Graphical Representation

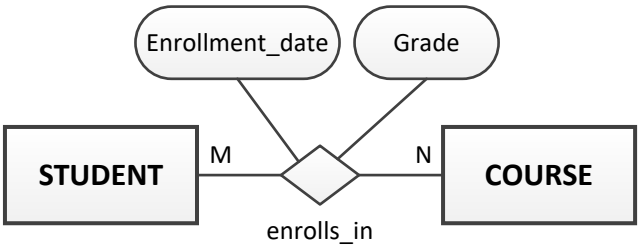


Figure 4.45: Many-to-Many Enrollment Relationship with Attributes

Figure 4.45 Description: STUDENT and COURSE entities connected by a diamond-shaped relationship labeled "enrolls_in" with M:N cardinality notation and two oval attributes (Enrollment_date and Grade) connected to the relationship diamond.

***Sentence 14*: "*Every student must be enrolled in at least one course*."**

Analysis: This specifies total participation for STUDENT in the enrolls_in relationship.

Identified Components:

- **Participation constraint**: STUDENT has total participation in enrolls_in

Graphical Representation

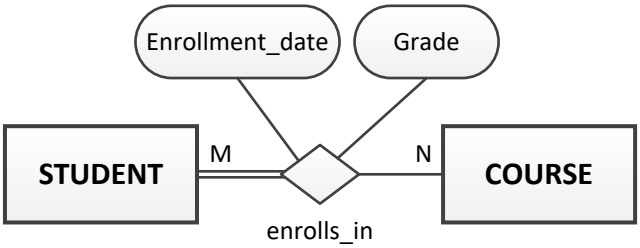


Figure 4.46: Total Participation - Students Must Enroll in Courses

Figure 4.46 Description: Double line connecting the STUDENT entity to the enrolls_in relationship, indicating total participation.

***Sentence 15*: "*There are professors in the university*."**

Analysis: This identifies our third entity - PROFESSOR.

Identified Components:

- **Entity**: PROFESSOR (Strong Entity)

Graphical Representation

Figure 4.47: Third Entity Identification - PROFESSOR Entity

Figure 4.47 Description: PROFESSOR entity shown as a single rectangle.

***Sentences 16-18*: "*For each professor, the system stores a professor ID, name, hire date, email address, office location, and academic title. The professor's name consists of a first name and a last name. The academic title includes positions such as assistant professor, associate professor, full professor, or lecturer*."**

Analysis: These sentences describe all attributes of the PROFESSOR entity, including a composite attribute and domain constraints.

Identified Components:

- **Key attribute**: Professor_ID
- **Composite attribute**: Name (First_name, Last_name)
- **Simple attributes**: Hire_date, Email, Office, Academic_title
- **Domain constraint**: Academic_title has specific values (assistant professor, associate professor, full professor, or lecturer)

Graphical Representation

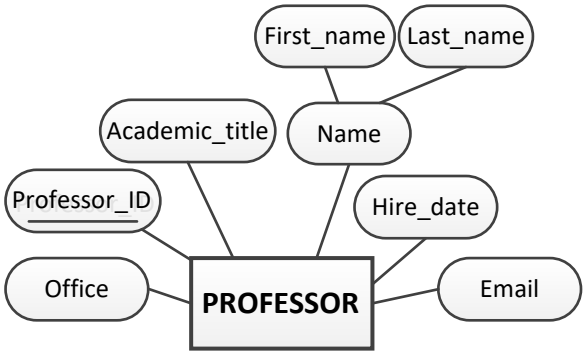


Figure 4.48: PROFESSOR Entity with Complete Attribute Structure

Figure 4.48 Description: PROFESSOR entity with complete attribute structure - Professor_ID underlined as key, hierarchical Name composite attribute, and simple attributes as single ovals.

***Sentences 19-21*: "Professors teach courses. A professor can teach multiple courses, but each course has exactly one main professor assigned to teach it. Every course must have a professor assigned to teach it."**

Analysis: These sentences establish a one-to-many relationship between PROFESSOR and COURSE with total participation for COURSE.

Identified Components:

- **Relationship**: teaches (between PROFESSOR and COURSE)
- **Cardinality**: 1:N (one professor to many courses)
- **Participation constraint**: COURSE has total participation in “teaches” relationship

Graphical Representation

Figure 4.49 Description: PROFESSOR and COURSE entities connected by "teaches" relationship diamond with 1:N cardinality notation and double line from COURSE indicating total participation.

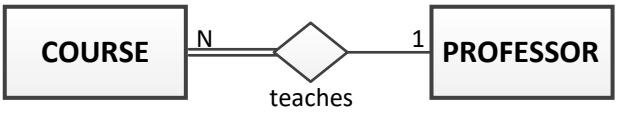


Figure 4.49: One-to-Many Teaching Relationship with Total Participation

***Sentence 22*: "There are faculties in the university."**

Analysis: This identifies our fourth entity - FACULTY.

Identified Components:

- **Entity**: FACULTY (Strong Entity)

Graphical Representation

Figure 4.50: Fourth Entity Identification - FACULTY Entity

Figure 4.50 Description: FACULTY entity shown as a single rectangle.

***Sentence 23*: "For each faculty, the system stores a faculty ID, name, budget, and location."**

Analysis: This describes the attributes of the FACULTY entity.

Identified Components:

- **Key attribute**: Faculty_ID
- **Simple attributes**: Name, Budget, Location

Graphical Representation

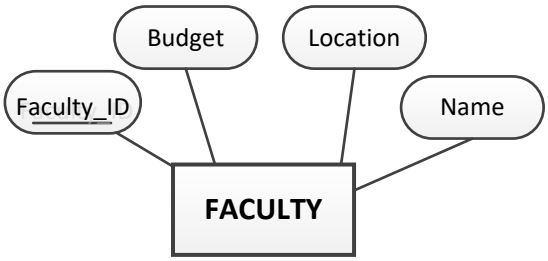


Figure 4.51: FACULTY Entity with Complete Attributes

Figure 4.51 Description: FACULTY entity with four attributes connected - Faculty_ID underlined as key attribute, and three simple attributes as single ovals.

***Sentences 24-26*: "Students belong to faculties. Each student belongs to exactly one faculty, and each faculty has multiple students. Every student must belong to a faculty, and every faculty must have students."**

Analysis: These sentences establish a one-to-many relationship between FACULTY and STUDENT with total participation for both entities.

Identified Components:

- **Relationship**: belongs_to (between STUDENT and FACULTY)
- **Cardinality**: 1:N (one faculty to many students)
- **Participation constraint**: Total participation for both STUDENT and FACULTY

Graphical Representation

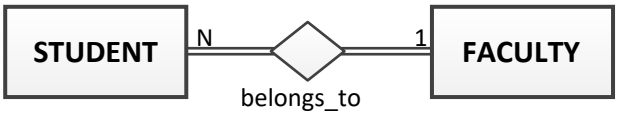


Figure 4.52: One-to-Many Student-Faculty Relationship with Total Participation

Figure 4.52 Description: FACULTY and STUDENT entities connected by "belongs_to" relationship diamond with 1:N cardinality notation and double lines from both entities indicating total participation.

***Sentences 27-29*: "Professors are affiliated with faculties. Each professor is affiliated with exactly one faculty, and each faculty has multiple professors. Every professor must be affiliated with a faculty, and every faculty must have professors."**

Analysis: These sentences establish a one-to-many relationship between FACULTY and PROFESSOR with total participation for both entities.

Identified Components:

- **Relationship**: affiliated_with (between PROFESSOR and FACULTY)
- **Cardinality**: 1:N (one faculty to many professors)
- **Participation constraint**: Total participation for both PROFESSOR and FACULTY

Graphical Representation

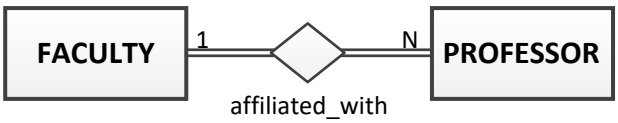


Figure 4.53: One-to-Many Professor-Faculty Affiliation with Total Participation

Figure 4.53 Description: FACULTY and PROFESSOR entities connected by "affiliated_with" relationship diamond with 1:N cardinality notation and double lines from both entities indicating total participation.

***Sentences 30-31*: "One professor from each faculty serves as the faculty head (dean). Each faculty has exactly one professor as its head."**

Analysis: These sentences establish a one-to-one relationship between PROFESSOR and FACULTY for the leadership role.

Identified Components:

- **Relationship**: heads (between PROFESSOR and FACULTY)
- **Cardinality**: 1:1 (one-to-one)
- **Participation constraint**: FACULTY has total participation (every faculty must have a head)

Graphical Representation

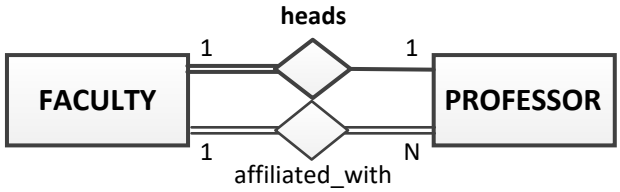


Figure 4.54: One-to-One Head/Dean Relationship

Figure 4.54 Description: PROFESSOR and FACULTY entities connected by a "heads" relationship diamond with 1:1 cardinality notation and a double line from FACULTY indicating total participation.

***Sentences 32-34*: "Courses are conducted through class sessions. Each course consists of multiple class sessions, and each class session belongs to exactly one course. Every course must have class sessions, and every class session must belong to a course."**

Analysis: These sentences introduce CLASS_SESSION as a weak entity that depends on COURSE for existence and identification, establishing an identifying relationship.

Identified Components:

- **Weak Entity**: CLASS_SESSION (depends on COURSE)
- **Identifying relationship**: conducts (between COURSE and CLASS_SESSION)
- **Cardinality**: 1:N
- **Participation constraint**: Total participation for both entities

Graphical Representation

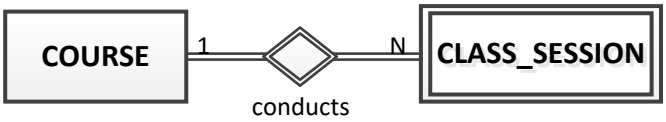


Figure 4.55: Weak Entity and Identifying Relationship - Course Conducts Class Sessions

Figure 4.55 Description: COURSE (single rectangle) connected to CLASS_SESSION (double rectangle) through "conducts" identifying relationship (double diamond) with 1:N cardinality and double lines from both entities.

***Sentences 35-36*: "For each class session, the system stores a session number, room, semester, year, and schedule. The schedule consists of the day, time, and duration."**

Analysis: These sentences describe the attributes of the CLASS_SESSION weak entity, including the partial key and the composite attribute.

Identified Components:

- **Partial key**: Session_number
- **Simple attributes**: Room, Semester, Year
- **Composite attribute**: Schedule (Day, Time, Duration)

Graphical Representation

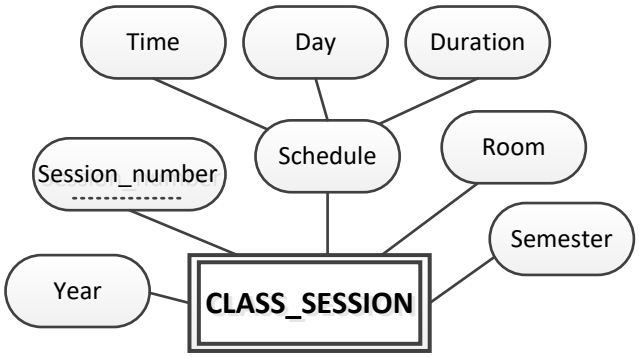


Figure 4.56: CLASS_SESSION Weak Entity with Complete Attribute Structure

Figure 4.56 Description: CLASS_SESSION weak entity with attributes - Session_number with dashed underline (partial key), simple attributes as single ovals, and Schedule as a hierarchical composite attribute.

***Sentences 37-39*: "Students attend class sessions. A student can attend multiple class sessions, and a class session can be attended by multiple students. The system tracks attendance for each student in each class session."**

Analysis: These sentences establish a many-to-many relationship between STUDENT and CLASS_SESSION with a relationship attribute.

Identified Components:

- **Relationship**: attends (between STUDENT and CLASS_SESSION)
- **Cardinality**: M:N (many-to-many)

- **Relationship attribute**: Attendance

Graphical Representation

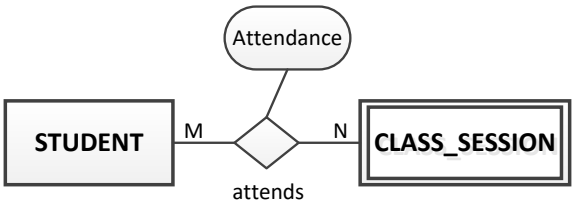


Figure 4.57: Many-to-Many Student-Class Session Attendance Relationship

Figure 4.57 Description: STUDENT and CLASS_SESSION entities connected by "attends" relationship diamond with M:N cardinality notation and Attendance attribute connected as an oval.

***Sentence 40*: "Textbooks are used at the university."**

Analysis: This identifies our fifth entity - TEXTBOOK.

Identified Components:

- **Entity**: TEXTBOOK (Strong Entity)

Graphical Representation

Figure 4.58: Fifth Entity Identification - TEXTBOOK Entity

Figure 4.58 Description: TEXTBOOK entity shown as a single rectangle.

***Sentence 41*: "For each textbook, the system stores an ISBN, title, author, edition, publisher, and number of available copies."**

Analysis: This describes the attributes of the TEXTBOOK entity.

Identified Components:

- **Key attribute**: ISBN
- **Simple attributes**: Title, Author, Edition, Publisher, Available_copies

Graphical Representation

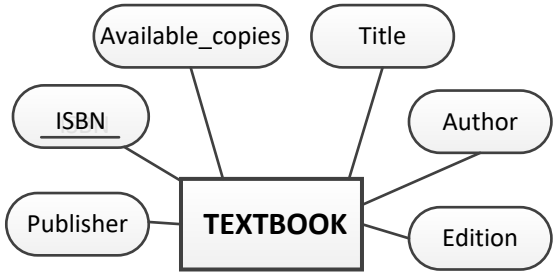


Figure 4.59: TEXTBOOK Entity with Complete Attributes

Figure 4.59 Description: TEXTBOOK entity with six attributes connected - ISBN underlined as key attribute, and five simple attributes as single ovals.

***Sentences 42-43*: "Professors assign textbooks to courses. A professor can assign multiple textbooks to a course, and a textbook can be assigned by multiple professors to different courses."**

Analysis: These sentences establish a ternary relationship between PROFESSOR, TEXTBOOK, and COURSE with many-to-many-to-many cardinality.

Identified Components:

- **Ternary relationship**: assigns (between PROFESSOR, TEXTBOOK, and COURSE)
- **Cardinality**: M:N:K (many-to-many-to-many)

Graphical Representation

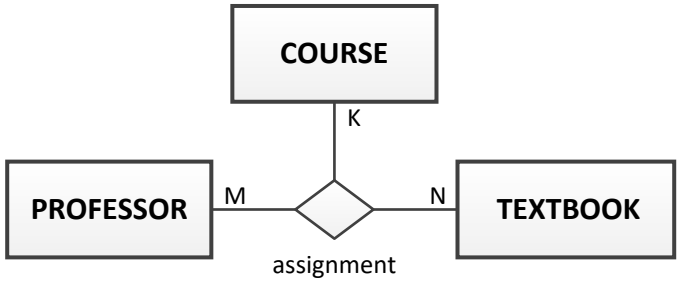


Figure 4.60: Ternary Assignment Relationship with M:N:K Cardinality

Figure 4.60 Description: Three entities (PROFESSOR, TEXTBOOK, COURSE) connected to a single relationship diamond labeled "assigns" with M, N, K cardinality labels on the connecting lines.

***Sentences 44-45*: "Faculties offer courses. Each faculty can offer multiple courses, and each course must be offered by exactly one faculty."**

Analysis: These sentences establish a one-to-many relationship between FACULTY and COURSE, with total participation for COURSE.

Identified Components:

- **Relationship**: offers (between FACULTY and COURSE)
- **Cardinality**: 1:N (one faculty to many courses)
- **Participation constraint**: COURSE has total participation in offers

Graphical Representation

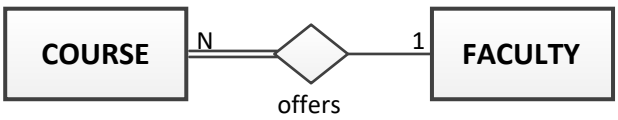


Figure 4.61: One-to-Many Faculty Course Offerings with Total Participation

Figure 4.61 Description: FACULTY and COURSE entities connected by "offers" relationship diamond with 1:N cardinality notation and double line from COURSE indicating total participation.

Summary of University Database Components

Entities Identified:

1. **STUDENT** (Strong Entity)
2. **COURSE** (Strong Entity)
3. **PROFESSOR** (Strong Entity)
4. **FACULTY** (Strong Entity)
5. **TEXTBOOK** (Strong Entity)
6. **CLASS_SESSION** (Weak Entity - depends on COURSE)

Attributes Summary:

STUDENT:

- Student_ID (Key, Simple)
- Name (Composite: First_name, Last_name)
- Birth_date (Simple)
- Age (Derived)
- Email (Multivalued)
- Address (Composite: Street, City, Zip_code, Country)

COURSE:

- Course_ID (Key, Simple)
- Title (Simple)
- Description (Simple)
- Credits (Simple)

PROFESSOR:

- Professor_ID (Key, Simple)
- Name (Composite: First_name, Last_name)
- Hire_date (Simple)
- Email (Simple)
- Office (Simple)
- Academic_title (Simple)

FACULTY:

- Faculty_ID (Key, Simple)

- Name (Simple)
- Budget (Simple)
- Location (Simple)

TEXTBOOK:

- ISBN (Key, Simple)
- Title (Simple)
- Author (Simple)
- Edition (Simple)
- Publisher (Simple)
- Available_copies (Simple)

CLASS_SESSION:

- Session_number (Partial Key)
- Room (Simple)
- Semester (Simple)
- Year (Simple)
- Schedule (Composite: Day, Time, Duration)

Relationships and Cardinalities:

1. **enrolls_in** (STUDENT M:N COURSE)
    - Attributes: Enrollment_date, Grade
    - Participation: Total for STUDENT, Partial for COURSE
2. **prerequisite** (COURSE M:N COURSE) - Recursive
    - Participation: Partial for both roles
3. **teaches** (PROFESSOR 1:N COURSE)
    - Participation: Partial for PROFESSOR, Total for COURSE
4. **belongs_to** (STUDENT N:1 FACULTY)
    - Participation: Total for both entities
5. **affiliated_with** (PROFESSOR N:1 FACULTY)
    - Participation: Total for both entities
6. **heads** (PROFESSOR 1:1 FACULTY)
    - Participation: Partial for PROFESSOR, Total for FACULTY
7. **conducts** (COURSE 1:N CLASS_SESSION) - Identifying relationship
    - Participation: Total for both entities
8. **attends** (STUDENT M:N CLASS_SESSION)

- Attributes: Attendance
- Participation: Partial for both entities

9. **assignment** (PROFESSOR M:N:K COURSE, TEXTBOOK) - Ternary
    - Participation: Partial for all entities
10. **offers** (FACULTY 1:N COURSE)
    - Participation: Total for COURSE, Partial for FACULTY

4.9.4 Constructing the Complete ER Diagram

Now that we have systematically identified all the components through our sentence-by-sentence analysis, we can construct the complete ER diagram by following a structured approach:

Step 1: Requirements Analysis and Business Rule Identification

- Carefully read and parse each sentence
- Identify business rules and constraints
- Look for temporal indicators, ownership patterns, and dependencies

Step 2: Entity Identification

- Look for **nouns** that represent objects of interest
- Apply the four key questions from Section 4.6.5:
    - Is it important to the system?
    - Does it exist independently?
    - Do we need multiple instances?
    - Can instances be uniquely identified?

Step 3: Draw Entities and Their Attributes

Begin by placing all identified entities (both strong and weak) on the diagram using appropriate notation - single rectangles for strong entities and double rectangles for weak entities. Then add all attributes to each entity using the correct symbols: single ovals for simple attributes, double ovals for multivalued attributes, dashed ovals for derived attributes, and composite attribute trees for complex attributes. Key attributes should be underlined, and partial keys in weak entities should have dashed underlining.

Step 4: Add Relationships and Cardinalities

Connect entities through relationship diamonds, ensuring that identifying relationships (for weak entities) use double diamonds. Label each relationship appropriately and add cardinality constraints (1, N, M) on the connecting lines. Pay special attention to recursive relationships like the prerequisite relationship within the COURSE entity.

Step 5: Specify Participation Constraints

Review each relationship to determine participation constraints. Use double lines to indicate total participation (mandatory) and single lines for partial participation (optional). This step ensures that business rules about required versus optional relationships are properly captured.

Step 6: Add Relationship Attributes

For relationships that have their own attributes (such as Grade in the enrolls_in relationship), connect these attributes directly to the relationship diamond using ovals.

Step 7: Validate and Refine

Review the complete diagram to ensure all requirements have been captured and that the diagram accurately reflects the business rules identified in the analysis:

- Test the model against business scenarios
- Verify that all requirements are captured
- Check for missing entities or relationships
- Ensure cardinality constraints match business rules

The complete ER diagram resulting from this systematic construction process is shown in Figure 4.62 and integrates all the components identified through our detailed requirements analysis.

Figure 4.62 Description: Complete ER diagram showing all six entities (STUDENT, COURSE, PROFESSOR, FACULTY, TEXTBOOK, CLASS_SESSION), their complete attribute structures, all relationships with proper cardinality and participation constraints, using Chen notation with appropriate symbols for weak entities, composite attributes, multivalued attributes, derived attributes, and identifying relationships.

This systematic analysis demonstrates how careful examination of requirements leads to a complete and accurate ER diagram. Each sentence contributes specific components that together form a comprehensive data model for the university system.

Next, we'll apply this same methodology to our Company Management System to reinforce these concepts in a different business context.

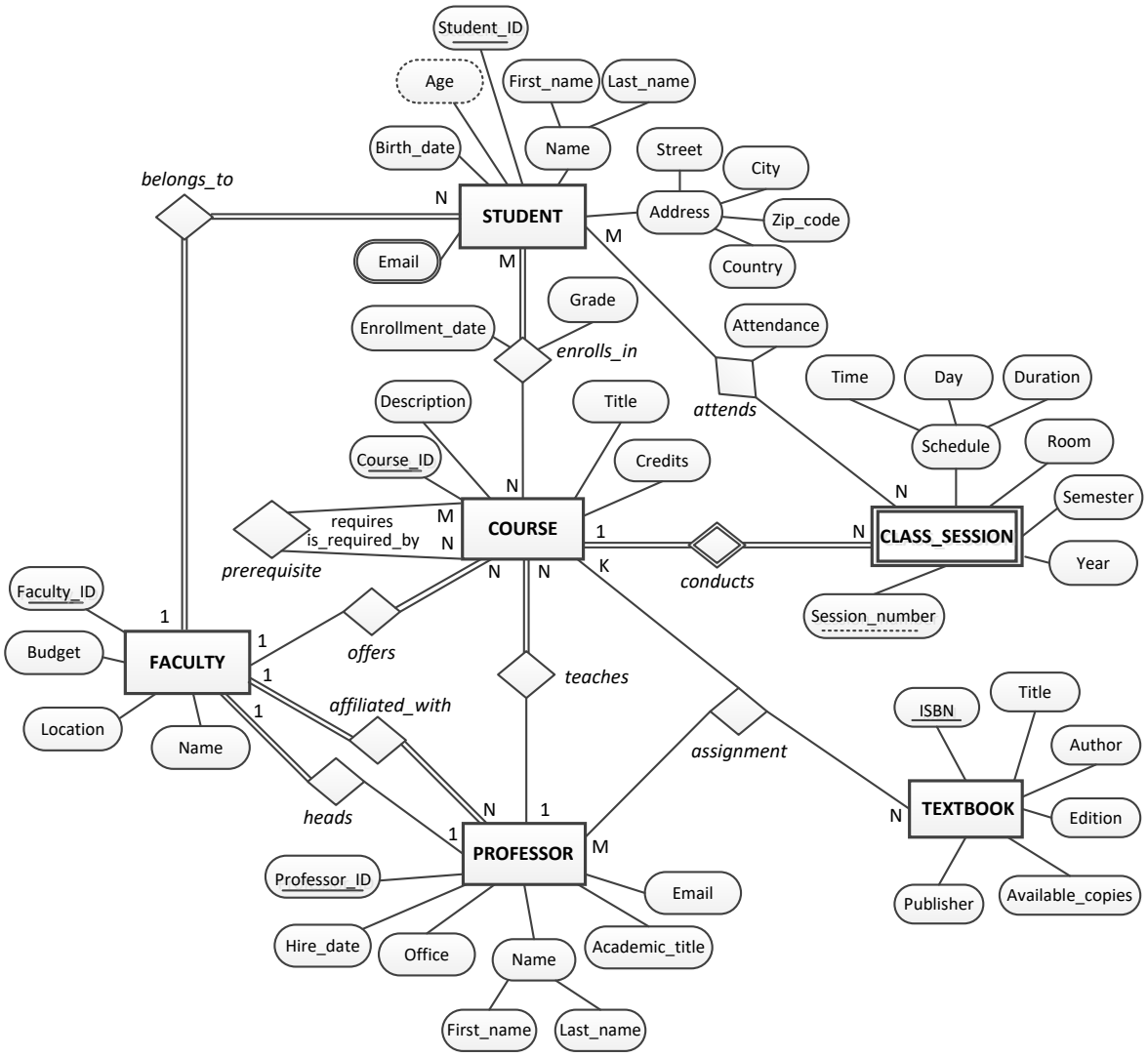


Figure 4.62: Complete University Database ER Diagram - Integrating All Components from Requirements Analysis

4.9.5 Case Study 2: Company Database System

Requirements Analysis

We need to design an ER diagram for a Company database system with the following requirements:

The company has employees. For each employee, the system stores an employee ID, email, salary, hire date, phone numbers, name, and address. An employee can have multiple phone numbers. The employee's name consists of a first name and a last name. The employee's address consists of a street, city, zip code, and country.

The company has departments. For each department, the system stores a department ID, department name, location, and budget. Employees work for departments. Each employee must work for exactly one department, and each department has multiple employees. Every employee must work for a department, and every department must have employees. An employee manages a department. Each department is managed by exactly one employee, but not every employee manages a department.

The company has warehouses. For each warehouse, the system stores a warehouse ID, location, and capacity. Departments oversee warehouses. Each department can oversee multiple warehouses, and each warehouse is overseen by exactly one department. Every warehouse must be overseen by a department. Employees work in warehouses. An employee can work at one warehouse, and each warehouse has multiple employees.

The company has orders. For each order, the system stores an order ID, status, order date, required date, and shipped date. The order date is when the customer placed the order; the required date is when the customer needs the order delivered; and the shipped date is when the order was actually sent to the customer. The total amount can be calculated from the sum of all order line amounts. Employees process orders. An employee can process multiple orders, and each order must be processed by exactly one employee. Every order must be processed by an employee.

The company has customers. For each customer, the system stores a customer ID, join date, email, phone numbers, name, and address. A customer can have multiple phone numbers. The customer's name consists of a first name and a last name. The customer's address consists of a street, city, zip code, and country. Customers place orders. A customer can place multiple orders, and each order must be placed by exactly one customer. Every order must be placed by a customer.

Orders contain order lines. Each order contains multiple order lines, and each order line belongs to exactly one order. Order lines cannot exist without orders. Every order must contain order lines, and every order line must belong to an order. For each order line, the system stores a line number, unit price, quantity, and discount.

The company has products. For each product, the system stores a product ID, product name, description, category, price, and weight. Products appear in order lines. Each product can appear in multiple order lines, and each order line references exactly one product. Every order line must reference a product.

Products are stored in warehouses. A product can be stored in multiple warehouses, and each warehouse can store multiple products. For each product stored in a warehouse, the system tracks the quantity on hand and reorder level. The quantity on hand represents the current stock level of the product in that specific warehouse, while the reorder level is the minimum stock level that triggers a reorder request.

The company has suppliers. For each supplier, the system stores a supplier ID, supplier name, email address, phone numbers, contact person's name, and address. A supplier can have multiple phone numbers. The contact person's name consists of a first name and a last name. The supplier's address consists of a street, city, zip code, and country. Suppliers supply products. A supplier can supply multiple products, and each product can be supplied by multiple suppliers. Every product must be supplied by at least one supplier. When a supplier supplies a product, the system stores the supply date and unit cost.

4.9.6 Sentence-by-Sentence Analysis

Now let's analyze each sentence systematically to identify the components of our ER diagram:

Sentence 1: **"*The company has employees*."**

Analysis: This identifies our first entity - EMPLOYEE, representing individual employees working for the company.

Note: Since we are designing a database for a single company, we don't create a separate entity for the company itself. If we were designing for multiple companies, we would need a COMPANY entity.

Identified Components:

- **Entity:** EMPLOYEE (Strong Entity)

Graphical Representation:

Figure 4.63: First Entity Identification - EMPLOYEE Entity

Figure 4.63 Description: Single rectangle labeled EMPLOYEE representing a strong entity in Chen notation.

Sentences 2-5: **"*For each employee, the system stores an employee ID, email, salary, hire date, phone numbers, name, and address. An employee can have multiple phone numbers. The employee's name consists of a first name and a last name. The employee's address consists of a street, city, zip code, and country*."**

Analysis: These sentences collectively describe all attributes of the EMPLOYEE entity, covering key, simple, multivalued, and composite attribute types.

Identified Components:

- **Key attribute**: Employee_ID
- **Simple attributes**: Email, Salary, Hire_date
- **Multivalued attribute**: Phone (multiple phone numbers)
- **Composite attribute**: Name (First_name, Last_name)
- **Composite attribute**: Address (Street, City, Zip_code, Country)

Graphical Representation:

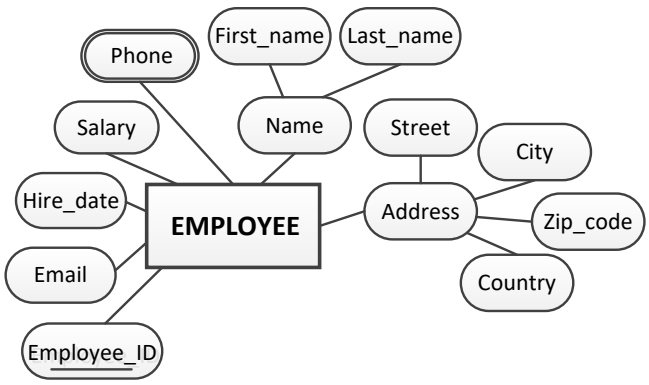


Figure 4.64: EMPLOYEE Entity with Complete Attribute Structure

Figure 4.64 Description: EMPLOYEE entity with complete attribute structure - Employee_ID underlined as key attribute, Email, Salary, and Hire_date as simple attributes, Phone as double oval for multivalued, and Name and Address as hierarchical composite attributes.

Sentence 6: **"*The company has departments*."**

Analysis: This identifies our second entity - DEPARTMENT.

Identified Components:

- **Entity:** DEPARTMENT (Strong Entity)

Graphical Representation:

Figure 4.65: Second Entity Identification - DEPARTMENT Entity

Figure 4.65 Description: DEPARTMENT entity shown as a single rectangle.

Sentence 7: **"*For each department, the system stores a department ID, department name, location, and budget*."**

Analysis: This describes the attributes of the DEPARTMENT entity.

Identified Components:

- **Key attribute**: Department_ID
- **Simple attributes**: Dept_name, Location, Budget

Graphical Representation:

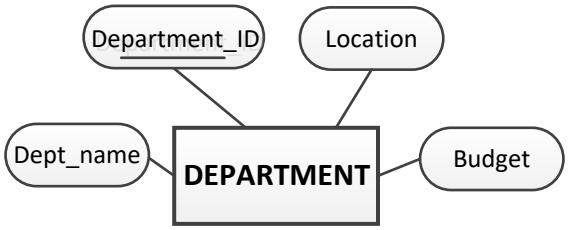


Figure 4.66: DEPARTMENT Entity with Complete Attributes

Figure 4.66 Description: DEPARTMENT entity with four attributes connected - Department_ID underlined as key attribute, and three simple attributes as single ovals.

Sentences 8-10: **"*Employees work for departments. Each employee must work for exactly one department, and each department has multiple employees. Every employee must work for a department, and every department must have employees*."**

Analysis: These sentences establish a one-to-many relationship between DEPARTMENT and EMPLOYEE with total participation for both entities.

Identified Components:

- **Relationship**: works_for (between EMPLOYEE and DEPARTMENT)
- **Cardinality**: 1:N (one department has many employees)

- **Participation constraint**: Total participation for both EMPLOYEE and DEPARTMENT

Graphical Representation:

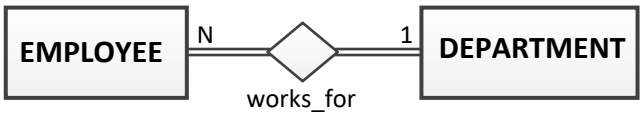


Figure 4.67: One-to-Many Employee-Department Relationship with Total Participation

Figure 4.67 Description: DEPARTMENT and EMPLOYEE entities connected by "works_for" relationship diamond with 1:N cardinality notation and double lines from both entities indicating total participation.

***Sentences 11-12*: "*An employee manages a department. Each department is managed by exactly one employee, but not every employee manages a department*."**

Analysis: These sentences establish a one-to-one relationship between EMPLOYEE and DEPARTMENT for management roles, with total participation only for DEPARTMENT.

Identified Components:

- **Relationship**: manages (between EMPLOYEE and DEPARTMENT)
- **Cardinality**: 1:1 (one-to-one)
- **Participation constraint**: DEPARTMENT has total participation, EMPLOYEE has partial participation

Graphical Representation:

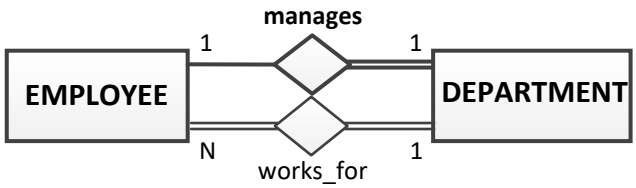


Figure 4.68: One-to-One Management Relationship

Figure 4.68 Description: EMPLOYEE and DEPARTMENT entities connected by "manages" relationship diamond with 1:1 cardinality notation and double line from DEPARTMENT indicating total participation.

***Sentence 13*: "*The company has warehouses*."**

Analysis: This identifies our third entity - WAREHOUSE.

Identified Components:

- **Entity:** WAREHOUSE (Strong Entity)

Graphical Representation:

Figure 4.69: Third Entity Identification - WAREHOUSE Entity

Figure 4.69 Description: WAREHOUSE entity shown as a single rectangle.

***Sentence 14*: "*For each warehouse, the system stores a warehouse ID, location, and capacity*."**

Analysis: This describes the attributes of the WAREHOUSE entity.

Identified Components:

- **Key attribute**: Warehouse_ID
- **Simple attributes**: Location, Capacity

Graphical Representation:

Figure 4.70 Description: WAREHOUSE entity with three attributes connected - Warehouse_ID underlined as key attribute, and two simple attributes as single ovals.

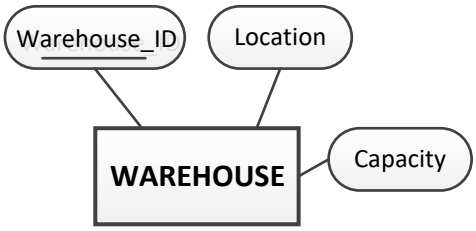


Figure 4.70: WAREHOUSE Entity with Complete Attributes

***Sentences 15-17*: "*Departments oversee warehouses. Each department can oversee multiple warehouses, and each warehouse is overseen by exactly one department. Every warehouse must be overseen by a department*."**

Analysis: These sentences establish a one-to-many relationship between DEPARTMENT and WAREHOUSE, with total participation for WAREHOUSE.

Identified Components:

- **Relationship**: oversees (between DEPARTMENT and WAREHOUSE)
- **Cardinality**: 1:N (one department oversees many warehouses)
- **Participation constraint**: WAREHOUSE has total participation

Graphical Representation:

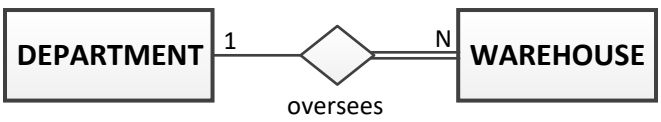


Figure 4.71: One-to-Many Department-Warehouse Oversight Relationship

Figure 4.71 Description: DEPARTMENT and WAREHOUSE entities connected by an "oversees" relationship diamond with 1:N cardinality notation and a double line from WAREHOUSE indicating total participation.

***Sentences 18-19*: "*Employees work at warehouses. An employee can work at one warehouse, and each warehouse has multiple employees*."**

Analysis: These sentences establish a many-to-one relationship between EMPLOYEE and WAREHOUSE with partial participation.

Identified Components:

- **Relationship**: works_at (between EMPLOYEE and WAREHOUSE)
- **Cardinality**: N:1 (many employees work at one warehouse)
- **Participation constraint**: Partial participation for both entities

Graphical Representation:

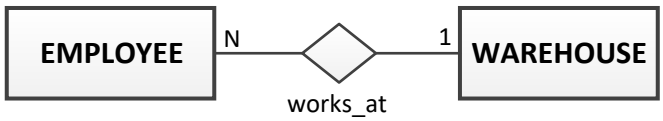


Figure 4.72: Many-to-One Employee-Warehouse Work Relationship

Figure 4.72 Description: EMPLOYEE and WAREHOUSE entities connected by "works_at" relationship diamond with N:1 cardinality notation and single lines indicating partial participation.

***Sentence 20*: "*The company has orders*."**

Analysis: This identifies our fourth entity - ORDER.

Identified Components:

- **Entity:** ORDER (Strong Entity)

Graphical Representation:

Figure 4.73: Fourth Entity Identification - ORDER Entity

Figure 4.73 Description: ORDER entity shown as a single rectangle.

***Sentences 21-22*: "*For each order, the system stores an order ID, status, order date, required date, and shipped date. The order date is when the customer placed the order; the required date is when the customer needs the order delivered; and the shipped date is when the order was actually sent to the customer. The total amount can be calculated from the sum of all order line amounts*."**

Analysis: These sentences describe the attributes of the ORDER entity, including a derived attribute.

Identified Components:

- **Key attribute**: Order_ID
- **Simple attributes**: Status, Order_date, Required_date, Shipped_date
- **Derived attribute**: Total_amount (calculated from order line amounts)

Graphical Representation:

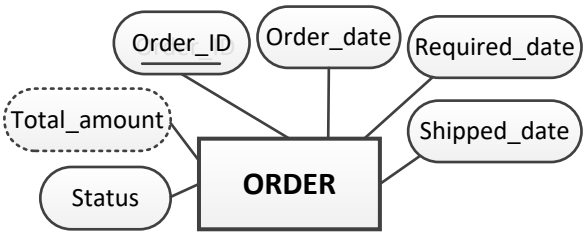


Figure 4.74: ORDER Entity with Attributes Including Derived Total Amount

Figure 4.74 Description: ORDER entity with attributes - Order_ID underlined as key, simple attributes as single ovals, and Total_amount as dashed oval for derived attribute.

***Sentences 23-25*: "*Employees process orders. An employee can process multiple orders, and each order must be processed by exactly one employee. Every order must be processed by an employee*."**

Analysis: These sentences establish a one-to-many relationship between EMPLOYEE and ORDER with total participation for ORDER.

Identified Components:

- **Relationship**: processes (between EMPLOYEE and ORDER)
- **Cardinality**: 1:N (one employee processes many orders)
- **Participation constraint**: ORDER has total participation

Graphical Representation:

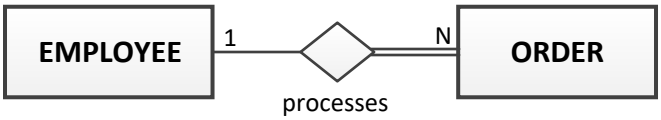


Figure 4.75: One-to-Many Employee-Order Processing Relationship

Figure 4.75 Description: EMPLOYEE and ORDER entities connected by "processes" relationship diamond with 1:N cardinality notation and double line from ORDER indicating total participation.

***Sentence 26*: "*The company has customers*."**

Analysis: This identifies our fifth entity - CUSTOMER.

Identified Components:

- **Entity:** CUSTOMER (Strong Entity)

Graphical Representation:

Figure 4.76: Fifth Entity Identification - CUSTOMER Entity

Figure 4.76 Description: CUSTOMER entity shown as a single rectangle.

***Sentences 27-30*: "*For each customer, the system stores a customer ID, join date, email, phone numbers, name, and address. A customer can have multiple phone numbers. The customer's name consists of a first name and a last name. The customer's address consists of a street, city, zip code, and country*."**

Analysis: These sentences describe all attributes of the CUSTOMER entity, including multivalued and composite attributes.

Identified Components:

- **Key attribute**: Customer_ID
- **Simple attributes**: Join_date, Email
- **Multivalued attribute**: Phone (multiple phone numbers)
- **Composite attribute**: Name (First_name, Last_name)
- **Composite attribute**: Address (Street, City, Zip_code, Country)

Graphical Representation:

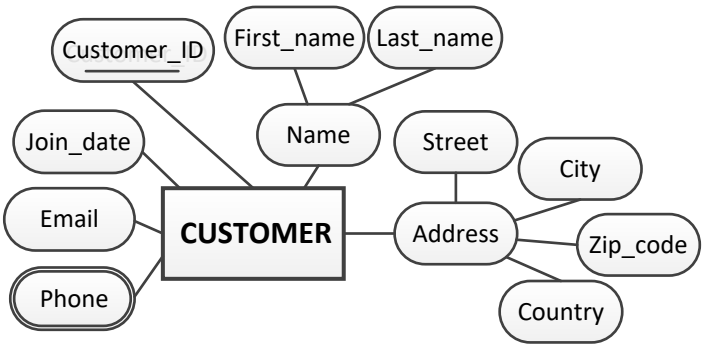


Figure 4.77: CUSTOMER Entity with Complete Attribute Structure

Figure 4.77 Description: CUSTOMER entity with complete attribute structure - Customer_ID underlined as key, Join_date and Email as simple attributes, Phone as double oval for multivalued, and Name and Address as hierarchical composite attributes.

***Sentences 31-33*: "*Customers place orders. A customer can place multiple orders, and each order must be placed by exactly one customer. Every order must be placed by a customer*."**

Analysis: These sentences establish a one-to-many relationship between CUSTOMER and ORDER with total participation for ORDER.

Identified Components:

- **Relationship**: places (between CUSTOMER and ORDER)
- **Cardinality**: 1:N (one customer places many orders)

- **Participation constraint**: ORDER has total participation

Graphical Representation:

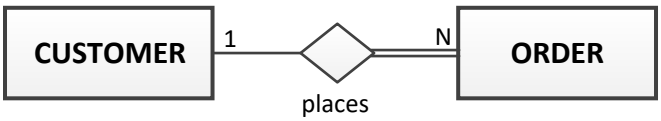


Figure 4.78: One-to-Many Customer-Order Placement Relationship

Figure 4.78 Description: CUSTOMER and ORDER entities connected by "places" relationship diamond with 1:N cardinality notation and double line from ORDER indicating total participation.

***Sentences 34-37*: "*Orders contain order lines. Each order contains multiple order lines, and each order line belongs to exactly one order. Order lines cannot exist without orders. Every order must contain order lines, and every order line must belong to an order*."**

Analysis: These sentences introduce ORDER_LINE as a weak entity that depends on ORDER for existence and identification, establishing an identifying relationship.

Identified Components:

- **Weak Entity**: ORDER_LINE (depends on ORDER)
- **Identifying relationship**: contains (between ORDER and ORDER_LINE)
- **Cardinality**: 1:N
- **Participation constraint**: Total participation for both entities

Graphical Representation:

Figure 4.79: Weak Entity and Identifying Relationship - Order Contains Order Lines

Figure 4.79 Description: ORDER (single rectangle) connected to ORDER_LINE (double rectangle) through "contains" identifying relationship (double diamond) with 1:N cardinality and double lines from both entities.

***Sentence 38*: "*For each order line, the system stores a line number, unit price, quantity, and discount*."**

Analysis: This describes the attributes of the ORDER_LINE weak entity.

Identified Components:

- **Partial key**: Line_number
- **Simple attributes**: Unit_price, Quantity, Discount

Graphical Representation:

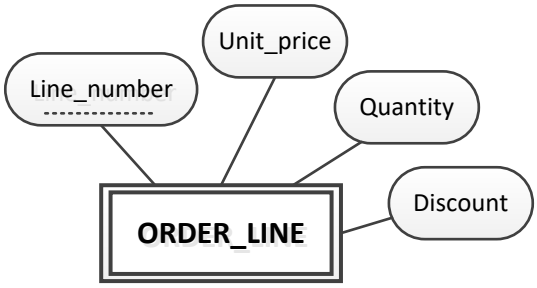


Figure 4.80: ORDER_LINE Weak Entity with Attributes

Figure 4.80 Description: ORDER_LINE weak entity with attributes - Line_number with dashed underline (partial key) and three simple attributes as single ovals.

Sentence 39: "***The company has products***."

Analysis: This identifies our sixth entity - PRODUCT.

Identified Components:

- **Entity:** PRODUCT (Strong Entity)

Graphical Representation:

Figure 4.81: Sixth Entity Identification - PRODUCT Entity

Figure 4.81 Description: PRODUCT entity shown as a single rectangle.

Sentence 40: "***For each product, the system stores a product ID, product name, description, category, price, and weight***."

Analysis: This describes the attributes of the PRODUCT entity.

Identified Components:

- **Key attribute**: Product_ID
- **Simple attributes**: Product_name, Description, Category, Price, Weight

Graphical Representation:

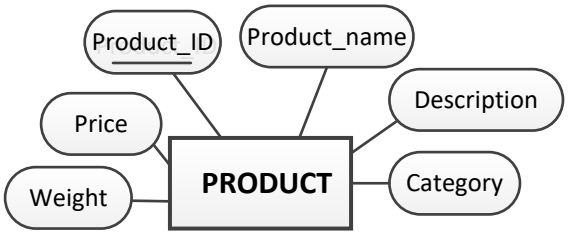


Figure 4.82: PRODUCT Entity with Complete Attributes

Figure 4.82 Description: PRODUCT entity with six attributes connected - Product_ID underlined as key attribute, and five simple attributes as single ovals.

Sentences 41-43: **"*Products appear in order lines. Each product can appear in multiple order lines, and each order line references exactly one product. Every order line must reference a product*."**

Analysis: These sentences establish a one-to-many relationship between PRODUCT and ORDER_LINE with total participation for ORDER_LINE.

Identified Components:

- **Relationship**: appears_in (between PRODUCT and ORDER_LINE)
- **Cardinality**: 1:N (one product appears in many order lines)
- **Participation constraint**: ORDER_LINE has total participation

Graphical Representation:

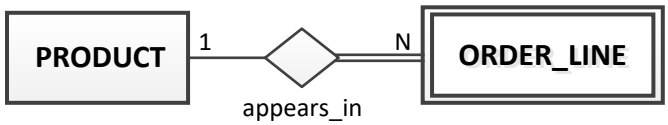


Figure 4.83: One-to-Many Product-Order Line Relationship

Figure 4.83 Description: PRODUCT and ORDER_LINE entities connected by "appears_in" relationship diamond with 1:N cardinality notation and double line from ORDER_LINE indicating total participation.

Sentences 44-46: **"*Products are stored in warehouses. A product can be stored in multiple warehouses, and each warehouse can store multiple products. For each product stored in a warehouse, the system tracks the quantity on hand and reorder level. The quantity on hand represents the current stock level of the product in that specific warehouse, while the reorder level is the minimum stock level that triggers a reorder request*."**

Analysis: These sentences establish a many-to-many relationship between PRODUCT and WAREHOUSE with relationship attributes for inventory management.

Identified Components:

- **Relationship**: stored_in (between PRODUCT and WAREHOUSE)
- **Cardinality**: M:N (many-to-many)
- **Relationship attributes**: Quantity_on_hand, Reorder_level
- **Participation constraint**: Partial participation for both entities

Graphical Representation:

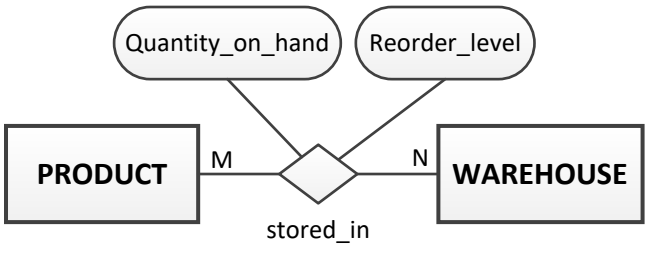


Figure 4.84: Many-to-Many Product-Warehouse Storage Relationship with Inventory Attributes

Figure 4.84 Description: PRODUCT and WAREHOUSE entities connected by "stored_in" relationship diamond with M:N cardinality notation and two oval attributes (Quantity_on_hand, Reorder_level) connected to the relationship diamond.

***Sentence 47*: "*The company has suppliers*."**

Analysis: This identifies our seventh entity - SUPPLIER.

Identified Components:

- **Entity:** SUPPLIER (Strong Entity)

Graphical Representation:

Figure 4.85: Seventh Entity Identification - SUPPLIER Entity

Figure 4.85 Description: SUPPLIER entity shown as a single rectangle.

***Sentences 48-51*: "*For each supplier, the system stores a supplier ID, supplier name, email addresses, phone numbers, contact person name, and address. A supplier can have multiple phone numbers. The contact person's name consists of a first name and a last name. The supplier's address consists of a street, city, zip code, and country*."**

Analysis: These sentences describe all attributes of the SUPPLIER entity, including multivalued and composite attributes.

Identified Components:

- **Key attribute**: Supplier_ID
- **Simple attributes**: Supplier_name, Email
- **Multivalued attribute**: Phone (multiple phone numbers)
- **Composite attribute**: Contact_person (First_name, Last_name)
- **Composite attribute**: Address (Street, City, Zip_code, Country)

Graphical Representation:

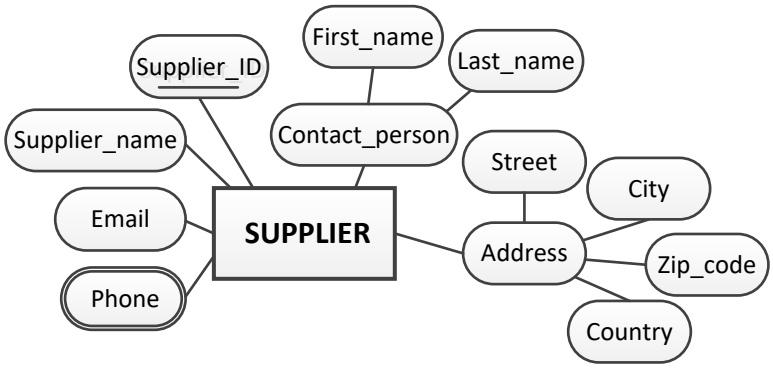


Figure 4.86: SUPPLIER Entity with Complete Attribute Structure

Figure 4.86 Description: SUPPLIER entity with complete attribute structure - Supplier_ID underlined as key, Supplier_name and Email as simple attributes, Phone as double oval for multivalued, and Contact_name and Address as hierarchical composite attributes.

***Sentences 52-55*: "Suppliers supply products. A supplier can supply multiple products, and each product can be supplied by multiple suppliers. Every product must be supplied by at least one supplier. When a supplier supplies a product, the system stores the supply date and unit cost."**

Analysis: These sentences establish a many-to-many relationship between SUPPLIER and PRODUCT with relationship attributes and total participation for PRODUCT.

Identified Components:

- **Relationship**: supplies (between SUPPLIER and PRODUCT)
- **Cardinality**: M:N (many-to-many)
- **Relationship attributes**: Supply_date, Unit_cost
- **Participation constraint**: PRODUCT has total participation

Graphical Representation:

Figure 4.87 Description: SUPPLIER and PRODUCT entities connected by "supplies" relationship diamond with M:N cardinality notation, double line from PRODUCT indicating total participation, and two oval attributes (Supply_date, Unit_cost) connected to the relationship diamond.

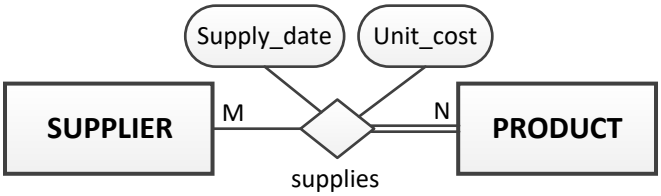


Figure 4.87: Many-to-Many Supplier-Product Supply Relationship with Supply Attributes

4.9.7 Summary of Company Database Components

Entities Identified:

1. **EMPLOYEE** (Strong Entity)
2. **DEPARTMENT** (Strong Entity)
3. **WAREHOUSE** (Strong Entity)
4. **ORDER** (Strong Entity)
5. **CUSTOMER** (Strong Entity)
6. **PRODUCT** (Strong Entity)
7. **SUPPLIER** (Strong Entity)

8. **ORDER_LINE** (Weak Entity - depends on ORDER)

Attributes Summary:

EMPLOYEE:

- Employee_ID (Key, Simple)
- Name (Composite: First_name, Last_name)
- Email (Simple)
- Salary (Simple)
- Hire_date (Simple)
- Phone (Multivalued)
- Address (Composite: Street, City, Zip_code, Country)

DEPARTMENT:

- Department_ID (Key, Simple)
- Dept_name (Simple)
- Location (Simple)
- Budget (Simple)

WAREHOUSE:

- Warehouse_ID (Key, Simple)
- Location (Simple)
- Capacity (Simple)

ORDER:

- Order_ID (Key, Simple)
- Status (Simple)
- Order_date (Simple)
- Required_date (Simple)
- Shipped_date (Simple)
- Total_amount (Derived)

CUSTOMER:

- Customer_ID (Key, Simple)
- Name (Composite: First_name, Last_name)
- Join_date (Simple)
- Email (Simple)
- Phone (Multivalued)
- Address (Composite: Street, City, Zip_code, Country)

PRODUCT:

- Product_ID (Key, Simple)
- Product_name (Simple)
- Description (Simple)
- Category (Simple)
- Price (Simple)
- Weight (Simple)

SUPPLIER:

- Supplier_ID (Key, Simple)
- Supplier_name (Simple)
- Email (Simple)
- Phone (Multivalued)
- Contact_person (Composite: First_name, Last_name)
- Address (Composite: Street, City, Zip_code, Country)

ORDER_LINE:

- Line_number (Partial Key)
- Unit_price (Simple)
- Quantity (Simple)
- Discount (Simple)

Relationships and Cardinalities:

1. **works_for** (EMPLOYEE N:1 DEPARTMENT)
    - Participation: Total for both entities
2. **manages** (EMPLOYEE 1:1 DEPARTMENT)
    - Participation: Total for DEPARTMENT, Partial for EMPLOYEE
3. **oversees** (DEPARTMENT 1:N WAREHOUSE)
    - Participation: Partial for DEPARTMENT, Total for WAREHOUSE
4. **works_at** (EMPLOYEE N:1 WAREHOUSE)
    - Participation: Partial for both entities
5. **processes** (EMPLOYEE 1:N ORDER)
    - Participation: Partial for EMPLOYEE, Total for ORDER
6. **places** (CUSTOMER 1:N ORDER)
    - Participation: Partial for CUSTOMER, Total for ORDER
7. **contains** (ORDER 1:N ORDER_LINE) - Identifying relationship

- Participation: Total for both entities

8. **appears_in** (PRODUCT 1:N ORDER_LINE)
    - Participation: Partial for PRODUCT, Total for ORDER_LINE
9. **stored_in** (PRODUCT M:N WAREHOUSE)
    - Attributes: Quantity_on_hand, Reorder_level
    - Participation: Partial for both entities
10. **supplies** (SUPPLIER M:N PRODUCT)
    - Attributes: Supply_date, Unit_cost
    - Participation: Partial for SUPPLIER, Total for PRODUCT

4.9.8 Constructing the Complete Company ER Diagram

Now that we have systematically identified all the components through our sentence-by-sentence analysis, we can construct the complete ER diagram by following the same structured approach we used for the University system:

Step 1: Requirements Analysis and Business Rule Identification

- Carefully read and parse each sentence
- Identify business rules and constraints
- Look for temporal indicators, ownership patterns, and dependencies

Step 2: Entity Identification

- Look for **nouns** that represent objects of interest
- Apply the four key questions from Section 4.6.5:
    - Is it important to the system?
    - Does it exist independently?
    - Do we need multiple instances?
    - Can instances be uniquely identified?

Step 3: Draw Entities and Their Attributes

Begin by placing all identified entities on the diagram using appropriate notation. Strong entities (EMPLOYEE, DEPARTMENT, WAREHOUSE, ORDER, CUSTOMER, PRODUCT, SUPPLIER) use single rectangles, while the weak entity ORDER_LINE uses a double rectangle. Add all attributes using the correct symbols: single ovals for simple attributes, double ovals for multivalued attributes (Phone for EMPLOYEE, CUSTOMER, and SUPPLIER), dashed ovals for derived attributes (Total_amount in ORDER), and composite attribute trees for complex attributes (Name and Address in multiple entities). Key attributes are underlined, and the partial key Line_number in ORDER_LINE has dashed underlining.

Step 4: Add Relationships and Cardinalities

Connect entities through relationship diamonds, with the identifying relationship "contains" (ORDER to ORDER_LINE) using a double diamond. Label each relationship appropriately and add cardinality constraints on the connecting lines. The Company system includes several 1:N relationships (works_for, oversees, processes, places, appears_in), one 1:1 relationship (manages), two M:N relationships (stored_in, supplies), and one identifying relationship for the weak entity.

Step 5: Specify Participation Constraints

Apply participation constraints based on business rules. Use double lines for total participation: EMPLOYEE and DEPARTMENT in works_for (all employees must work for departments, all departments must have employees), ORDER and ORDER_LINE in contains (orders must have lines, lines must belong to orders), ORDER in processes and places (all orders must be processed and placed), WAREHOUSE in oversees (all warehouses must be overseen), ORDER_LINE in appears_in (all order lines must reference products), and PRODUCT in supplies (all products must have suppliers). Use single lines for partial participation in manages, works_at, and stored_in relationships.

Step 6: Add Relationship Attributes

Connect relationship attributes to their respective relationship diamonds. The stored_in relationship includes Quantity_on_hand and Reorder_level attributes for inventory management, while the supplies relationship includes Supply_date and Unit_cost attributes for procurement tracking.

Step 7: Validate and Refine

Review the complete diagram against business requirements to ensure all operational needs are captured. Verify that the diagram supports essential business processes like order processing, inventory management, employee organization, and supplier coordination.

The complete ER diagram resulting from this systematic construction process is shown in Figure 4.88, which integrates all the components we identified through our detailed requirements analysis of the Company Management System.

Figure 4.88 Description: Complete ER diagram showing all seven entities (EMPLOYEE, DEPARTMENT, WAREHOUSE, ORDER, CUSTOMER, PRODUCT, SUPPLIER, ORDER_LINE), their complete attribute structures, all relationships with proper cardinality and participation constraints, using Chen notation with appropriate symbols for weak entities, composite attributes, multivalued attributes, derived attributes, and identifying relationships.

This systematic analysis demonstrates how careful examination of requirements leads to a complete and accurate ER diagram. Each sentence contributes specific components that, together, form a comprehensive data model for the company's system, following the same methodology we applied to our university example.

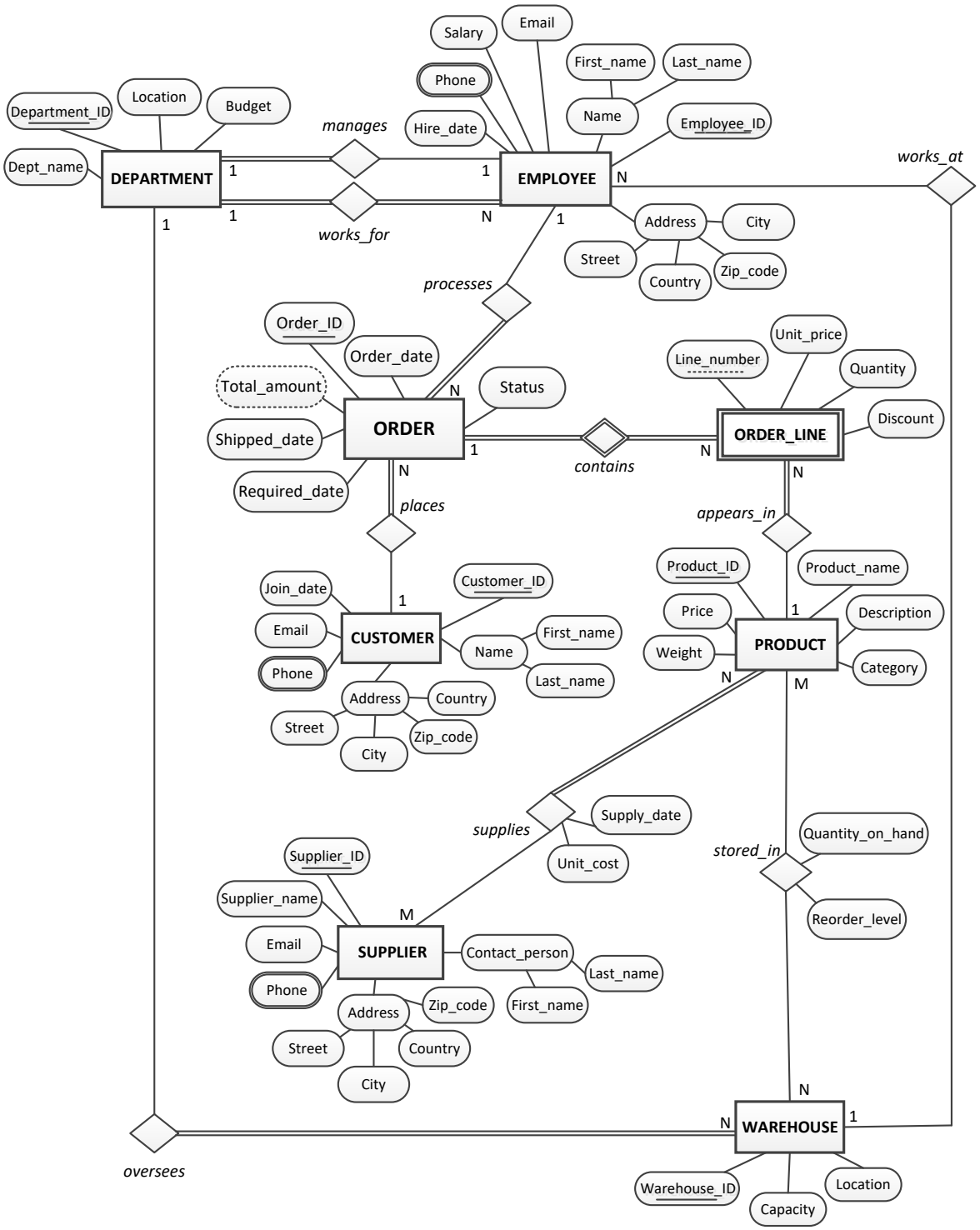


Figure 4.88: Complete Company Database ER Diagram - Integrating All Components from Requirements Analysis

4.10 Bringing It All Together

Our sentence-by-sentence approach is designed to reflect how stakeholders naturally think about their organizations. When the student affairs employee says that “students enroll in courses and receive grades,” they are describing the academic rules at the university. Later, this explanation is turned into database tables with rules and constraints. Our method

aims to respect this natural way of thinking while providing the technical accuracy needed for database implementation.

Our approach follows a minimal data rule: everything in our diagrams is essential and has a real purpose. All elements reflect how the university actually functions. In our method, the elements didn't appear fully formed at first. For example, the STUDENT entity developed as we identified attributes, established relationships, and understood constraints. Every element in the process serves a genuine business purpose.

The most critical aspect of our methodology is validation, which involves testing our model against real business scenarios. This includes conversations, discussions, and testing between database designers and the people who will use that database. Are all requirements met? Do we handle all possible scenarios? How will our model perform in the dynamic reality of institutional or organizational life? For example: Can our university model handle a student who withdraws from a course? Does it accommodate professors who teach multiple sections of the same course? Can our Company model support rush orders that skip normal processing steps?

This validation process can reveal some gaps in our initial understanding, allowing us to address them. Otherwise, these gaps might remain hidden until the database implementation, which is a more costly process.

In the next step, ER diagrams serve as a blueprint for database implementation (Chapter 9), through the relational model covered in Chapter 7. ER modeling is both an analytical discipline and a creative process. If our ER diagram is well-designed, it will translate smoothly into efficient database structures. Poorly designed ER diagrams can create implementation issues that will persist throughout the entire lifetime of the database system.

In Chapter 5, we will explore Enhanced ER (EER) modeling, where we'll learn to manage inheritance, specialization, and other advanced modeling concepts that extend the capabilities of basic ER techniques learned in this chapter.

4.11 Chapter Summary

In this chapter, we explored the Entity-Relationship (ER) model as the foundation of conceptual database design. We introduced Chen notation as a precise visual language for ER modeling and explained why it remains the most effective approach for conceptual design.

We systematically analyzed the core components of ER diagrams: entities (both strong and weak), attributes (simple, composite, multivalued, derived, and key), and relationships (unary, binary, and ternary) with their cardinality and participation constraints. We showed how to identify entities in requirements, classify different attribute types, and design relationships that accurately reflect business rules.

The chapter provided an in-depth overview of weak entities and their identifying relationships, illustrating how existence dependencies are modeled using double rectangles, double diamonds, and composite keys. We covered relationship design principles that balance business accuracy with clear implementation, and pointed out common design mistakes to avoid.

Finally, we demonstrated the full process of creating ER diagrams from scratch using two detailed case studies: a University Management System and a Company Management System. Through sentence-by-sentence analysis, we showed how raw requirements turn into complete, validated ER diagrams that act as blueprints for database implementation.

The methodologies and principles outlined in this chapter establish the foundation for effective conceptual database design, ensuring that systems accurately represent organizational reality while remaining practical to implement and maintain.

■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■

Review Questions

4.1. What is an Entity-Relationship (ER) model? Explain the historical context of its development by Dr. Peter Chen and why it remains relevant today.

4.2. Define the following Chen notation elements and explain their visual representation: a) Strong entity vs. weak entity b) Simple, composite, multivalued, and derived attributes c) Binary, unary, and ternary relationships

4.3. Explain the difference between cardinality and participation constraints in relationships. Provide examples of each type (1:1, 1:N, M:N) and (total, partial).

4.4. What is a weak entity? Describe the characteristics that distinguish weak entities from strong entities and explain when identifying relationships are needed.

4.5. Discuss the concept of key attributes and partial keys. How do they differ, and why are partial keys only found in weak entities?

4.6. Compare and contrast the four types of attributes (simple, composite, multivalued, derived) using examples from the University or Company systems presented in this chapter.

4.7. Explain the relationship design principles discussed in section 4.8.8. Why is it important to balance business accuracy with implementation clarity?

4.8. What are the common mistakes in relationship design identified in section 4.8.9? Provide specific examples and explain how to avoid each mistake.

4.9. Describe the sentence-by-sentence analysis methodology for creating ER diagrams. What are the advantages of this systematic approach?

4.10. What is the purpose of domains in attribute specification? Provide examples of domain constraints from both the University and Company systems.

4.11. Discuss the validation process for ER diagrams. How do you test whether an ER model accurately captures business requirements?

■ ■ ■ ■ ■ ■

Exercises

4.1. Entity and Attribute Identification. Consider the following requirements for a Library Management System: "A library has books, members, and staff. Each book has an ISBN, title, author, publication year, and category. Members have a membership ID, name,

address, phone numbers, and join date. Staff members have employee IDs, names, positions, and hire dates."

a) Identify all entities in this description. b) Classify each attribute as simple, composite, multivalued, or derived. c) Determine the key attributes for each entity. d) Draw the entities with their attributes using Chen notation

4.2. Relationship Analysis Using the Library Management System from Exercise 4.1, analyze these additional requirements: "Members borrow books from the library. A member can borrow multiple books, and each book can be borrowed by different members over time. Staff members manage the borrowing process. Each borrowing transaction must be handled by exactly one staff member."

a) Identify all relationships and their participating entities. b) Determine the cardinality for each relationship. c) Specify participation constraints (total or partial) for each entity. d) Identify any relationship attributes. e) Draw the complete ER diagram using Chen notation

4.3. Weak Entity Recognition. Consider this Hospital Management System requirement: "A hospital has patients and rooms. Each patient can have multiple visits, and each visit belongs to exactly one patient. Visits cannot exist without a patient. For each visit, we track the visit number, admission date, discharge date, and diagnosis."

a) Identify which entity should be modeled as a weak entity and explain why. b) Determine the partial key for the weak entity. c) Draw the identifying relationship using proper Chen notation. d) Explain what happens to visits if a patient record is deleted

4.4. Cardinality and Participation Analysis. For each of the following business scenarios, determine the cardinality and participation constraints:

a) "Each employee works for exactly one department, and each department has multiple employees." b) "Students can enroll in multiple courses, and courses can have multiple students enrolled." c) "Every order must be placed by exactly one customer, but customers may not have placed any orders yet." d) "An employee may manage at most one department, but not every employee is a manager."

4.5. ER Diagram Construction. Design a complete ER diagram for an Online Bookstore System with the following requirements:

"The bookstore sells books to customers. Customers have IDs, names, email addresses, and shipping addresses. Books have ISBNs, titles, authors, prices, and stock quantities. Customers place orders, and each order contains multiple order items. Each order item references one book and includes quantity and line price. Orders have order numbers, order dates, and total amounts. The bookstore sources books from publishers. Publishers have IDs, names, and contact information. Each book is published by one publisher."

a) Identify all entities and classify them as strong or weak. b) List all attributes and their types for each entity. c) Identify all relationships with their cardinalities and participation constraints. d) Draw the complete ER diagram using Chen notation. e) Validate your design against the given requirements

4.6. Sentence-by-Sentence Analysis Practice. Apply the sentence-by-sentence analysis methodology to this Restaurant Management System:

"The restaurant has tables, customers, and menu items. Customers make reservations for tables. Each reservation is for one table and includes the reservation date, time, and party size. Menu items have names, descriptions, categories, and prices. Customers order menu items during their visit. Each order includes the quantity of each menu item and any special instructions."

a) Analyze each sentence separately. b) Identify entities, attributes, and relationships for each sentence. c) Determine cardinalities and participation constraints. d) Create the complete ER diagram. e) Validate the design against realistic scenarios

4.7. Advanced Relationship Design. Design an ER model for a University Course Scheduling System:

"The university offers courses that are taught by professors in specific classrooms during certain time slots. A course can have multiple sections, each taught by one professor in one classroom during specific days and times. Professors can teach multiple course sections, and classrooms can host multiple course sections at different times. Students enroll in specific course sections."

a) Handle the complex many-to-many-to-many relationships appropriately. b) Consider whether any entities should be weak. c) Include all necessary attributes and constraints. d) Justify your design decisions

4.8. Complex ER Diagram Construction - Hospital Management System. Design a complete ER diagram for a comprehensive Hospital Management System with the following requirements:

"The hospital has doctors, nurses, patients, departments, and medical equipment. Each doctor has a license number, name, specialization, phone numbers, and years of experience. Nurses have employee IDs, names, shift schedules, certifications, and contact information. Patients have patient IDs, names, addresses, phone numbers, insurance information, and emergency contacts.

The hospital is organized into departments such as Cardiology, Emergency, Surgery, and Pediatrics. Each department has a department ID, name, location, budget, and head doctor. Doctors work for departments. Each doctor must work for exactly one department, and each department has multiple doctors. One doctor from each department serves as the department head.

Patients are admitted to the hospital and assigned to departments based on their medical needs. Each admission has an admission number, admission date, discharge date, and diagnosis. An admission cannot exist without a patient. During their stay, patients are treated by doctors. A patient can be treated by multiple doctors, and a doctor treats multiple patients. For each treatment, the system tracks the treatment date, procedure performed, and treatment notes.

Nurses are assigned to patients during their admission. A nurse can care for multiple patients, and a patient can be cared for by multiple nurses during different shifts. The system tracks which nurse cared for which patient on which date and shift.

The hospital has medical equipment, including X-ray machines, MRI scanners, and ventilators. Each piece of equipment has an equipment ID, name, model, manufacturer, purchase date, and maintenance schedule. Equipment is assigned to departments. Each piece of equipment belongs to exactly one department, but departments can have multiple pieces of equipment.

Doctors use equipment for patient treatments. The system tracks which doctor used which equipment for which patient treatment, including the usage date and duration. Medical equipment requires regular maintenance performed by external service providers. Each maintenance record includes the service provider name, maintenance date, type of service, and cost."

a) Identify all entities and classify them as strong or weak b) List all attributes for each entity, specifying their types (simple, composite, multivalued, derived) c) Identify all relationships with their degrees (unary, binary, ternary) d) Determine cardinalities and participation constraints for each relationship e) Identify any relationship attributes f) Draw the complete ER diagram using Chen notation g) Validate your design by testing it against realistic hospital scenarios

4.9. Complex ER Diagram Construction - E-Commerce Platform Design a complete ER diagram for a comprehensive E-Commerce Platform with the following requirements:

"The e-commerce platform connects customers, sellers, products, and orders. Customers have customer IDs, usernames, passwords, email addresses, phone numbers, names, and multiple shipping addresses. Sellers have seller IDs, business names, contact persons, email addresses, phone numbers, business addresses, and tax identification numbers.

Products are listed by sellers on the platform. Each product has a product ID, name, description, category, brand, price, weight, dimensions, and stock quantity. A product is listed by exactly one seller, but sellers can list multiple products. The platform organizes products into categories such as Electronics, Clothing, Books, and Home & Garden. Each category has a category ID, name, and description.

Customers place orders for products. Each order has an order ID, order date, shipping address, payment method, order status, and total amount. The total amount is calculated from all order items. Orders contain order items, and each order item belongs to exactly one order. Order items cannot exist without orders. Each order item includes a line number, product reference, quantity, unit price, and line total.

The platform supports multiple payment methods, including credit cards, PayPal, and bank transfers. Each payment method has specific attributes: credit cards have card numbers, expiration dates, and CVV codes; PayPal accounts have email addresses; bank transfers have account numbers and routing numbers.

Customers can write reviews for products they have purchased. Each review has a review ID, rating (1-5 stars), review text, review date, and helpfulness votes. A customer can write multiple reviews, and a product can receive multiple reviews, but each customer can review each product only once.

The platform has warehouses where products are stored before shipping. Each warehouse has a warehouse ID, location, capacity, and contact information. Products are stored in warehouses, and the system tracks the quantity of each product in each warehouse and the reorder level that triggers restocking.

Sellers supply products to warehouses through shipments. Each shipment has a shipment ID, sender (seller), receiver (warehouse), shipping date, delivery date, and tracking number. Shipments contain shipment items that specify which products and quantities are being shipped."

a) Identify all entities and classify them as strong or weak b) Design appropriate attribute structures including composite and multivalued attributes c) Handle the multiple types of payment methods appropriately d) Identify all relationships and their cardinalities e) Determine participation constraints for each entity in each relationship f) Identify relationship attributes where appropriate g) Draw the complete ER diagram using Chen notation h) Consider how the system handles product reviews, inventory management, and order fulfillment

4.10. Complex ER Diagram Construction - Airline Reservation System Design a complete ER diagram for a comprehensive Airline Reservation System with the following requirements:

"The airline operates flights between airports using aircraft. Airports have airport codes, names, cities, countries, and time zones. Aircraft have registration numbers, models, manufacturers, seating capacities, and maintenance schedules. The airline has different aircraft types, such as Boeing 737, Airbus A320, each with specific seating configurations.

Flights are scheduled services between airports. Each flight has a flight number, departure airport, arrival airport, scheduled departure time, scheduled arrival time, and flight duration. The same flight number operates on multiple dates, creating flight instances. Each flight instance has a specific date, actual departure time, actual arrival time, and status (scheduled, delayed, cancelled, completed).

Aircraft are assigned to flight instances. Each flight instance is operated by exactly one aircraft, but aircraft can operate multiple flight instances over time. The system tracks which aircraft operated which flight instance on which date.

Passengers make reservations for flight instances. Each passenger has a passenger ID, name, passport number, nationality, date of birth, phone numbers, and email address. Reservations have reservation numbers, booking dates, and total prices. Each reservation can include multiple passengers (for family bookings) and multiple flight segments (for connecting flights).

Flight instances have seats with different classes: First Class, Business Class, and Economy Class. Each seat has a seat number, class, and availability status. When passengers make reservations, they are assigned specific seats. The system tracks seat assignments, including seat number, passenger, and flight instance.

The airline employs crew members, including pilots and flight attendants. Crew members have employee IDs, names, positions, hire dates, qualifications, and contact information.

Pilots have additional attributes such as license numbers and aircraft type certifications. Flight attendants have language skills and safety certifications.

Crew members are assigned to flight instances. Each flight instance requires a specific crew composition (captain, co-pilot, and flight attendants). The system tracks crew assignments, including crew member, flight instance, and role on that flight.

Passengers can purchase tickets through different channels: online, travel agents, or airport counters. Each ticket has a ticket number, issue date, price, fare class, and payment information. Tickets are issued for specific reservations and flight instances.

The airline has frequent flyer programs where passengers earn miles for flights. Each passenger can be a member of the frequent flyer program with a membership number, tier level (Silver, Gold, Platinum), current mile balance, and enrollment date. Passengers earn miles based on flight distance and fare class."

a) Identify all entities, distinguishing between strong and weak entities b) Handle the temporal aspects (scheduled vs. actual times, flight instances) c) Design the complex relationships between flights, aircraft, crew, and passengers d) Model the seat assignment process appropriately e) Handle the frequent flyer program and mile earning system f) Determine all cardinalities and participation constraints g) Identify relationship attributes for crew assignments, seat assignments, and mile earning h) Draw the complete ER diagram using Chen notation i) Validate your design against realistic airline operation scenarios such as flight delays, crew changes, and passenger rebooking

References and Recommended Readings

Primary References

Chen, P. P. (1976). The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, 1(1), 9-36.

Date, C. J. (2003). *An Introduction to Database Systems* (8th ed.). Addison-Wesley.

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of Database Systems* (7th ed.). Pearson.

Additional References

Connolly, T. M., & Begg, C. E. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson.

Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database Systems: The Complete Book* (2nd ed.). Prentice Hall.

Teorey, T. J., Lightstone, S. S., Nadeau, T., & Jagadish, H. V. (2011). *Database Modeling and Design: Logical Design* (5th ed.). Morgan Kaufmann.

Historical and Theoretical Foundations

Chen, P. P. (1980). Entity-relationship modeling: Historical events, future trends, and lessons learned. In *Software pioneers* (pp. 296-310). Springer.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), 377-387.

Contemporary Perspectives

Batini, C., Ceri, S., & Navathe, S. B. (1992). *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings.

Song, I. Y., Evans, M., & Park, E. K. (1995). A comparative analysis of entity-relationship diagrams. *Journal of Computer and Software Engineering*, 3(4), 427-459.

Practical Implementation Guides

Hernandez, M. J. (2013). *Database Design for Mere Mortals: A Hands-On Guide to Relational Database Design* (3rd ed.). Addison-Wesley.

Stephens, R. K. (2009). *Beginning Database Design: From Novice to Professional* (2nd ed.). Apress.

Advanced Topics

Abiteboul, S., Hull, R., & Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

Ullman, J. D., & Widom, J. (2008). *A First Course in Database Systems* (3rd ed.). Prentice Hall.

Chapter 5

5. Enhanced ER (EER) Modeling

5.1 From ER to EER: When Basic Modeling Isn't Enough

The University and Company ER diagrams in Chapter 4 represent the essential data requirements for these organizations. However, as they evolve, new requirements can arise and challenge the existing ER model. They are not failures in our initial diagram; they are new requirements that emerge as the organization evolves and new business rules appear.

5.1.1 Example of the Evolution of Requirements

Several months after implementing the University database system described in Chapter 4, the university administration presented additional requirements. They emerged from rules and policies that were not examined in detail during the initial modeling of the system, but are necessary for proper system functioning:

"Our current STUDENT entity cannot adequately represent the diverse nature of our students. Graduate students conduct research under a professor's mentorship and must complete a thesis (for a master's program) or a dissertation (for a PhD program). They pursue either a one-year Master's program (60 credits), a two-year Master's program (120 credits), or a three-year PhD program (180 credits). Undergraduate students follow a different path. They either complete 240 credits over four years, or 180 credits over three years."

Similarly, the Human Resources (HR) department has additional requirements for the implemented Company system:

"We identified some limitations in our current EMPLOYEE entity. It can't capture the different rules and roles of our employees. Managers require budget authority tracking and staff supervision capabilities. Sales representatives need territory assignments and commission calculations. Warehouse workers need shift scheduling and tracking of safety certifications. Administrative staff needs software access management and multi-department support capabilities."

5.1.2 Limitations of ER Modeling

The Additional Attributes Problem

To handle these additional requirements, we can consider extending our existing STUDENT entity with additional attributes, for example:

STUDENT Entity (Extended with Additional Attributes):

- Student_ID (Key) *// attribute in the existing ER diagram*
- Name *// attribute in the existing ER diagram*
- Birth_date *// attribute in the existing ER diagram*
- Address *// attribute in the existing ER diagram*
- Email *// attribute in the existing ER diagram*
- Major_field *// additional attribute - only for undergraduates*
- Study_year *// additional attribute - only for undergraduates*
- Program_duration *// additional attribute - only for graduate students*
- Master_thesis_topic *// additional attribute - only for graduate students*
- Research_area *// additional attribute - only for PhD students*
- Dissertation_topic *// additional attribute - only for PhD students*

This approach creates some problems:

1. **Sparse Data Storage**: There will be a lot of null values for most student records. For example, all undergraduate students will have null values for graduate-specific attributes (Master_thesis_topic, Research_area, Dissertation_topic), while all graduate students will have null values for undergraduate-specific attributes (Major_field, Study_year). This leads to wasted storage space and complicates queries.
2. **Implementation of the Rules**: How do we ensure that only graduate students have thesis mentors (advisors)?
3. **Query Complexity**: Finding all graduate students requires checking for non-null values in graduate-specific attributes.
4. **Maintenance Issue**: Adding new student types requires modifying the core STUDENT entity, potentially affecting all existing applications.

The Multiple Entity Problem

Another approach is to create separate entities for each student type:

- BACHELOR_STUDENT
- MASTERS_STUDENT
- DOCTORAL_STUDENT

This approach avoids the sparse data problem but creates new challenges:

1. **Attribute Redundancy**: Common student attributes (Student_ID, Name, Address, etc.) must be repeated in each entity.

2. **Relationship Duplication**: Each student type needs its own enrollment relationships with courses, creating multiple parallel relationship structures.
3. **Inconsistent Updates**: Changes to common student attributes require coordinating updates across multiple entities.
4. **More Complicated Queries**: Finding all students regardless of type requires union operations across multiple entities.

All these problems require a new solution and approach: creating an **Enhanced Entity-Relationship (EER)** diagram.

> **Definition: Enhanced Entity-Relationship (EER) Model**
>
> The Enhanced Entity-Relationship (EER) model extends the basic ER model by adding concepts such as specialized entity types, inheritance relationships, and complex classification schemes (Teorey et al., 1986; Elmasri & Navathe, 2016).

We will demonstrate EER modeling using our case studies of a university and a company.

5.1.3 New Scenarios Requirements

🎓 *University Scenario 1: Differentiated Academic Mentoring*

Requirements: "Undergraduate students are assigned academic advisors who assist with course selection, major requirements, and graduation planning. Graduate students work with research advisors or mentors who supervise thesis/dissertation work. For each student type, there are completely different attributes, constraints, and requirements."

Basic ER Limitation: Our current single relationship between students and professors cannot distinguish between academic and research advising.

🎓 *University Scenario 2: Credit System Management*

Requirements: "Undergraduate students must register for exactly 5 courses per semester (30 credits total), comprising core courses, major electives, and minor electives, each worth 6 credits. They must complete 240 credits over four years for graduation, or 180 credits over three years. Master's students in one-year programs take 5 courses per semester (30 credits) for 2 semesters, totaling 60 credits. Master's students in two-year programs complete 120 credits over four semesters. Doctoral students must complete 180 credits over three years, with later semesters focused on dissertation research rather than coursework."

Basic ER Limitation: This complex structure cannot be naturally represented in a single STUDENT entity.

🏢 *Company Scenario 1: Performance Management Systems*

Requirements: "Managers undergo performance reviews that focus on their leadership effectiveness, budget management, and team development. Sales representatives are evaluated based on territory performance, the quality of their customer relationships, and revenue achievement against quotas. Warehouse workers are evaluated on safety

compliance, inventory accuracy, and operational efficiency. Administrative staff is evaluated on document management quality, inter-departmental coordination, and software proficiency."

Basic ER Limitation: A single performance-review entity cannot accommodate all evaluation criteria, feedback sources, and reporting structures required for each employee type.

Company Scenario 2: Compensation Administration

Requirements: "Managers receive a base salary plus bonuses calculated on their performance and departmental metrics. Sales representatives are compensated with a base salary plus commission calculated on territory sales volume. Warehouse workers earn a base salary plus shift differential pay and safety incentive bonuses. Administrative staff receive a base salary with annual performance-based increases."

Basic ER Limitation: These compensation rules require different attributes and calculations that can't be applied to a single employee entity.

5.1.4 The Need for Semantic Richness

Basic ER models represent the essential entities, relationships, and constraints, but they can't capture the **semantic richness** of domains in which entities form hierarchical classifications with inherited characteristics, as in the scenarios above.

Semantic richness refers to a data model's ability to express not only what data exists and how it relates, but also what the data *means* (Hull & King, 1987; Smith & Smith, 1977). When we say "Sarah is a graduate student," we're invoking a set of attributes, relationships, rules, and constraints that apply specifically to graduate students and are inherited from the broader student classification.

5.1.5 The IS-A Relationship

The common factor in all these scenarios is the need for so-called **IS-A relationships**:

- A graduate student *IS-A* student (with all basic student properties plus specialized characteristics)
- A manager *IS-AN* employee (with all basic employee properties plus management-specific attributes)
- A PhD student *IS-A* graduate student (inheriting both graduate and basic student properties while adding PhD-specific requirements)

These IS-A relationships suggest hierarchies where specialized types inherit characteristics from more general types while adding their own specific properties and constraints.

> **Key Definition: IS-A Relationship**
>
> An IS-A relationship represents a classification hierarchy in which a specialized entity type (subclass) inherits all attributes and relationships from a more general entity type (superclass), while potentially adding its own specialized characteristics (Smith & Smith, 1977; Elmasri & Navathe, 2016).

5.1.6 Preview: The EER Solution

Enhanced ER modeling addresses these limitations through a few concepts from the EER literature (Teorey et al., 1986; Elmasri & Navathe, 2016) that we'll explore throughout this chapter:

Specialization and Generalization: Formal mechanisms for creating and organizing entity type hierarchies that reflect natural classifications while eliminating attribute and relationship redundancy.

Inheritance Mechanisms: The automatic transfer of attributes and relationships from general types (superclasses) to specialized types (subclasses).

Constraint Systems: Rules that specify how entities can belong to specialized types, including disjoint/overlapping constraints and total/partial participation requirements.

Semantic Precision: Visual representations and formal definitions that clarify the meaning and structure of entity hierarchies are understandable to both technical implementers and business stakeholders.

5.1.7 The Path Forward

This chapter builds upon the requirements analysis techniques presented in Chapter 3 and the ER construction methods introduced in Chapter 4, demonstrating how EER concepts emerge from real organizational needs.

In the following sections, we'll extend our University and Company case studies with EER concepts, demonstrating how the limitations we've identified can be resolved. We'll see how our basic ER diagrams from Chapter 4 evolve into EER diagrams that capture the full semantic richness.

5.2 Subclasses and Superclasses: The Foundation of EER

We need a formal mechanism and a theoretical foundation to capture the semantic richness that basic ER modeling cannot express: the concepts of subclasses and superclasses.

5.2.1 The Concept of Subclasses and Superclasses

To understand subclasses and superclasses, we will consider our University system and analyze the limitations of the basic ER approach:

Entity Analysis from Our University System:

Consider three actual student tuples from our University system:

- **S001**: Birth_date: 25.08.1995, First_name: Sarah, Last_name: Smith, enrolled in CS303 (Database Systems) with Grade: 9, Faculty: Computer Science
- **S002**: Birth_date: 20.11.2002, First_name: Bojan, Last_name: Jovansen, enrolled in EE201 (Circuit Analysis) and EE301 (Advanced Circuit Analysis), Faculty: Electrical Engineering
- **S003**: Birth_date: 10.03.2004, First_name: Kosta, Last_name: Dojcinoski, enrolled in MA301 (Linear Algebra) and MA401 (Matrix theory and advanced vector spaces), Faculty: Bioengineering

The Classification Challenge:

It is clear that all three individuals are students and share student characteristics, including Student_ID, Name, Birth_date, and Address. However, they represent different academic levels or different program types that would require specialized attributes:

- Sarah (S001) appears to be an undergraduate Computer Science student taking CS303
- Bojan (S002) is enrolled in multiple Electrical Engineering courses (EE201 and EE301), suggesting progression through the program
- Kosta (S003) is taking advanced Mathematics courses (MA301 and MA401), indicating upper-year or graduate-level study

This scenario illustrates the need for **subclasses and superclasses**.

> **Definition: Subclass and Superclass**
>
> A **subclass** (or subtype) is a distinct entity type whose instances are also instances of another entity type called the **superclass** (or supertype). The subclass represents a specialized subset of the superclass, inheriting all attributes and relationships from the superclass while adding its own specialized characteristics.

Formal Relationship: If S is a subclass of superclass C, then S ⊆ C (S is a subset of C) (Elmasri & Navathe, 2016; Smith & Smith, 1977).

5.2.2 Identity of the Entity

A subclass entity and its corresponding superclass entity represent **the same real-world object**, viewed at different levels of detail and specialization.

***Example from Our University System*:**

When Bojan Jovansen (S002) exists as both:

- A STUDENT entity (with Student_ID: S002, First_name: Bojan, Last_name: Jovansen, Birth_date: 20.11.2002)
- A potentially specialized student entity (enrolled in multiple EE courses, suggesting advanced or graduate-level study)

These are not two separate people or two separate database entities. Instead, they represent **the same person** (Bojan) viewed through different concepts:

- As a STUDENT, Bojan shares common characteristics with all university students
- As a specialized student type, Bojan may have characteristics specific to his program level and field of study

This **characteristic of multiple conceptual views** distinguishes subclass/superclass relationships from regular ER relationships (Elmasri & Navathe, 2016; Smith & Smith, 1977).

5.2.3 The Superclass/Subclass Relationship

The relationship between a superclass and its subclasses is called a **superclass/subclass relationship** or simply a **class/subclass relationship**. This relationship differs from the ones we studied in Chapter 4.

Comparison with Basic ER Relationships:

Basic ER Relationship (Chapter 4):

STUDENT —— enrolls_in ⟶ COURSE

(Two separate entities: a student and a course)

Superclass/Subclass Relationship (EER):

STUDENT —— IS-A ⟶ MASTERS_STUDENT

(Same entity viewed at different specialization levels)

***University System Class/Subclass Relationships*:**

From our case study, we can identify several class/subclass relationships:

1. **STUDENT/BACHELOR_STUDENT**: Every undergraduate (bachelor) student is a student
2. **STUDENT/MASTERS_STUDENT**: Every master's student is a student
3. **STUDENT/DOCTORAL_STUDENT**: Every doctoral (PhD) student is a student
4. **PROFESSOR/REGULAR_PROFESSOR**: Every regular professor is a professor
5. **PROFESSOR/VISITING_PROFESSOR**: Every visiting professor is a professor

5.2.4 When Subclasses Are Needed

Based on our analysis of the reference literature (Teorey et al., 1986; Teorey et al., 2011; Elmasri & Navathe, 2016) and our case studies, subclasses are necessary in two situations:

Situation 1: Specialized Attributes

When different types of entities require attributes that don't apply to all members of the general entity type.

University Example - Specialized Student Attributes

STUDENT (Superclass Attributes):

- Student_ID, Name, Birth_date, Address, Email

BACHELOR_STUDENT (Additional Local Attributes):

- Major_field, Study_year, Total_ECTS_required, Defense_commitee

MASTERS_STUDENT (Additional Local Attributes):

- Program_duration, Master_thesis_topic, Total_ECTS_required, Defense_commitee

DOCTORAL_STUDENT (Additional Local Attributes):

- Research_area, Dissertation_topic, Total_ECTS_required, Defense_commitee, Qualifying_exam_date, Comprehensive_exam_date, Expected_defense_date

Company Examples - Specialized Employee Attributes:

From our Company system data, we can see different employee types that would benefit from specialization:

EMPLOYEE (Superclass Attributes):

- Employee_ID, Name, Email, Salary, Hire_date, Phone, Address, Department_ID, Warehouse_ID

Potential MANAGER_EMPLOYEE subclass (based on employees who have Manages_Dept_ID) attributes:

- Management_level, Budget_authority, Strategic_planning_responsibility, Leadership_training_complete

Potential WAREHOUSE_EMPLOYEE subclass (based on Warehouse_ID assignments) attributes:

- Primary_warehouse_ID, Physical_location, Inventory_management_system_trained, Shift_schedule

Potential SALES_EMPLOYEE subclass (based on employees in Sales department D1) attributes:

- Sales_territory, Customer_portfolio_size, Commission_rate, Monthly_sales_quota

Looking at our employee data, employee E1 (Zoran Pejovski) works in the Sales department (D1) and also manages it (D1), demonstrating that a single employee can hold multiple roles that would benefit from specialized modeling.

Note: This represents a subset of the complete employee specialization structure. Section 5.3.1.2 presents the full implementation, including comprehensive attributes, and introduces employment type as a second independent specialization dimension.

Situation 2: Specialized Relationships

When some relationship types can only be participated in by specific subtypes of entities.

University Examples - Specialized Relationships:

- **bachelor_thesis_supervision**: BACHELOR_STUDENT entities work with mentors for their bachelor's/final thesis (typically 6-12 credits)

- **masters_thesis_supervision**: MASTERS_STUDENT entities work with supervisors for a research thesis (typically 18-30 credits depending on one-year/two-year program)
- **doctoral_dissertation_supervision**: DOCTORAL_STUDENT entities work with supervisors and committees for original research dissertations
- **academic_advising**: All students receive academic advising, but undergraduate students focus on course selection
- **research_methodology_training**: Graduate students (Master's/Doctoral) require training in research methods
- **thesis_defense_committee**: All students defend before committees, but committee composition varies by level (2 members for bachelor's, 3 members for master's, 5 members for doctoral)

🏢 ***Company Examples - Specialized Relationships*:**

- **budget_approval**: Only MANAGER_EMPLOYEE entities can participate in budget approval processes
- **performance_review_conducting**: Only MANAGER_EMPLOYEE entities conduct performance reviews for their employees
- **customer_territory_assignment**: Only SALES_EMPLOYEE entities are assigned to sales territories
- **shift_assignment**: Only WAREHOUSE_EMPLOYEE entities are assigned to warehouse shifts and physical inventory locations
- **document_management**: Only ADMIN_EMPLOYEE entities participate in official document processing and archive management

5.2.5 Attribute and Relationship Inheritance

Subclasses **inherit** all attributes and relationships automatically **from their superclass**.

Inheritance Principle (Elmasri & Navathe, 2016; Smith & Smith, 1977):

Every subclass automatically inherits:

1. **All attributes** from its superclass
2. **All relationships** from its superclass
3. **All constraints** associated with superclass attributes and relationships

🎓 ***University Example - Specialized Student Inheritance*:**

Using our University data structure, a specialized student would have:

Specialized STUDENT Complete Attribute Set:

Inherited from STUDENT (from our ER model (described in Chapter 4) and relational model (Chapter 7)):

Student_ID (Key attribute - inherited), Birth_date (inherited), Age (Derived attribute - inherited), First_name, Last_name (inherited), Street, City, Country, Zip_code (Address components - inherited), Email (inherited)

Local Attributes (specific to specialized student types):

- Program_level (Bachelor, Master's, Doctoral), Specialization_field (CS, EE, MA, PH, HU based on course patterns), Academic_advisor (for undergraduates), Research_advisor (for graduate students), Thesis_topic (for Masters/PhD), Expected_graduation_date, Research_area

Inherited Relationships (from our ER model (described in Chapter 4) and relational model (Chapter 7)):

- enrolled_in (with COURSE)

- attends (with CLASS_SESSION)

- email (multivalued relationship)

Local Relationships (specific to specialized students):

- advised_by (with specific PROFESSOR types)

- thesis_committee (with PROFESSOR group for graduate students)

- research_project (with RESEARCH_PROJECT for graduate students)

Looking at our actual data (Chapter 11), student S003 (Kosta Dojcinoski) enrolled in MA301 (Linear Algebra) and MA401 (Matrix theory and advanced vector spaces), suggesting upper-year or graduate-level study, which would benefit from specialized attributes not present in our current STUDENT structure.

Key Inheritance Rules (Elmasri & Navathe, 2016; Smith & Smith, 1977):

1. **Key Inheritance**: Subclasses always inherit the key attributes from their superclass. A MASTERS_STUDENT is identified by the same Student_ID that identifies the corresponding STUDENT.
2. **Complete Attribute Inheritance**: All superclass attributes become part of the subclass, ensuring consistency and eliminating redundancy.
3. **Relationship Participation**: Subclasses can participate in all relationships defined for their superclass, plus their own specialized relationships.
4. **Constraint Inheritance**: If a superclass has total participation in a relationship, all subclasses inherit this constraint.

5.2.6 Mathematical Foundation

To provide the theoretical background, we present the formal mathematical definitions for subclass/superclass relationships (Elmasri & Navathe, 2016; Smith & Smith, 1977):

Formal Definition: Class and Entity Relationship

A **class** C is a set or collection of entities. In EER modeling, this includes entity types, subclasses, superclasses, and categories.

A **subclass** S is a class whose entities must always be a subset of those of another class, called the **superclass** C. We denote this relationship as C/S.

For any superclass/subclass relationship C/S, we must always have: $S \subseteq C$

This means that every entity in the subclass S is also an entity in the superclass C, but the superclass C may contain entities that are not in the subclass S

***Mathematical Example from Our University System*:**

Let STUDENT = {S001(Sarah), S002(Bojan), S003(Kosta), S004, S005, S006, ...} (all students)

Let COMPUTER_SCIENCE_STUDENT = {S001(Sarah), S006(Anica), ...} (students in CS (Computer Science) programs)

Let ELECTRICAL_ENG_STUDENT = {S002(Bojan), ...} (students in EE (Electrical Engineering) programs)

Let MATHEMATICS_STUDENT = {S003(Kosta), ...} (students in MA (Mathematics) programs)

Then:

COMPUTER_SCIENCE_STUDENT $\subseteq$ STUDENT (every CS student is a student)

ELECTRICAL_ENG_STUDENT $\subseteq$ STUDENT (every EE student is a student)

MATHEMATICS_STUDENT $\subseteq$ STUDENT (every MA student is a student)

Note: These subsets may overlap if we allow double majors, or be disjoint if students can only have one primary field of study.

5.2.7 Advantages of Subclass/Superclass Modeling

The subclass/superclass approach provides several critical advantages over basic ER modeling (Elmasri & Navathe, 2016; Teorey et al., 2011):

1. Semantic Precision

The model explicitly captures the classification structure. This makes the business rules and entity relationships understandable to stakeholders.

2. Attribute Organization

Common attributes are defined once in the superclass and inherited by all subclasses. This enables the elimination of redundancy and supports specialization.

3. Relationship Clarity

Different types of entities can participate in appropriately specialized relationships while maintaining their ability to participate in general relationships.

4. Constraint Enforcement

The rules about entity classifications can be formally represented and enforced at the data model level.

5. Query Flexibility

Applications can query at the appropriate level of the hierarchy. For example, we can find all students, regardless of type, or target only graduate students for research-related queries.

6. Extensibility

New specialized entity types are added by extending the hierarchy instead of modifying existing entity definitions.

In the next section, we'll explore how these concepts are organized through the processes of specialization and generalization.

5.3 Specialization and Generalization

Specialization and **generalization** are two complementary processes that create the hierarchical structures of subclasses and superclasses. They are two distinct conceptual approaches that produce the same EER diagram, and understanding both perspectives is important.

5.3.1 Specialization: Top-Down Approach

Specialization is a **top-down conceptual approach** that starts with a general entity type (superclass) and systematically identifies specialized subgroups (subclasses). Each subclass has its own characteristics, such as specific attributes or relationships that are not applicable to the superclass.

> **Definition: Specialization**
>
> Specialization is the process of defining a set of subclasses of an entity type, where each subclass represents entities that share common characteristics beyond those inherited from the superclass. Specialization introduces additional attributes and relationships for each subclass and maintains inheritance from the superclass (Elmasri & Navathe, 2016; Smith & Smith, 1977; Teorey et al., 1986).

5.3.1.1 Graphical Representation of Specialization

In Chapter 4, we explained the Chen notation for the elements of the ER diagrams. Here, we introduce specialized symbols to represent specialization.

The ISA Triangle Symbol in Specialization

The most common and widely accepted way to represent specialization (and later generalization) in EER diagrams is through the use of a **triangle symbol** containing the text **"IS A"** or **"ISA"** (Elmasri & Navathe, 2016; Hristoski & Dimovski, 2022). This triangle serves as a visual connection between the superclass and its subclasses, clearly indicating the "IS A" semantic relationship that defines specialization.

For specialization, the triangle is **oriented downward, reflecting the top-down approach.** In this process, we start with a general superclass at the top and refine it into more subclasses below (Figure 5.1). This downward orientation visualizes the direction of the specialization process: from the superclass to the subclasses.

Basic Specialization Notation Structure:

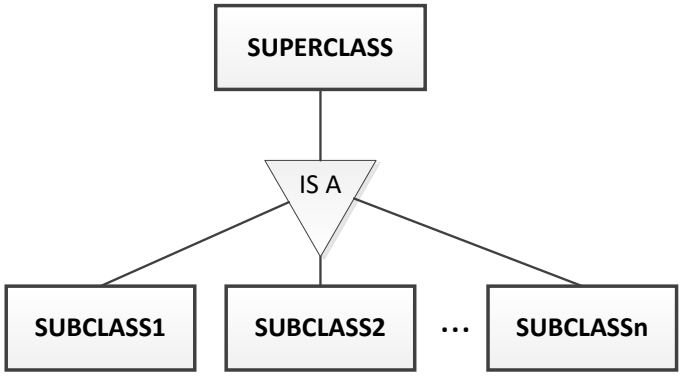


Figure 5.1: Specialization notation

Interpretation

The ISA triangle notation captures the semantic relationship: every entity in a subclass **IS A** specialized type of entity in the superclass. For example:

- Every BACHELOR_STUDENT **IS A** STUDENT
- Every MASTERS_STUDENT **IS A** STUDENT
- Every DOCTORAL_STUDENT **IS A** STUDENT

This "IS A" relationship is not just a visual notation. It represents **class membership** and **property inheritance**, where each subclass entity inherits all attributes and relationships of its superclass while adding its own specialized attributes.

University Example: Student Specialization

To illustrate this notation, we will extend the STUDENT entity from Chapter 4 (Figure 4.41) to include academic-level specializations (Figure 5.2).

In Figure 5.2, we have enhanced the STUDENT entity by adding an **ECTS_completed** attribute. ECTS stands for the European Credit Transfer and Accumulation System, a standardized framework for measuring student workload across European universities. The attribute **ECTS_completed** represents the current ECTS credits an individual student has earned in their academic career. We introduce the ECTS_completed as a common superclass attribute because every student, regardless of their academic level, accumulates credits as they progress through courses. This distinguishes it from **Total_ECTS_required**, which is dependent and varies by program type and thus belongs to the specialized subclasses (as explained below).

Figure 5.2 explained**:**

- The **STUDENT** superclass appears at the top with its general attributes
- A single line connects STUDENT to the **ISA triangle**

- The triangle points **downward**, indicating top-down specialization
- Three lines extend from the triangle's base to three subclass rectangles
- Each subclass shows only its **specialized attributes** (inherited attributes from STUDENT are not repeated). It is important to note that for illustration purposes**,** Figure 5.2 displays only a representative selection of specialized attributes for each subclass, not the complete set of all possible attributes.

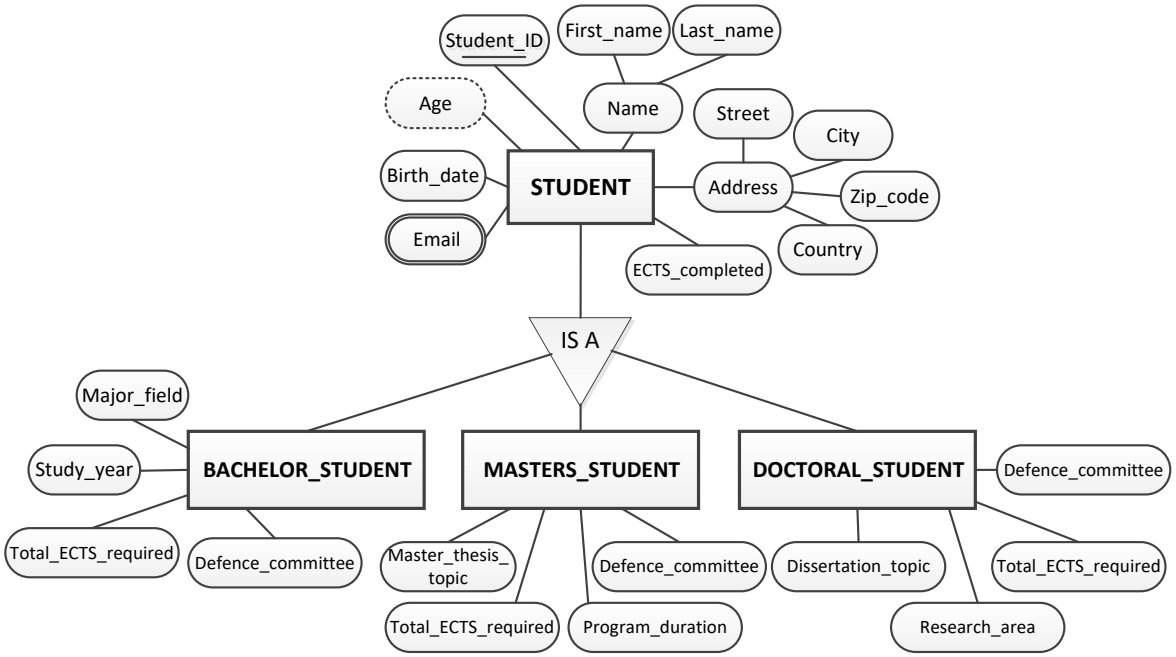


Figure 5.2: Student Specialization Using ISA Triangle Notation

Chen Notation for Figure 5.2:

1. **Superclass Rectangle** (STUDENT): Standard Chen notation entity rectangle containing common attributes shared by all students
2. **Connection Line**: Single line from the superclass to the triangle
3. **ISA Triangle**: Downward-pointing triangle containing "IS A" text. It serves as the specialization junction point
4. **Subclass Rectangles**: Standard Chen entity rectangles for specialized student types. The subclasses contain just their unique attributes; the general attributes from the superclass are not repeated
5. **Distribution Lines**: Lines from the triangle to each subclass

Semantic Meaning:

We can “read” the diagram as follows:

- Every BACHELOR_STUDENT **IS A** STUDENT (inherits all STUDENT attributes)
- Every MASTERS_STUDENT **IS A** STUDENT (inherits all STUDENT attributes)
- Every DOCTORAL_STUDENT **IS A** STUDENT (inherits all STUDENT attributes)
- Each subclass **adds specialized attributes** relevant to that academic level

- A student entity exists in the superclass and in **exactly one** subclass (assuming a disjoint constraint), which means a student is one of these three options: a bachelor student, a master student, or a doctoral student.

Visualization of Attribute Inheritance

An important principle in reading EER diagrams with specialization: **subclasses inherit all superclass attributes,** although we don't repeat them in the subclass rectangles.

For example, **BACHELOR_STUDENT** in our university case study has these attributes:

- **Inherited from STUDENT**: Student_ID, Name (composite: First_name, Last_name), Birth_date, Age (derived), Email (multivalued), Address (composite: Street, City, Zip_code, Country), ECTS_completed
- **Specialized attributes**: Major_field, Study_year, Total_ECTS_required (180 or 240 ECTS depending on program structure), Defense_committee (2 members)

The **Total_ECTS_required** attribute differs across subclasses:

- **BACHELOR_STUDENT**: Total_ECTS_required = 180 (three-year program) or 240 ECTS (four-year program)
- **MASTERS_STUDENT**: Total_ECTS_required = 60 (one-year program) or 120 ECTS (two-year program)
- **DOCTORAL_STUDENT**: Total_ECTS_required = 180 ECTS (three-year program)

European universities operating under the Bologna Process follow a three-cycle degree system (Bachelor, Master, PhD) and standardized ECTS credit requirements (European Commission, 2015). Each academic level has different general requirements, including ECTS credits. That is why the attribute **Total_ECTS_required** is placed in the subclasses rather than in the superclass. In contrast, **ECTS_completed** belongs in the STUDENT superclass because all students, regardless of level, accumulate credits in the same manner as they progress through coursework.

The **Defense_committee** attribute indicates the number of committee members required to defend the thesis or dissertation at each academic level. At our university, following common practice in the Bologna framework, committee sizes differ across academic levels:

- **BACHELOR_STUDENT**: Defense_committee = 2 members (for bachelor's thesis defense)
- **MASTERS_STUDENT**: Defense_committee = 3 members (for master's thesis defense)
- **DOCTORAL_STUDENT**: Defense_committee = 5 members (for doctoral dissertation defense)

Similar to the Total_ECTS_required attribute, **Defense_committee** is specialized and placed in the subclasses because requirements vary by academic level. At our university, bachelor's students are evaluated by a committee of two professors. Master's students conduct more substantial research and are evaluated by three professors who assess both the methodology and their contributions to the field. Doctoral students produce original

research contributions and require more rigorous evaluation by five professors, including external experts. These varying committee sizes make the Defense_committee attribute inherently specialized and not generalizable to the STUDENT superclass.

> **Complete Attribute Set** = Superclass Attributes + Subclass-Specific Attributes

Real-World Example: Sarah Smith (S001)

To illustrate how these attributes work together in practice, consider Sarah Smith (Student_ID: S001), a Bachelor's student in Computer Science at our university:

Inherited Superclass Attributes (from STUDENT):

- Student_ID: S001
- Name: Sarah Smith (First_name: Sarah, Last_name: Smith)
- Birth_date: 25.08.1995
- Age: 30 (derived, calculated in 2025)
- Email: sarah.smith@university.edu, sarah.smith@gmail.com (multivalued)
- Address: Jane Sandanski 32, Ohrid, Macedonia, 6000
- ECTS_completed: 45 credits (what she has earned so far)

Specialized Subclass Attributes (BACHELOR_STUDENT):

- Major_field: Computer Science (CS)
- Study_year: Second
- Total_ECTS_required: 240 ECTS (four-year program)
- Defense_committee_size: 2 members

Inherited Relationships:

- enrolls_in CS303 (Database Systems) with Grade: 9
- belongs_to Faculty F001 (Computer Science)

This example demonstrates how the **ECTS_completed** attribute tracks Sarah's actual progress, while **the Total_ECTS_required** attribute establishes her graduation target. As she completes more courses, ECTS_completed will increase until it reaches Total_ECTS_required. At that point, she becomes eligible to graduate and to defend her bachelor's thesis with a 2-member committee.

Next, we will validate this notation with additional examples from our University system, then examine how it applies to our Company case study.

5.3.1.2 The Specialization Process: Case Study Applications

This section demonstrates how to implement the established top-down specialization methodology, from initial entity analysis to final validation.

The Specialization Process: University Example

We demonstrate the systematic specialization process using our University system, building upon the STUDENT entity from Chapter 4 and the notation introduced in Figure 5.2.

Step 1: Identify the General Entity Type (Superclass)

Our starting point is the STUDENT entity from Chapter 4 (Figure 4.41). It contains common attributes shared by all students: Student_ID (key attribute), Name (composite attribute: First_name, Last_name), Birth_date, Age (derived attribute), Email (multivalued attribute), Address (composite attribute: Street, City, Zip_code, Country), and the newly added ECTS_completed attribute. Our current enrollment data includes students such as S001 (Sarah Smith, enrolled in CS303), S002 (Bojan Jovansen, enrolled in EE201 and EE301), and S003 (Kosta Dojcinoski, enrolled in MA301 and MA401).

Step 2: Recognize Different Characteristics

Analyzing our student population and Bologna Process requirements (European Commission, 2015), we identify characteristics that distinguish different types of students and necessitate specialized attributes:

- **Degree-level requirements**: Different academic levels have different graduation requirements that cannot be captured in a single entity
- **Research expectations:** Various academic levels involve different complexities and scopes of the required research work.
- **Academic progression tracking**: Each level requires different monitoring mechanisms
- **Committee structures**: Thesis/dissertation evaluation requires differently sized committees

This analysis demonstrates that students at different academic levels cannot be adequately described by a single set of attributes, underscoring the need for specialization to capture these distinctions.

Step 3: Create Specialized Subclasses

Based on the Bologna Process structure, we create three specialized subclasses representing distinct academic levels, as illustrated in Figure 5.3:

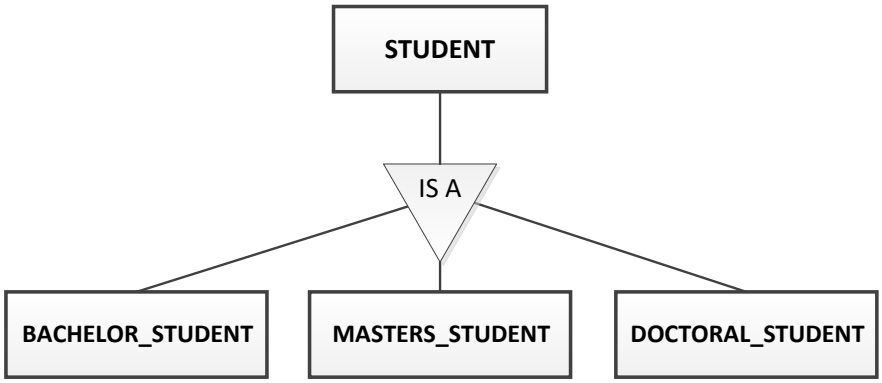


Figure 5.3: Student Specialization Hierarchy - Academic Level Classification

Each subclass represents a formal academic level with specific requirements. They inherit all STUDENT attributes while adding their own specialized attributes.

Step 4: Define Specialized Attributes

Each subclass requires its own specialized attributes that capture its unique academic requirements. As discussed in Section 5.3.1.1, these specialized attributes complement the inherited attributes of the superclass.

BACHELOR_STUDENT Specialized Attributes:

- Major_field (CS, EE, MA, PH, HU, etc.)
- Minor_field (optional)
- Study_year (First, Second, Third, Fourth)
- Total_ECTS_required (180 or 240)
- Credits_per_year_target
- Bachelor_thesis_topic
- Academic_advisor_ID (references the PROFESSOR entity/table)
- Expected_graduation_date
- Defense_committee (2 members)

MASTERS_STUDENT Specialized Attributes:

- Program_duration (one-year or two-year program)
- Total_ECTS_required (60 or 120)
- Master_thesis_topic
- Research_supervisor_ID (references the PROFESSOR entity/table)
- Research_methodology_completed (Boolean)
- Specialization_track
- Expected_defense_date
- Defense_committee (3 members)
- Previous_degree_institution

DOCTORAL_STUDENT Specialized Attributes:

- Total_ECTS_required (180)
- Dissertation_topic
- Primary_supervisor_ID (references the PROFESSOR entity/table)
- Co_supervisor_ID (optional, references the PROFESSOR entity/table)
- Research_area
- Qualifying_exam_date
- Comprehensive_exam_date

- Candidacy_status (Boolean)
- Dissertation_proposal_date
- Proposal_approval_date
- Expected_defense_date
- Defense_committee (5 members)
- Publications_count
- Teaching_obligation_hours (if Teaching Assistant (TA))

As established in Section 5.3.1.1, attributes such as Total_ECTS_required and Defense_committee vary by academic level and must therefore be specialized attributes. The attribute ECTS_completed applies uniformly across all student types and remains in the superclass.

Step 5: Establish Specialized Relationships

Beyond inheriting common relationships from STUDENT (enrolls_in COURSE, belongs_to FACULTY, attends CLASS_SESSION), each subclass participates in relationships specific to its academic level and research requirements:

BACHELOR_STUDENT Specialized Relationships:

- *academically_advised_by* (PROFESSOR): N:1
    - Each bachelor student has one academic advisor
    - Attributes: Advising_start_date, Meeting_frequency
    - Participation: Total for BACHELOR_STUDENT, Partial for PROFESSOR

MASTERS_STUDENT Specialized Relationships:

- *research_supervised_by* (PROFESSOR): N:1
    - Each master's student has one research supervisor for the thesis
    - Attributes: Supervision_start_date, Research_status
    - Participation: Total for MASTERS_STUDENT
- *defends_masters_thesis_to* (MASTERS_COMMITTEE): 1:1
    - Each master's student defends to one committee of 3 members
    - Attributes: Defense_date, Defense_result, Committee_feedback
    - Participation: Total (all must defend to graduate)

DOCTORAL_STUDENT Specialized Relationships:

- *supervised_by* (DOCTORAL_COMMITTEE): 1:1
    - Each doctoral student has a dissertation committee
    - Committee includes primary supervisor, co-supervisor, and members
    - Attributes: Committee_formation_date, Annual_review_results

    - Participation: Total for DOCTORAL_STUDENT
- *defends_dissertation_to* (DOCTORAL_DEFENSE_COMMITTEE): 1:1
    - Each doctoral student defends to one committee of 5 members
    - Attributes: Defense_date, Defense_outcome, Revisions_required
    - Participation: Total (all must defend to graduate)

This five-step specialization process demonstrates the evolution of the STUDENT entity from a basic ER model (Chapter 4, Figure 4.41) to an extended EER model that captures the complexity of different academic levels. Next, we will apply the same approach to our Company case study.

Company System Specialization Example

To capture the organization's diverse roles and employment types in our Company system, we extend the EMPLOYEE entity from Chapter 4 (Figure 4.64) using the same specialization process established for the University system.

Step 1: Identify the General Entity Type (Superclass)

Our starting point is the EMPLOYEE entity from Chapter 4, which contains common attributes shared by all employees: Employee_ID (key attribute), Name (composite attribute: First_name, Last_name), Email, Salary, Hire_date, Phone (multivalued attribute), and Address (composite attribute: Street, City, Zip_code, Country). The entity also maintains foreign key relationships, explained in Chapter 7 (Relational Model): Department_ID (works_for), Warehouse_ID (works_at, partial), and Manages_Dept_ID (manages, partial).

In our company database (see Chapter 11: Integrated Case Studies), we have eight employees (E1 through E8) working across three departments: Sales (D1), Marketing (D2), and Human Resources (D3), with some employees also assigned to warehouses (W1-W4).

Step 2: Recognize Different Characteristics

By analyzing the organizational structure of our company, we identify two independent specialization criteria for employees:

Job Function (based on role and responsibilities):

- **Managers**: Employees with Manages_Dept_ID values (E1: Zoran Pejovski manages D1, E3: Marija Naumoska manages D2, E5: Naum Stojanoski manages D3)
- **Departmental employees**: Sales (D1) require customer relationship and revenue generation skills; Marketing (D2) need campaign and brand management expertise; HR (D3) focus on personnel administration
- **Warehouse employees**: Employees with Warehouse_ID assignments (E2 at W1, E6 at W3, E8 at W2) perform inventory management

Employment Type (based on work arrangement):

- **Full-time employees**: Annual salary with standard benefits
- **Part-time employees**: Reduced work hours and hourly compensation

- **Contract employees**: Hourly rates for project-based engagement

These two independent criteria reveal that a single employee can belong to multiple categories simultaneously (e.g., E1 is both a MANAGER_EMPLOYEE by job function AND a FULL_TIME_EMPLOYEE by employment type).

Step 3: Create Specialized Subclasses

Building on the conceptual employee specialization introduced in Section 5.2.3, we now present the complete specialization structure, along with comprehensive attribute sets, for our Company system. The earlier section identified potential employee subclasses (MANAGER_EMPLOYEE, WAREHOUSE_EMPLOYEE, SALES_EMPLOYEE) with representative attributes to illustrate the specialization concept. Here, we extend that foundation with a complete implementation that includes:

- A fourth job function subclass (ADMIN_EMPLOYEE) to represent administrative staff
- Comprehensive specialized attribute sets for each subclass
- A second, independent specialization dimension based on employment type (FULL_TIME_EMPLOYEE, PART_TIME_EMPLOYEE, CONTRACT_EMPLOYEE)

Based on these two independent specialization criteria, we create two separate specialization hierarchies (Figures 5.4 and 5.5):

Job Function Specialization:

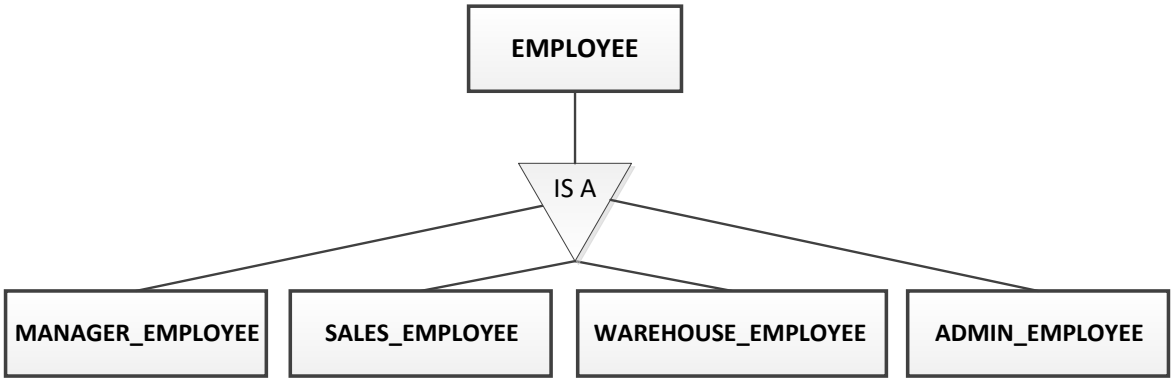


Figure 5.4: Employee Specialization by Job Function

Employment Type Specialization:

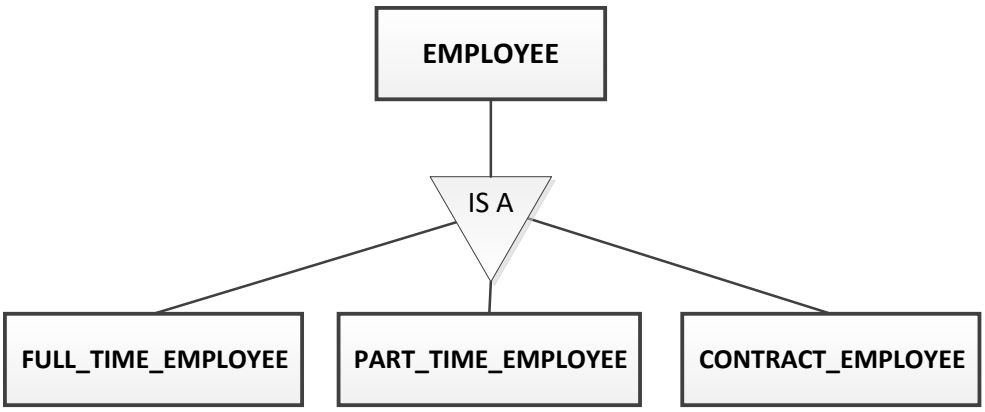


Figure 5.5: Employee Specialization by Employment Type

Please note that these are **two independent specializations** of the same EMPLOYEE superclass, not a hierarchical structure. An employee can simultaneously belong to one subclass from each specialization (e.g., E1 is both MANAGER_EMPLOYEE and FULL_TIME_EMPLOYEE).

Step 4: Define Specialized Attributes

Each subclass requires its own specialized attributes that capture its unique characteristics. Following the same principle established for students, attributes that vary by subclass type belong to the specialized entities.

Job Function Specialization Attributes:

MANAGER_EMPLOYEE Specialized Attributes:

- Manages_Department_ID (references DEPARTMENT and is already in base model as Manages_Dept_ID)
- Management_level (Department_Level, Division_Level, Executive_Level)
- Budget_authority (maximum budget approval amount)
- Number_of_direct_reports
- Leadership_training_completed (Boolean)
- Performance_bonus_percentage
- Strategic_planning_responsibility (Boolean)

Examples from our data:

- E1 (Zoran Pejovski): Manages D1 (Sales), Salary: 60000
- E3 (Marija Naumoska): Manages D2 (Marketing), Salary: 50000
- E5 (Naum Stojanoski): Manages D3 (HR), Salary: 40000

SALES_EMPLOYEE Specialized Attributes:

- Sales_territory (geographic region or account segment)
- Commission_rate (percentage of sales revenue)
- Monthly_sales_quota (target revenue)
- Year_to_date_sales (current performance)
- Customer_portfolio_size (number of active accounts)
- Territory_assignment_date

WAREHOUSE_EMPLOYEE Specialized Attributes:

- Primary_warehouse_ID (references WAREHOUSE and is already in base model as Warehouse_ID)
- Shift_schedule (Morning, Afternoon, Night)
- Forklift_certified (Boolean)
- Inventory_management_system_trained (Boolean)

- Physical_capacity_rating (weight handling certification level)
- Safety_training_current (Boolean)
- Preferred_warehouse_section (Receiving, Storage, Shipping)
- Physical_location (Skopje, Bitola, Ohrid, Tetovo)

Examples from our data:

- E2: Assigned to W1 (Skopje warehouse)
- E6 (Hristina Miloseska): Assigned to W3 (Ohrid warehouse)
- E7: Assigned to W2 (Bitola warehouse)

ADMIN_EMPLOYEE Specialized Attributes:

- Office_location (specific office or building)
- Administrative_level (Junior, Senior, Lead)
- Department_support_areas (multivalued attribute; which departments supported)
- Software_proficiencies (multivalued attribute: office applications, databases, etc.)
- Administrative_certifications (multivalued attribute)
- Document_management_responsibility (Boolean)

Employment Type Specialization Attributes:

FULL_TIME_EMPLOYEE Specialized Attributes:

- Annual_salary_amount (vs. hourly calculation)
- Benefits_package_level (Basic, Standard, Premium)
- Vacation_days_per_year (20, 25, or 30 days)
- Health_insurance_plan_type (individual, family coverage)
- Retirement_contribution_percentage (employer matching)
- Full_time_status_start_date

PART_TIME_EMPLOYEE Specialized Attributes:

- Hourly_rate
- Maximum_hours_per_week (contractual limit)
- Schedule_flexibility (Fixed, Flexible)
- Eligible_for_benefits (Boolean; based on hours worked)
- Average_weekly_hours (tracked for benefit calculation)
- Part_time_start_date

CONTRACT_EMPLOYEE Specialized Attributes:

- Contract_start_date
- Contract_end_date

- Contract_renewal_option (Boolean)
- Contracting_agency_or_independent (Enum: Agency, Independent)
- Payment_rate
- Contract_deliverables (text description)
- Original_contract_value

Attribute Placement Explanation:

Similar to the student specialization, attributes are placed in subclasses when their values differ by type:

- **Job-specific attributes** (like Sales_territory, Warehouse Shift_schedule) vary by job function → belong in job function subclasses
- **Employment-specific attributes** (like Benefits_package_level, Hourly_rate) vary by employment type → belong in employment type subclasses
- **Common attributes** (like Salary, Hire_date, Email) apply to all employees → remain in EMPLOYEE superclass

5.3.2 Generalization: Bottom-Up Approach

Generalization is the **inverse process** of specialization. It is a **bottom-up approach** that starts with subclass entities and identifies common attributes that can be abstracted into a superclass.

> **Definition: Generalization**
>
> Generalization is the process of creating a generalized entity type (superclass) from multiple existing entity types (subclasses) by suppressing their differences and identifying their common features. The original entity types become specialized subclasses of the new superclass (Elmasri & Navathe, 2016; Smith & Smith, 1977; Teorey et al., 1986).

5.3.2.1 Graphical Representation of Generalization

To indicate the bottom-up direction, generalization uses the ISA triangle symbol with the **opposite orientation** to that of specialization.

The ISA Triangle Symbol in Generalization

For generalization, the triangle is **oriented upward** (see Figure 5.6) to indicate the direction of the generalization process, from subclasses to the superclass. In the generalization bottom-up process, we start with individual entities (subclasses) and synthesize them into a general superclass.

Basic Generalization Notation Structure:

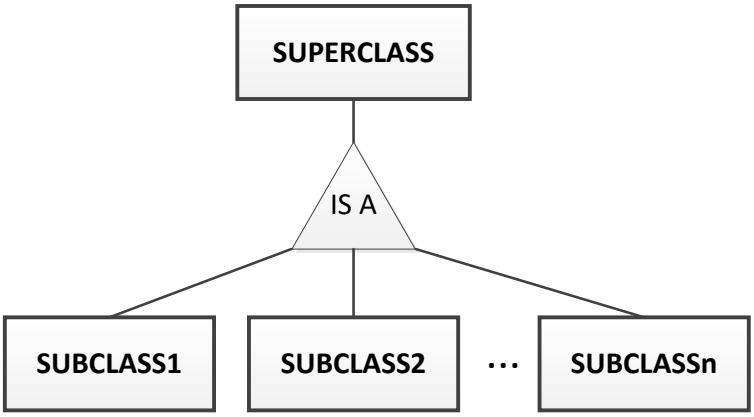


Figure 5.6: Generalization Notation

Interpretation

The upward ISA triangle orientation indicates that multiple specific entity types **are** all specialized instances of a more general concept. For example:

University Example:

- Every BACHELOR_STUDENT **IS A** STUDENT
- Every MASTERS_STUDENT **IS A** STUDENT
- Every DOCTORAL_STUDENT **IS A** STUDENT

Company Example:

- Every MANAGER_EMPLOYEE **IS AN** EMPLOYEE
- Every SALES_EMPLOYEE **IS AN** EMPLOYEE
- Every WAREHOUSE_EMPLOYEE **IS AN** EMPLOYEE
- Every ADMIN_EMPLOYEE **IS AN** EMPLOYEE

The meaning of the "IS A" relationship is identical to specialization; what is different is the modeling logic and direction.

5.3.2.2 The Generalization Process: University Example

To demonstrate the generalization process, we will first identify the distinct entity types during requirements analysis, then recognize their common characteristics (attributes).

Step 1: Identify Distinct Entity Types

Based on the description of the student categories, we discovered these distinct entity types and their characteristics:

- **BACHELOR_STUDENT**: Undergraduate-level students in three- or four-year programs, with major fields of study, and a two-member committee for bachelor's thesis defense
- **MASTERS_STUDENT**: Graduate-level students in one- or two-year programs, with research supervisors, thesis topics, and a three-member committee for master's thesis defense.

- **DOCTORAL_STUDENT**: Advanced research students in three-year doctoral programs, with dissertation committees and a five-member committee for PhD dissertation defense.

In the initial requirements analysis, stakeholders such as deans and administrators described these as distinct student categories with different characteristics and requirements. Initially, they are modeled as separate, distinct entities.

Step 2: Recognize Common Characteristics

Analyzing these entities, we identify shared attributes and relationships:

Common Attributes:

- Student_ID (unique identifier for all student types)
- Name (composite: First_name, Last_name)
- Birth_date and derived Age
- Email (multivalued for communication)
- Address (composite: Street, City, Zip_code, Country)
- ECTS_completed (credit accumulation applies to all)

Common Relationships:

- All enroll in courses (enrolls_in COURSE)
- All belong to faculties (belongs_to FACULTY)
- All attend class sessions (attends CLASS_SESSION)

Common Constraints:

- All follow university academic policies
- All have academic records
- All have enrollment dates

Step 3: Create Generalized Superclass

By extracting the common characteristics of the distinct entity types, we create a **STUDENT superclass** while the original entities become specialized subclasses (Figure 5.7):

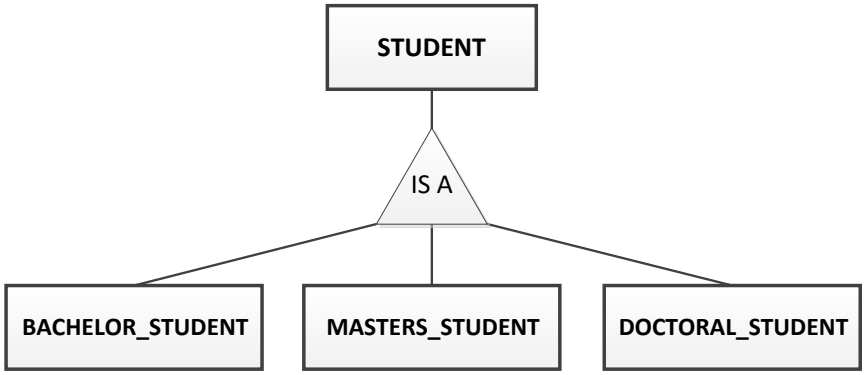


Figure 5.7: Student Generalization - Bottom-Up Approach

Step 4: Reorganize Attributes and Relationships

- **Move common attributes up** to STUDENT superclass: Student_ID, Name, Birth_date, Age, Email, Address, ECTS_completed
- **Keep specialized attributes** in respective subclasses: Major_field (bachelor), Master_thesis_topic (masters), Dissertation_topic (doctoral)
- **Establish inheritance relationships** so subclasses inherit all attributes and relationships from STUDENT
- **Result**: Identical structure to specialization (Figure 5.3), discovered through the opposite process

5.3.2.3 Company System Generalization Example

Step 1: Identify Distinct Entity Types

Based on the description of the employee categories, we discovered these distinct entity types and their characteristics:

- **MANAGER_EMPLOYEE**: Entities with budget authority, department supervision responsibilities, and strategic planning roles (E1, E3, E5 from our database in Chapter 11)
- **SALES_EMPLOYEE**: Entities with sales territories, commission structures, and quota achievement responsibilities (E2 from our database in Chapter 11)
- **WAREHOUSE_EMPLOYEE**: Entities with physical location assignments and inventory management responsibilities (E2, E6, E8 from our database in Chapter 11)
- **ADMIN_EMPLOYEE**: Entities with office locations and document management responsibilities

Note that E2 appears in both categories, demonstrating that employees can have multiple roles.

In the initial requirements analysis, managers described these as distinct categories using different systems and metrics. Initially, they are modeled as separate entities.

Step 2: Recognize Common Characteristics

All these entities share the common employment characteristics:

Common Employment Attributes:

- Employee_ID (unique identifier)
- Name (composite: First_name, Last_name)
- Email, Salary, Hire_date
- Phone (multivalued)
- Address (composite: Street, City, Zip_code, Country)

Common Organizational Relationships:

- All work for departments (works_for DEPARTMENT)

- All may have supervisors
- All processes or support business operations

Common Administrative Requirements:

- All subject to company policies and procedures
- All participate in payroll systems
- All have performance evaluations

Step 3: Create EMPLOYEE Superclass

By extracting the common characteristics of the distinct entity types, we create an **EMPLOYEE superclass** while the original entities become specialized subclasses (Figure 5.8):

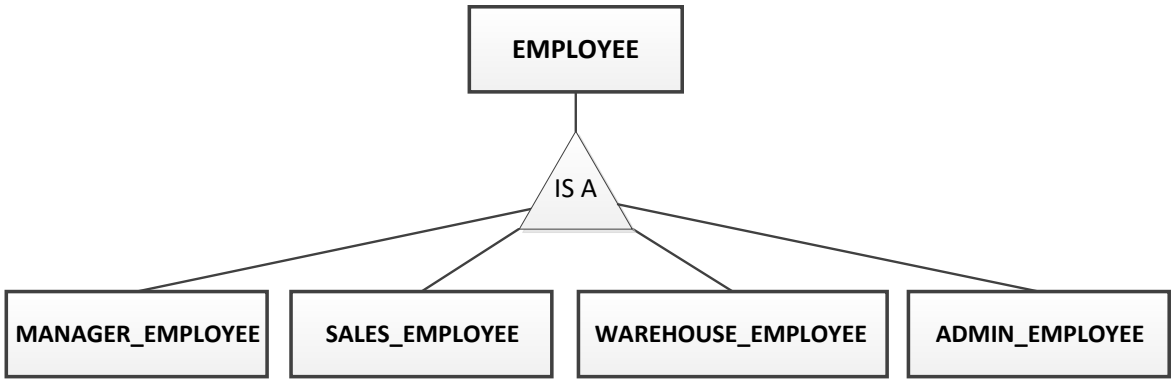


Figure 5.8: Employee Generalization by Job Function

Step 4: Reorganize Attributes and Relationships

- **Move common attributes up** to EMPLOYEE superclass: Employee_ID, Name, Email, Salary, Hire_date, Phone, Address
- **Keep specialized job-function attributes** in respective subclasses, for example, Management_level, Budget_authority (MANAGER_EMPLOYEE), Commission_rate, Sales_territory (SALES_EMPLOYEE), Shift_schedule, Physical_location (WAREHOUSE_EMPLOYEE), Office_location, Administrative_level (ADMIN_EMPLOYEE)
- **Establish inheritance relationships** so subclasses inherit all attributes and relationships from EMPLOYEE
- **Result**: Identical structure to specialization (Figure 5.4), discovered through the opposite process

5.3.2.4 Alternative Notation: The Unified Circle Symbol

Because specialization and generalization are two different representations of the same process and lead to the same visual result, **a unified graphical symbol** in the form of a **circle** (Figure 5.9) is increasingly used to represent both concepts (Hristoski & Dimovski, 2022).

This approach offers several advantages: the circle notation has no directional orientation, recognizing that specialization and generalization produce identical structures; the circle provides space for constraint symbols (d/o for disjoint/overlapping, covered in Section 5.4); and the subset symbol (⊆) on connecting lines shows that each subclass is a subset of the superclass.

Interpreting the Subset Symbol (⊆):

The symbol ⊆ can be understood in two complementary ways:

- **Union Perspective**: The superclass represents the **union** of all subclasses (using the ∪ set operation concept), meaning that the general type includes all specialized types
- **Subset Perspective**: Each subclass is a **subset** of the superclass (using the ⊆ set relation), meaning that each specialized entity type is a proper subset of the general entity type

Both interpretations are mathematically equivalent and support the same semantic relationship.

The Circle Notation:

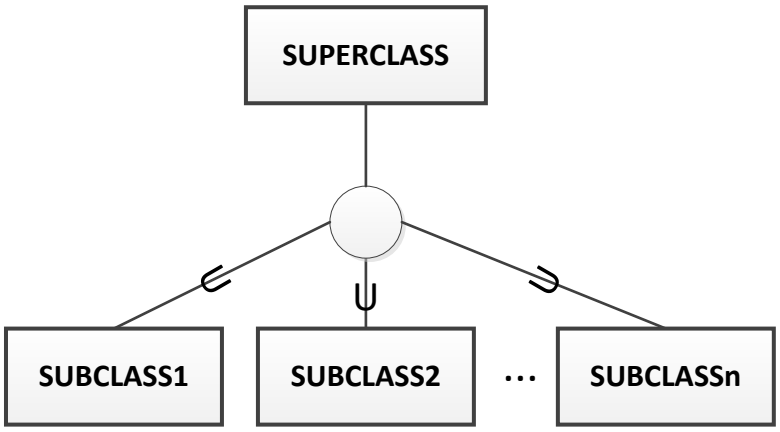


Figure 5.9: Unified Circle Notation for Specialization/Generalization

5.3.3 Specialization vs. Generalization: Understanding the Relationship

Specialization and generalization are equivalent processes that produce **identical EER structures**. The difference is not in the final result, but in the direction of the discovery and modeling process.

To illustrate this, let's consider our University student hierarchy:

- **Using Specialization**: "We have a STUDENT entity, and we need to specialize it into bachelor's, master's, and doctoral types to meet Bologna system requirements."
- **Using Generalization**: "We've identified bachelor's, master's, and doctoral student types, but they share some characteristics that we should generalize them under a STUDENT superclass."

Both approaches produce the same EER diagram shown in Figure 5.3, which includes a STUDENT superclass with common attributes, three subclasses with their own attributes,

inheritance "IS A" relationships, and identical constraint specifications (covered in Section 5.4).

Table 5.1 compares specialization and generalization.

Table 5.1: Process Characteristics Comparison

Aspect	*Specialization*	*Generalization*
Direction	Top-down	Bottom-up
Starting point	General entity type	Multiple separate entity types
Discovery process	Identify specialized subclasses within a general entity	Identify common attributes across separate entities
Design focus	What makes subclasses different from the superclass	What makes separate entities similar
Visual Notation	Downward-pointing triangle (▽)	Upward-pointing triangle (△)
Unified Notation	Circle (○) with subset symbols	Circle (○) with subset symbols
Design flow	ENTITY → subclasses	entities → SUPERCLASS
Attribute Movement	Push specialized attributes down to subclasses	Pull common attributes up to the superclass
When used	Extending existing ER models	Consolidating initially separate entity types

5.3.3.1 Design Process in Practice

***Specialization Example - University System*:**

Design Session Dialogue:

Database Analyst: "We have the STUDENT entity from Chapter 4. Now, let's extend it by examining the Bologna Process requirements..."

Academic Administrator: "Students have very different requirements depending on their academic level. Bachelor's students need 180 or 240 ECTS for three- or four-year programs, master's students need 60-120 ECTS for one- or two-year programs, and doctoral students have completely different structures with comprehensive exams and dissertations requiring five-member committees."

Database Analyst: "So our STUDENT entity needs specialization into these different academic levels, each with specific attributes and relationships."

Result: Top-down specialization from STUDENT to BACHELOR_STUDENT, MASTERS_STUDENT, DOCTORAL_STUDENT (Figure 5.3)

***Generalization Example - University System Alternative Discovery*:**

Design Session Alternative Dialogue:

Academic Administrator: "We track bachelor's students separately from master's and doctoral students because they are really different..."

Database Analyst: "Let me review what you've told me about each type: Bachelor's students have IDs, names, addresses, enrollment dates, belong to faculties, enroll in courses... Master's students have IDs, names, addresses, enrollment dates, belong to

faculties, enroll in courses... Doctoral students have IDs, names, addresses... They share a lot of attributes."

Academic Administrator: " You're right! They actually share many characteristics. But there are also important differences, like thesis requirements and ECTS structures."

Database Analyst: "Exactly. We should create a general STUDENT superclass capturing what they share, with specialized subclasses for what makes them different."

Result: Bottom-up generalization from BACHELOR_STUDENT, MASTERS_STUDENT, DOCTORAL_STUDENT to STUDENT superclass (Figure 5.7)

***Specialization Example - Company System*:**

Design Session Dialogue:

Database Analyst: "We have the EMPLOYEE entity from Chapter 4 (Figure 4.64). Looking at your organizational structure..."

HR Manager: "Our employees have very different roles. Managers have budget authority and strategic responsibilities. Sales employees have territories and commission structures. Warehouse employees work shifts and need safety certifications. Administrative staff supports multiple departments with various software tools."

Database Analyst: "So we need to specialize EMPLOYEE by job function, with each specialized type having unique attributes while inheriting common employment attributes."

Result: Top-down specialization from EMPLOYEE to MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, ADMIN_EMPLOYEE (Figure 5.4)

***Generalization Example - Company System Alternative Discovery*:**

Design Session Alternative Dialogue:

HR Manager: "We track managers in one system, sales representatives in another, warehouse workers in a third, administrative staff in a fourth..."

Database Analyst: "What information do you track for managers?"

HR Manager: "Employee ID, name, salary, hire date, department, email, phone, address... Plus their management level and budget authority."

Database Analyst: "And for sales representatives?"

HR Manager: "Employee ID, name, salary, hire date, department, email, phone, address... Plus their sales territory and commission rate."

Database Analyst: " And warehouse workers and administrative staff?"

HR Manager: " Employee ID, name, salary, hire date, contact details…but there are some job-specific attributes like shift assignments for warehouse workers and software access for administrative staff."

Database Analyst: "I notice there are many common attributes. All have basic employment information, such as ID, name, salary, hire date, and contact details. The

differences are in job-specific attributes. We will create an EMPLOYEE superclass with specialized subclasses for different job functions."

Result: Bottom-up generalization from separate MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, ADMIN_EMPLOYEE to EMPLOYEE superclass with job function subclasses (Figure 5.8)

Design Process in Practice Summary

University System:

- **Top-down**: STUDENT entity → is specialized into academic levels, OR
- **Bottom-up**: Separate bachelor/master/doctoral entities → generalize into STUDENT

Company System:

- **Top-down**: EMPLOYEE entity → is specialized into job functions, OR
- **Bottom-up**: Separate manager/sales/warehouse entities → generalize into EMPLOYEE

The resulting EER diagram structure remains exactly the same regardless of the discovery process. The choice of process depends on:

- **Your analysis starting point** (Did you identify a general concept first, or specific types first?)
- **How domain experts conceptualize** their domain (Top-down or bottom-up thinking?)
- **Your design context** (Refining existing design vs. consolidating separate systems?)

The final model structure doesn't depend on the chosen approach; both specialization and generalization produce the same EER diagram. That is why the circle symbol is used in the EER diagrams in Sections 5.4 and beyond: once the hierarchy is established, specialization and generalization are functionally and structurally identical.

5.3.4 Systematic EER Development Methodology

Extending the basic ER diagram to an EER diagram requires not only technical skills but also a deep understanding of the problem being modeled and a careful, methodical approach that accounts for the complexity of institutions and organizations.

5.3.4.1 Understanding the Requirements

For both ER and EER modeling, it is crucial to understand the requirements. The process of requirements gathering involves not only what stakeholders explicitly state but also the implicit meaning in their descriptions. The *implicit requirements* are usually very valuable, because they reflect how the stakeholders conceptualize the problem domain. For example, the academic administrator might explain "students enroll in courses" while simultaneously mentioning that "doctoral students have a dissertation committee of five members". Database designers must identify the general requirements that apply to all entity types and can be placed in the superclass, as well as those that are specific to subclasses.

Looking at our University data, we observed that each student has the same attributes, such as Student_ID, Name, and Address. However, students also participate in specialized academic structures with specific attributes. This indicates that we can't model this complexity with a basic ER diagram; we need EER modeling instead.

5.3.4.2 Building the EER Diagram Incrementally

The transition from the basic ER diagram to the EER diagram is not a single-step process. Our EER model is based on the ER model from Chapter 4 and is built incrementally by identifying specialized subclasses through specialization or generalization. In each step, we validate our choices before introducing additional elements.

Consider how we developed the University student hierarchy. We started with the STUDENT entity from the ER diagram in Chapter 4, initially focusing on the distinct academic levels required by the Bologna Process (European Commission, 2015). The result was the creation of subclasses such as BACHELOR_STUDENT, MASTERS_STUDENT, and DOCTORAL_STUDENT, using either a top-down (specialization) or a bottom-up (generalization) approach. Once validated, this structure can be further extended with additional components, such as distinguishing between one-year and two-year master's programs as separate subclasses, if needed.

This incremental approach allows us to **validate each modeling decision** before proceeding with the next step. For example, we tested our new specialization/generalization against questions like: "Can Sarah Smith (S001) simultaneously be both a bachelor's student and a master's student?" and "Do the specific attributes we've assigned to BACHELOR_STUDENT make sense for every bachelor's student in our data?" This helps ensure that our solution accurately reflects the requirements.

We applied the same incremental methodology to our Company system. First, we established job-function specialization (MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, ADMIN_EMPLOYEE). After validating it against our employee data, we introduced employment type specializations (FULL_TIME_EMPLOYEE, PART_TIME_EMPLOYEE, CONTRACT_EMPLOYEE).

5.3.4.3 Validating the Model

Model validation extends beyond technical correctness and should align with how the organization or institution being modeled actually operates.

For example, we need to confirm with the university administration whether the size of the defense committee varies by academic level (two members for bachelor's, three for master's, five for doctoral) before we assign the Defense_committee attribute to specialized student subclasses rather than to the STUDENT superclass. Similarly, when we placed the Sales_territory attribute in SALES_EMPLOYEE rather than in the general EMPLOYEE entity, we verified that territory assignments apply only to sales roles, not to managers, warehouse workers, or administrative staff.

Constraint specifications also need careful validation. When we model the student academic level specialization as **disjoint** (a student belongs to exactly one academic level at a time), this means that students cannot simultaneously pursue bachelor's and master's

degrees under the Bologna Process structure. Conversely, when we model employee job-function specialization as **partial** (not every employee belongs to the specialized categories included in the model), we recognize that companies employ many people in general roles that don't fit into the specialized subclasses we've defined. This is explained in more detail in Section 5.4.

To verify that our EER model captures realistic entity evolution, we validate it against real entity instances and their life cycles. Consider how Sarah Smith (S001) progresses through the university system. She begins as a BACHELOR_STUDENT with 0 ECTS_completed and a Total_ECTS_required value of 240. As she completes courses, her ECTS_completed increases. Upon graduation with 240+ credits, she leaves the BACHELOR_STUDENT subclass. If she later returns for graduate study, she is entered into the system as a new MASTERS_STUDENT instance. Similarly, we can validate our dual-specialization approach by examining the employee E1 (Zoran Pejovski) in our Company system. He simultaneously belongs to MANAGER_EMPLOYEE (by job function, managing Sales department D1) and FULL_TIME_EMPLOYEE (by employment type, with annual salary and benefits).

5.3.4.4 Balancing Semantic Richness and Implementation Efficiency

Creating EER diagrams with many levels of subclasses can accurately represent organizational structures, but can also increase system complexity and require more complex queries. When designing the model, we must balance semantic expressiveness with implementation efficiency.

Consider our University system, with three possible levels (PERSON → STUDENT → academic level subclasses). This structure is sufficiently descriptive while maintaining low implementation complexity. Adding an additional layer, such as specializing BACHELOR_STUDENT into subclasses for first-year, second-year, third-year, and fourth-year, would complicate the model and add little semantic value. The Study_year attribute provides this information more efficiently than additional subclass levels.

The number of subclasses also needs careful consideration. For example, we modeled our employee job function specialization with four subclasses (MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, ADMIN_EMPLOYEE). We could potentially add specialized subclasses for accountants, IT professionals, and other roles. Doing so would complicate the model. We need to carefully consider whether additional subclasses add value to the model or whether the information can be captured more simply.

Multiple independent specializations provide powerful modeling capabilities but increase implementation complexity. For example, we modeled our Company system using both job-function and employment-type specializations. For each employee, there are twelve possible combinations (four job functions times three employment types), though not all may exist in practice. This complexity is justified when both dimensions require distinct, specialized attributes, but simpler alternatives should be considered when the information can be captured through attributes of existing entity types rather than through additional specialization hierarchies.

5.3.4.5 Methodology Principles

Through our University and Company case studies, we identified a few principles for effective EER development:

1. **Start simple, add complexity only when justified**: Begin with basic ER structures and introduce specialization/generalization only when flat entities cannot adequately capture the requirements.
2. **Validate incrementally at each level**: Test each hierarchy level against real data and business scenarios before adding additional complexity. Sarah Smith's (S001) progression through bachelor's studies and E1's dual role as both a manager and a full-time employee validate our modeling decisions.
3. **Balance semantic precision with implementation efficiency**: Every hierarchy level and specialized relationship must provide sufficient value to justify its implementation complexity. The Study_year attribute proves more efficient than multiple subclass levels for tracking student progression.
4. **Ensure constraints reflect actual business rules**: Disjoint/overlapping and total/partial constraints must match organizational policies. University policies determine whether students can pursue multiple degrees simultaneously; company practices determine whether all employees must fit specialized categories.

In Section 5.4, we'll explore the constraint mechanisms that govern these structures. We will examine how disjoint/overlapping and total/partial constraints specify the business rules that determine subclass membership and specialization behavior.

5.4 Constraints on Specialization and Generalization

Having understood the processes of specialization and generalization through our University and Company case studies, we now need to examine the constraint mechanisms that govern how entities participate in these hierarchical structures. These constraints ensure that our EER models accurately reflect organizational reality while maintaining logical consistency. They are used to determine whether students can pursue multiple degrees simultaneously, whether employees must fit into specialized categories, and how organizational policies translate into formal database structures.

5.4.1 The Two Constraint Dimensions

EER specialization and generalization structures have **two independent dimensions of constraints** that define entity classification rules.

Disjointness Constraints define whether an entity can belong to multiple subclasses simultaneously. When Sarah Smith (S001) completes her bachelor's degree, can she remain a BACHELOR_STUDENT while also becoming a MASTERS_STUDENT, or must her classification change from one to the other? The disjointness constraint answers this question.

Completeness Constraints define whether every superclass entity must belong to at least one subclass. Must every employee in our Company system fit into specialized

categories (MANAGER_EMPLOYEE, SALES_EMPLOYEE, etc.), or can some employees exist as general employees without specialized classification? The completeness constraint determines this requirement.

> **Constraint Dimensions**
>
> Disjointness and completeness constraints are independent dimensions. They can be combined in any way that reflects business requirements. This independence enables us to model four distinct organizational patterns, each capturing different classification rules. (Elmasri & Navathe, 2016; Teorey et al., 1986)

These two dimensions combine to create four possible constraint combinations: **disjoint-total**, **disjoint-partial**, **overlapping-total**, and **overlapping-partial**. Each combination captures different classification patterns.

5.4.2 Disjointness Constraints

The disjointness constraint addresses a fundamental question: can a single entity instance belong to multiple subclasses simultaneously, or must membership be mutually exclusive? This constraint directly impacts our model.

5.4.2.1 Disjoint Specialization

A specialization is **disjoint** (mutually exclusive) if an entity in the superclass can belong to **at most one** subclass in the specialization.

Formal Definition: A specialization Z = {S_1, S_2, ..., S_n} is disjoint if:

$S_i \cap S_j = \emptyset$ (empty set) for all $i \neq j$

This means that the subclasses have no members in common; no entity can exist in multiple subclasses simultaneously (Elmasri & Navathe, 2016).

University System Example - Disjoint Specialization:

In our University system, student classification by academic level is disjoint:

STUDENT is specialized into {BACHELOR_STUDENT, MASTERS_STUDENT, DOCTORAL_STUDENT}

Rule: A student cannot be enrolled as both a bachelor's and a master's student at the same time under the Bologna Process framework. When Sarah Smith (S001) completes her bachelor's degree with 240 ECTS credits, she must graduate from BACHELOR_STUDENT status before she can be admitted to a master's program as a MASTERS_STUDENT. At any given moment, she belongs to exactly one academic level.

Company System Example - Disjoint Specialization:

Employee classification by primary job function demonstrates disjoint specialization in organizational contexts:

EMPLOYEE is specialized into {MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, ADMIN_EMPLOYEE}

Rule: Each employee has one primary job classification. Employee E1 (Zoran Pejovski) works for the Sales department (D1) and manages the Sales department (D1), but his primary classification is MANAGER_EMPLOYEE, not SALES_EMPLOYEE.

Although E1 manages a sales department and may perform sales-related activities, his primary classification is as a manager. In organizations, employees typically have one primary job classification, even when their responsibilities span multiple functional areas. The disjoint constraint captures this "primary classification" rule that organizations commonly apply.

5.4.2.2 Overlapping Specialization

A specialization is **overlapping** (non-disjoint) if an entity in the superclass can belong to **multiple subclasses simultaneously**.

Formal Definition: A specialization $Z = \{S_1, S_2, ..., S_n\}$ is overlapping if:

$S_i \cap S_j \neq \emptyset$ (may have common members) for some $i \neq j$

This means that subclasses may share members; entities can exist in multiple subclasses simultaneously (Elmasri & Navathe, 2016).

***University System Example - Overlapping Specialization*:**

To illustrate the concept of overlapping specialization in our University system, we will introduce the PERSON superclass that is specialized into five subclasses: STUDENT, PROFESSOR, INSTRUCTOR, STAFF, and ALUMNUS.

PERSON is specialized into {STUDENT, PROFESSOR, INSTRUCTOR, STAFF, ALUMNUS}

Rule: A person can have multiple roles simultaneously within the university. This overlapping classification reflects the complex relationships universities maintain with people:

- A doctoral student may teach undergraduate courses as an INSTRUCTOR while enrolled as a STUDENT (receiving teaching compensation while pursuing a degree)
- A professor who graduated from the university holds roles as both PROFESSOR (current employment) and ALUMNUS (past enrollment)
- A staff member taking evening courses for professional development appears as both STAFF (employment relationship) and STUDENT (enrollment relationship)
- A graduate student working as a teaching assistant functions as both STUDENT (pursuing a degree) and INSTRUCTOR (teaching courses)

In this example, the overlapping constraint becomes necessary because these roles are **not mutually exclusive**.

Each role has specific relationships and attributes. As a STUDENT, the PERSON enrolls in courses, accumulates ECTS credits, and progresses toward a degree. As a PROFESSOR, the PERSON teaches courses, receives a salary, and participates in faculty governance. These are complementary roles held simultaneously by the same person.

The INSTRUCTOR subclass represents people who teach courses but are not full faculty members (professors). This includes:

- Teaching assistants (graduate students who teach undergraduate sections)
- Lecturers (non-tenure-track teaching positions)
- Adjunct instructors (part-time teaching staff)
- Lab instructors and tutorial leaders

INSTRUCTOR attributes include Course_taught, Teaching_schedule, and Student_evaluation_average.

***Company System Example - Overlapping Specialization*:**

To illustrate the concept of overlapping specialization in our Company system, we introduce a different specialization approach based on professional certifications. The EMPLOYEE superclass is specialized into three subclasses:

EMPLOYEE is specialized into {CERTIFIED_PROGRAMMER, CERTIFIED_ACCOUNTANT, CERTIFIED_PROJECT_MANAGER}

Rule: Employees can hold multiple professional certifications simultaneously, qualifying them for diverse responsibilities. For example, an employee might be both a certified programmer and a certified project manager. This overlapping structure reflects a common pattern in modern organizations, where employees have multiple skills.

Note: This certification-based specialization is independent from the job-function specialization (MANAGER_EMPLOYEE, SALES_EMPLOYEE, etc.) discussed earlier. The same employee can have both a job function classification and multiple certification classifications.

5.4.3 Completeness Constraints

The completeness constraint addresses whether every superclass entity must belong to at least one subclass, or whether some entities can exist without specialized classification.

There are two types of completeness constraints: **total** and **partial**.

5.4.3.1 Total Specialization

A specialization is **total** if every entity in the superclass **must** belong to at least one subclass.

Formal Definition: A specialization Z = {S_1, S_2, ..., S_n} with superclass G is total if:

$S_1 \cup S_2 \cup ... \cup S_n = G$

Every superclass entity appears in at least one subclass; there are no "unclassified" entities (Elmasri & Navathe, 2016).

***University System Example - Total Specialization*:**

In our University system, students' academic level specialization demonstrates total constraint:

STUDENT is specialized into {BACHELOR_STUDENT, MASTERS_STUDENT, DOCTORAL_STUDENT}

Rule: According to the Bologna Process (European Commission, 2015), each student must enroll in one of these degree levels. This ensures that no students have an "unclassified" degree level; all students must belong to a specific academic level.

When students like S001 (Sarah Smith), S002 (Bojan Jovansen), or S003 (Kosta Dojcinoski) appear in our STUDENT entity, they must also appear in exactly one of the academic level subclasses. There is no general “student” status, so all applicants must specify whether they are pursuing a bachelor's, master's, or doctoral program. A student's progress makes sense only in the context of the specific requirements of their degree level.

The total constraint captures this university policy: every student must have a degree-level classification.

5.4.3.2 Partial Specialization

A specialization is **partial** if some entities in the superclass **may not** belong to any subclass.

Formal Definition: A specialization Z = {S_1, S_2, ..., S_n} with superclass G is partial if:

$S_1 \cup S_2 \cup ... \cup S_n \subset G$ (proper subset)

Some superclass entities don't appear in any subclass; entities can remain "general" or "unclassified" (Elmasri & Navathe, 2016).

***Company System Example - Partial Specialization*:**

In our Company system, employee job function specialization demonstrates partial constraint:

EMPLOYEE is specialized into {MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, ADMIN_EMPLOYEE}

Rule: Not all employees have specialized roles requiring distinct classification. There are employees who don’t belong to any specialized category; they perform general duties. In our Company data, employees E4, E6, E7, and E8 work in their assigned departments without management responsibilities and without specialized sales, warehouse, or administrative classifications. These general employees perform functions that do not warrant specialized subclass modeling.

Employee E4 (Anita Toshevska), for instance, works in the Marketing department (D2) as a general employee without specialized classification. The partial constraint permits this.

5.4.4 Combined Constraint Scenarios

The disjointness and completeness constraint dimensions combine to create four distinct scenarios, each capturing different organizational classification patterns. Understanding how these constraints combine is essential for creating accurate system models.

5.4.4.1 Scenario 1: Disjoint, Total (d, t)

Definition: Every superclass entity belongs to **exactly one** subclass.

***University Example*:**

To demonstrate this scenario, let's consider student academic level specialization:

STUDENT is specialized into {BACHELOR_STUDENT, MASTERS_STUDENT, DOCTORAL_STUDENT}

The constraint for this scenario is ***disjoint and total (d, t)***.

Explanation: Every student belongs to exactly one academic level. There are no students without a degree-level classification (total), and no students simultaneously pursue multiple Bologna-defined degree levels (disjoint). For example, student Sarah Smith (S001) is enrolled in one and only one degree program at any given time.

Justification: The Bologna Process (European Commission, 2015) defines degree levels as distinct educational stages rather than overlapping categories. The university's structure requires exclusive degree-level classification because credit requirements, thesis expectations, tuition structures, and graduation criteria all depend on mutually exclusive program classifications.

5.4.4.2 Scenario 2: Disjoint, Partial (d, p)

Definition: Each superclass entity belongs to **at most one** subclass; some may belong to **none**.

***Company Example*:**

To demonstrate this scenario, let's consider employee job function specialization:

EMPLOYEE is specialized into {MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, ADMIN_EMPLOYEE}

The constraint for this scenario is ***disjoint and partial (d, p)***.

Explanation: An employee can have at most one specialized classification (disjoint). For example, an employee cannot simultaneously be classified as both a manager and a sales employee, because they are mutually exclusive categories. However, many employees in our data, including E4, E6, E7, and E8, are general employees without any specialized classification (partial).

Justification: This reflects a typical organizational structure, combining exclusive specialized roles with the recognition that specialization is not universal. Organizational structure accommodates both specialized roles with distinct responsibilities and general positions that do not fit predefined categories. General employees who do not belong to any specialized category perform valuable work without requiring specialized classification.

5.4.4.3 Scenario 3: Overlapping, Total (o, t)

Definition: Every superclass entity belongs to **at least one** subclass; entities can belong to **multiple** subclasses.

***University Example*:**

To demonstrate this scenario, let's consider person specialization:

PERSON is specialized into {STUDENT, PROFESSOR, STAFF, ALUMNUS, INSTRUCTOR}

The constraint for this scenario is ***overlapping and total (o, t)***.

Explanation: Every person recorded in the university system must have at least one relationship with the institution (total). However, people often hold multiple overlapping relationships: doctoral students teach (STUDENT + INSTRUCTOR), professors who graduated from the university maintain both employment and alumni relationships (PROFESSOR + ALUMNUS), and staff members pursue professional development courses (STAFF + STUDENT).

Justification: The total constraint ensures that the university tracks only people with actual institutional relationships, while the overlapping constraint recognizes that these relationships are not mutually exclusive. A doctoral student teaching undergraduate courses functions in both roles, receiving student services and faculty compensation, with distinct business processes for each.

5.4.4.4 Scenario 4: Overlapping, Partial (o, p)

Definition: Superclass entities can belong to **any combination** of subclasses, including **none**.

***Company Example*:**

To demonstrate this scenario, let's consider employee organizational involvement specialization:

EMPLOYEE is specialized into {PROJECT_TEAM_MEMBER, COMMITTEE_MEMBER, MENTOR}

The constraint for this scenario is ***overlapping and partial (o, p)***.

Explanation: Employees can participate in multiple activities simultaneously (overlapping): Employee E1 might serve on project teams and organizational committees while also mentoring junior staff. However, not all employees participate in these activities (partial). For example, employee E8 might not participate in any projects, committees, or mentoring relationships, remaining a general employee focused solely on regular departmental duties.

Justification: These specializations represent assignment-based or voluntary roles that vary by employees' interests, skills, and organizational needs. The overlapping constraint reflects that participation in multiple activities is encouraged in organizations. The partial constraint recognizes that these are not mandatory classifications; some employees do not have these additional roles. This combination provides maximum flexibility while accurately modeling how organizations actually structure optional involvement.

5.4.5 Determining Correct Constraints

Determining correct constraints requires systematic analysis of business rules and organizational policies. We must examine actual entity instances, interview domain experts, and validate our constraint decisions against realistic scenarios.

Analytical Questions for Disjointness

To determine whether specialization should be disjoint or overlapping, we examine these questions with domain experts:

1. **Can an entity logically belong to multiple subclasses simultaneously?**
   - If NO → Disjoint (student cannot be both bachelor's and master's simultaneously)
   - If YES → Overlapping (person can be both student and professor simultaneously)
2. **Do subclasses represent mutually exclusive states or classifications?**
   - If YES → Disjoint (academic levels are mutually exclusive stages)
   - If NO → Overlapping (professional certifications are complementary credentials)
3. **Would belonging to multiple subclasses create business rule conflicts?**
   - If YES → Disjoint (conflicting credit requirements or thesis expectations)
   - If NO → Overlapping (compatible role responsibilities)
4. **Do organizational processes treat subclass membership as exclusive categories?**
   - If YES → Disjoint (payroll assigns one primary job classification)
   - If NO → Overlapping (multiple certifications tracked independently)

Analytical Questions for Completeness

To determine whether specialization should be total or partial, we examine:

1. **Must every superclass entity have a specialized classification?**
   - If YES → Total (every student must have a degree level)
   - If NO → Partial (some employees don't have specialized roles)
2. **Are there valid "general" or "unclassified" entities in organizational operations?**
   - If YES → Partial (general employees without specialized classification)
   - If NO → Total (no unclassified students under Bologna Process)
3. **Do business processes require the classification of all entities?**
   - If YES → Total (credit tracking requires degree level for all students)
   - If NO → Partial (many employees function effectively without specialization)
4. **Would forcing classification on all entities create artificial categories?**
   - If YES → Partial (avoid forcing employees into inappropriate specializations)
   - If NO → Total (natural requirement that all students pursue specific degrees)

Tables 5.2 and 5.3 illustrate constraint analysis for the University and Company systems, respectively.

University System Constraint Analysis

Table 5.2: University System Constraint Analysis

Specialization	*Disjoint?*	*Total?*	*Constraint*	*Justification*
STUDENT by academic level	Yes	Yes	(d, t)	Exclusive Bologna Process degree programs; all students enrolled in specific programs with defined requirements
PERSON by university role	No	Yes	(o, t)	Multiple concurrent roles are possible (doctoral students teaching); all people in the system have institutional relationships

Company System Constraint Analysis

Table 5.3: Company System Constraint Analysis

Specialization	*Disjoint?*	*Total?*	*Constraint*	*Justification*
EMPLOYEE by job function	Yes	No	(d, p)	Primary job classification is exclusive; many general employees without specialized roles
EMPLOYEE by certifications	No	No	(o, p)	Multiple professional certifications are possible; not all employees hold certifications

These analytical frameworks and constraint determinations reflect actual business rules rather than convenience in modeling. When we specify (d, t) for student academic levels, this captures genuine university policy requiring exclusive degree-level enrollment. When we specify (d, p) for employee job functions, this reflects the organizational reality that primary roles are exclusive, but specialization is not universal. The constraints ensure that our EER models accurately reflect how organizations actually operate.

5.4.6 Constraint Notation Preview

Understanding how constraints are visually represented in EER diagrams enables us to read and create precise models. While we examine formal Chen notation comprehensively in Section 5.6, a preview here connects our conceptual constraint understanding to its visual representation:

Visual Notation Summary:

- **Disjoint (d)**: Shown with lowercase "d" symbol in the specialization circle
- **Overlapping (o)**: Shown with lowercase "o" symbol in the specialization circle
- **Total**: Shown with a **double line** from the superclass to the specialization structure
- **Partial**: Shown with a **single line** from the superclass to the specialization structure

These symbols combine to create comprehensive constraint specifications. A disjoint, total specialization appears with "d" in the circle, and a double line from the superclass indicates that every superclass entity belongs to exactly one subclass. An overlapping, partial specialization appears with "o" in the circle, and a single line indicates that entities may belong to any combination of subclasses, including none.

In Section 5.6, we'll examine these notational elements in detail, showing how to construct complete EER diagrams with proper constraint specification for our University and Company case studies.

In the next section, we'll explore hierarchies and lattices, the structural patterns that emerge when specializations extend across multiple levels or when entities inherit from multiple superclasses.

5.5 Specialization Hierarchies and Lattices

In more complex EER models, entities often participate in multiple levels of specialization, creating structures that extend beyond the simple two-level hierarchies we have discussed so far. These complex structures can range from clean tree-like hierarchies, in which each subclass has exactly one parent, to complex lattices, in which subclasses inherit from multiple superclasses simultaneously (Elmasri & Navathe, 2016).

Hierarchies and **lattices** represent two different classification approaches. In hierarchies, entities have a single primary categorization path and form clear inheritance chains. Lattices recognize that some entities naturally combine multiple classifications simultaneously, requiring multiple inheritance to capture their true nature. Careful analysis of organizational practices is needed to choose between these modeling approaches. The following subsections examine each structure in detail.

5.5.1 Specialization Hierarchies

> **Definition: Specialization Hierarchy**
>
> A specialization hierarchy is a tree-like structure in which each subclass participates in only one class/subclass relationship. Each subclass has exactly one parent (superclass), forming inheritance paths from the root to the leaves. No subclass appears in multiple branches of the tree, ensuring single inheritance throughout the structure. (Elmasri & Navathe, 2016)

Hierarchies provide a simple, clearly defined parent-child structure. When we examine Sarah Smith (S001) as a BACHELOR_STUDENT in Computer Science, the hierarchy provides a straightforward classification path: she is a PERSON (root concept), a STUDENT (first-level specialization), and specifically a BACHELOR_STUDENT (second-level specialization). This reflects how universities actually conceptualize students by specializing general concepts into specific classifications.

5.5.1.1 Characteristics of Hierarchical Structures

Hierarchies have several characteristics that distinguish them from lattice structures:

Single Inheritance: Each subclass inherits from exactly one immediate superclass. For example, BACHELOR_STUDENT inherits only from STUDENT. While BACHELOR_STUDENT also inherits indirectly from PERSON (through STUDENT), the direct inheritance relationship remains singular and straightforward.

Tree Structure: No subclass appears in multiple branches. A BACHELOR_STUDENT exists in the STUDENT branch and cannot simultaneously appear in the PROFESSOR branch. Entities follow a single path through the classification hierarchy, creating the tree structure.

Clear Parent-Child Relationships: The inheritance path from any subclass to the root superclass is unique and straightforward. Tracing FIRST_YEAR_BACHELOR's attributes follows exactly one path: through BACHELOR_STUDENT, then STUDENT, and finally to PERSON. There are no alternative paths, which eliminates inheritance ambiguity.

Simpler Implementation: The tree structure maps straightforwardly to database schemas. Implementation strategies remain conceptually simple because single inheritance eliminates the need for attribute conflict resolution (Elmasri & Navathe, 2016).

University System Hierarchy Example

Our University system creates a hierarchical structure reflecting the Bologna Process's academic classifications (European Commission, 2015). Recall from Section 5.4.2.2 that the complete PERSON specialization includes five subclasses: STUDENT, PROFESSOR, INSTRUCTOR, STAFF, and ALUMNUS. To illustrate multi-level hierarchies, Figure 5.10 focuses on two branches of the PERSON specialization, STUDENT and PROFESSOR, and extends each with additional hierarchical levels. For clarity, the diagram shows only entity names and hierarchical relationships; the specialized attributes of the superclass and each subclass are described in the accompanying text below.

The hierarchy in Figure 5.10 reflects how universities naturally organize people. At the root, PERSON contains attributes universal to all people the university tracks, such as SSN, Name, Birth_date, and Address. The first-level specialization separates people by their primary university role to capture the distinction between those pursuing degrees (STUDENT) and those employed to teach and conduct research (PROFESSOR).

The second level refines these primary roles. STUDENT is specialized into BACHELOR_STUDENT, MASTERS_STUDENT, and DOCTORAL_STUDENT. PROFESSOR is specialized into ASSISTANT_PROFESSOR, ASSOCIATE_PROFESSOR, and FULL_PROFESSOR based on academic rank and seniority. Each second-level specialization adds attributes specific to that classification: BACHELOR_STUDENT includes Major_field, Study_year, and Total_ECTS_required (180 or 240), while ASSISTANT_PROFESSOR includes Employment_type and Years_in_current_rank.

The third level provides even finer classification granularity. BACHELOR_STUDENT is specialized into year-based subclasses (FIRST_YEAR_BACHELOR through FOURTH_YEAR_BACHELOR), each potentially adding year-specific attributes, such as Student_ID_card_issued, Enrollment_status_verified, and Course_registration_completed for first-year students or Graduation_application_submitted, Thesis_proposal_approved, and Required_credits_completed for fourth-year students. MASTERS_STUDENT is

specialized into ONE_YEAR_MASTERS and TWO_YEAR_MASTERS, distinguishing between 60-ECTS and 120-ECTS programs. DOCTORAL_STUDENT is specialized by program phase (COURSEWORK_PHASE_PHD, QUALIFYING_PHASE_PHD, DISSERTATION_PHASE_PHD), capturing the distinct milestones in doctoral progression.

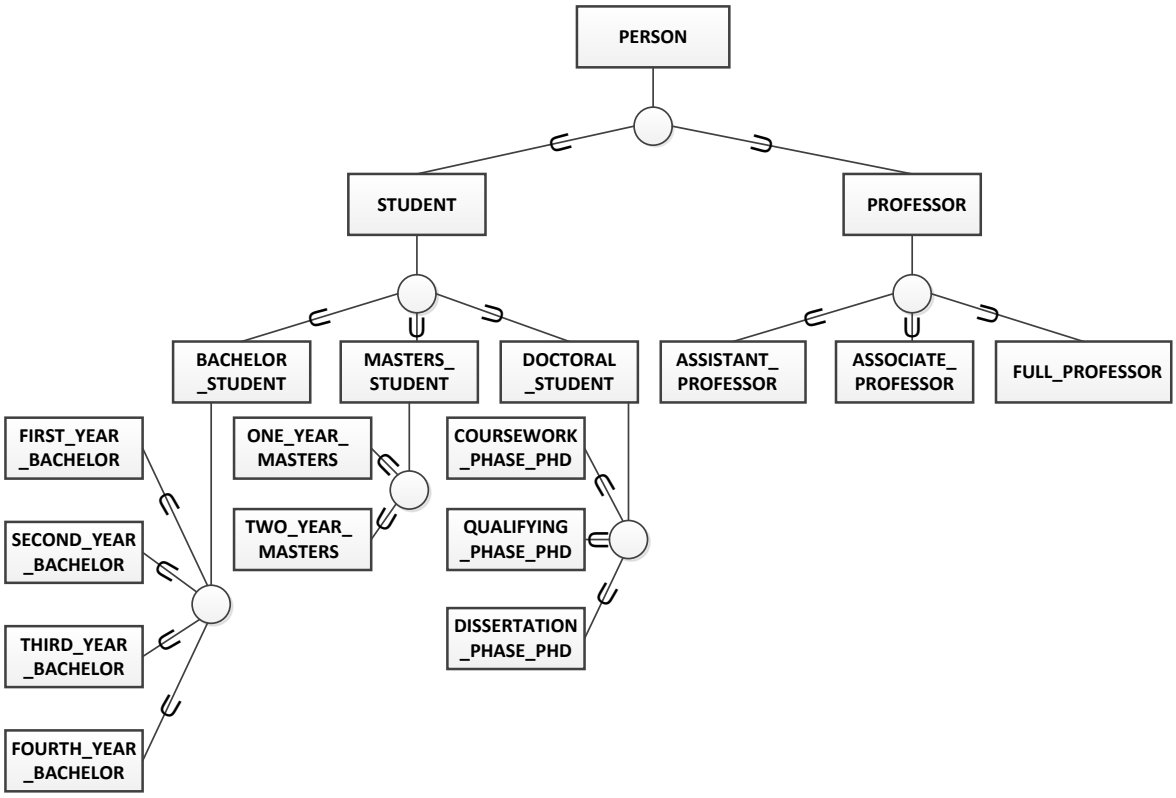


Figure 5.10: University System Hierarchy Example

Attribute Inheritance Through Multiple Levels

Consider how a FIRST_YEAR_BACHELOR entity like Sarah Smith (S001) inherits attributes through the hierarchy. She accumulates attributes from every level of her classification path:

From PERSON (root):

- SSN (unique person identifier)
- Name (composite: First_name, Last_name)
- Birth_date and derived Age
- Address (composite: Street, City, Zip_code, Country)
- Email

From STUDENT (first level):

- Student_ID (university student identifier)
- Enrollment_date
- ECTS_completed

- Faculty_affiliation (belongs_to relationship with FACULTY entity)
- Inherits all PERSON attributes plus adds student-specific attributes

From BACHELOR_STUDENT (second level):

- Major_field (Computer Science for Sarah)
- Study_year (first, second, third, fourth)
- Total_ECTS_required (240 for her 4-year program)
- Bachelor_thesis_supervisor (assigned professor)
- Expected_graduation_date
- Defense_committee (2 members)
- Inherits all STUDENT and PERSON attributes plus bachelor-specific attributes

From FIRST_YEAR_BACHELOR (third level):

- Student_ID_card_issued (received official student ID)
- Enrollment_status_verified (administrative confirmation completed)
- Course_registration_completed (registered for first semester)
- Inherits all BACHELOR_STUDENT, STUDENT, and PERSON attributes plus first-year-specific attributes

This cumulative inheritance demonstrates how the hierarchical structure works. Sarah's complete attribute set reflects her position in all levels of the classification tree. She possesses PERSON attributes, STUDENT attributes, BACHELOR_STUDENT attributes, and FIRST_YEAR_BACHELOR attributes, which together form a complete profile assembled through single inheritance along the hierarchical path.

Company System Hierarchy Example

Our Company system also arranges employees in a hierarchical structure, although the classification criteria differ from those used in academic contexts. To illustrate multi-level hierarchies within a company, Figure 5.11 presents an alternative employee classification structure. This example differs from the job function specialization (MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, ADMIN_EMPLOYEE) discussed in earlier sections; here, we demonstrate hierarchical depth through broad role categories that further specialize by experience level and management scope. For clarity, the diagram shows only entity names and hierarchical relationships; the specialized attributes are described below.

The organizational hierarchy in Figure 5.11 illustrates a typical company structure. At the root, PERSON contains universal attributes such as Name (First_name, Last_name), Email, and Address (Street, City, Country, Zip). EMPLOYEE adds employment-related attributes, including Employee_ID, Hire_date, Salary, and Department_ID.

The second level is specialized into broad job categories: MANAGEMENT_EMPLOYEE, TECHNICAL_EMPLOYEE, and OPERATIONAL_EMPLOYEE, reflecting role distinctions within the company. Each category then further specializes by experience level or specific function.

MANAGEMENT_EMPLOYEE is specialized into DEPARTMENT_MANAGER, DIVISION_MANAGER, and EXECUTIVE_MANAGER based on the scope of authority and organizational level. TECHNICAL_EMPLOYEE is specialized by seniority and responsibility into JUNIOR_DEVELOPER, SENIOR_DEVELOPER, and LEAD_DEVELOPER, reflecting technical expertise and leadership responsibility. OPERATIONAL_EMPLOYEE is specialized by specific operational function into WAREHOUSE_WORKER, CUSTOMER_SERVICE_REPRESENTATIVE, and SALES_REPRESENTATIVE.

Employee E1 (Zoran Pejovski), as a DEPARTMENT_MANAGER, inherits attributes through his classification path: from PERSON (Name, Email, Address), from EMPLOYEE (Employee_ID, Hire_date, Salary, Department_ID), from MANAGEMENT_EMPLOYEE (Budget_authority, Team_size, Management_level), and from DEPARTMENT_MANAGER (Department_managed, Strategic_planning_responsibility). This inheritance path remains straightforward because the hierarchy maintains single inheritance at all levels.

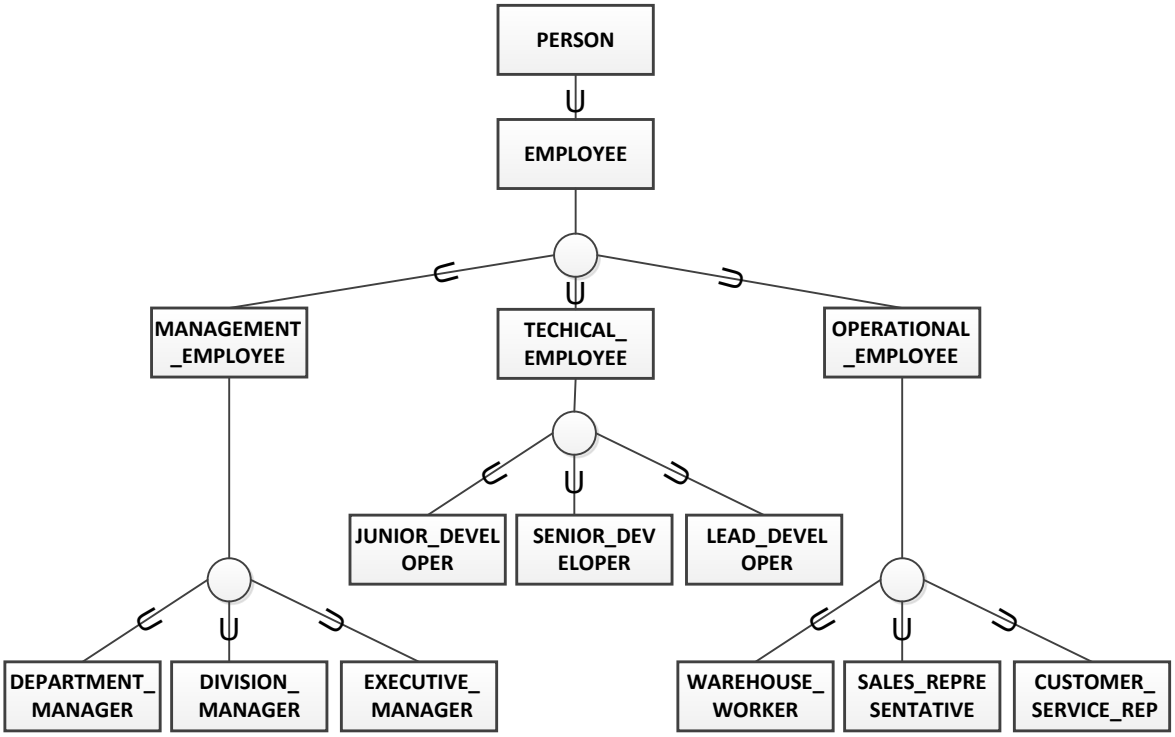


Figure 5.11: Company System Hierarchy Example

5.5.1.2 Advantages of Hierarchical Structures

Hierarchies offer several advantages that often make them the preferred structural choice when single inheritance appropriately captures business requirements:

Clear Inheritance Paths: Database developers follow exactly one path through the hierarchy when designing relational schemas or implementing queries. This makes the inheritance chain and attribute sources explicit and unambiguous.

Simplified Queries: Database queries follow straightforward paths for data retrieval. For example, to access all attributes for a FIRST_YEAR_BACHELOR, we follow one inheritance path through the hierarchy: FIRST_YEAR_BACHELOR to BACHELOR_STUDENT to STUDENT to PERSON. No alternative paths exist.

Easier Maintenance: Changes propagate along a single path. If STUDENT gains a new attribute, such as Disability_accommodation, it becomes available to all student subclasses through single inheritance, and the propagation path is easily traced.

Intuitive Structure: Hierarchies reflect organizational structures and natural classifications that stakeholders understand. When business analysts review the EER diagram, they immediately recognize the tree structure as matching their mental model of organizational roles. This alignment between the EER model and stakeholder mental models helps prevent miscommunication.

Implementation Efficiency: Database relational schemas derived from hierarchies remain relatively simple (Elmasri & Navathe, 2016).

These advantages explain why hierarchies are widely used in EER modeling. They provide effective modeling solutions when entities naturally follow one primary categorization path.

5.5.2 Specialization Lattices

When entities combine multiple classifications simultaneously and inherit characteristics from multiple specializations, we need a different classification approach: **specialization lattices**.

> **Definition: Specialization Lattice**
>
> A specialization lattice is a structure where a subclass can belong to more than one superclass-subclass relationship. This creates multiple inheritance, allowing subclasses to have multiple parents, forming a directed acyclic graph (DAG), a structure with multiple inheritance paths but no cycles, rather than a simple tree (Elmasri & Navathe, 2016).

In a lattice structure, a subclass can inherit from multiple superclasses simultaneously, combining attributes and relationships from multiple classification dimensions

Lattices are necessary when business requirements require entities to belong to multiple classification hierarchies simultaneously. For example, consider a TEACHING_ASSISTANT who is simultaneously a DOCTORAL_STUDENT (enrolled in a doctoral program, working toward a dissertation) and an INSTRUCTOR (teaching undergraduate courses for compensation). The TEACHING_ASSISTANT inherits from both DOCTORAL_STUDENT and INSTRUCTOR, creating multiple inheritance. From DOCTORAL_STUDENT, the teaching assistant inherits student-specific attributes like Dissertation_topic, Research_area, and Qualifying_exam_date. From INSTRUCTOR, they inherit teaching-related attributes like Course_taught, Teaching_schedule, and Student_evaluation_average. This dual classification accurately reflects the reality that doctoral students who teach operate in both roles simultaneously.

5.5.2.1 Characteristics of Lattice Structures

Lattices have several characteristics that distinguish them from hierarchies and enable more complex modeling:

Multiple Inheritance: Subclasses inherit from multiple superclasses simultaneously. For example, a TEACHING_ASSISTANT is simultaneously a subclass of both STUDENT and INSTRUCTOR, inheriting attributes and relationships from both parents.

Shared Subclasses: Some subclasses serve as connection points between multiple paths, representing entities that combine classifications from multiple dimensions. For example, a TEACHING_ASSISTANT is a *shared subclass* at the intersection of the STUDENT and INSTRUCTOR classifications.

Complex Inheritance Paths: Multiple paths may exist from a shared subclass to root ancestors. TEACHING_ASSISTANT inherits from STUDENT, which inherits from PERSON; simultaneously, TEACHING_ASSISTANT inherits from INSTRUCTOR, which also inherits from PERSON. Two different paths trace back to PERSON, creating potential attribute conflicts that require careful resolution.

Greater Modeling Flexibility: Lattices model complex overlapping classifications that hierarchies cannot represent. When business requirements involve entities that combine multiple independent classification dimensions, lattices provide the structural flexibility necessary for accurate representation (Elmasri & Navathe, 2016).

University System Lattice Example

To illustrate the concept of lattices, we examine in detail the structure around teaching and research assistants in our University system.

The lattice structures shown in Figure 5.12 illustrate that doctoral students serving as teaching assistants hold dual organizational roles. TEACHING_ASSISTANT is a **shared subclass** because it simultaneously inherits from both DOCTORAL_STUDENT (through the student specialization hierarchy) and INSTRUCTOR (directly). Teaching assistants in universities receive employee benefits and salaries while maintaining full doctoral student status, including dissertation work and degree progression.

For clarity, Figure 5.12 shows only entity names and their inheritance relationships to illustrate the lattice structure. The specialized attributes from each inheritance path are described in detail below.

Consider a specific teaching assistant, perhaps a doctoral student teaching an undergraduate database systems course. This individual appears in our system as a TEACHING_ASSISTANT entity. Through multiple inheritance, they accumulate attributes from diverse sources:

From PERSON (via two inheritance paths):

- SSN, Name, Birth_date, Address, Email
- (Note: These appear in both inheritance paths but represent a single entity; the system recognizes one set of PERSON attributes, not duplicates)

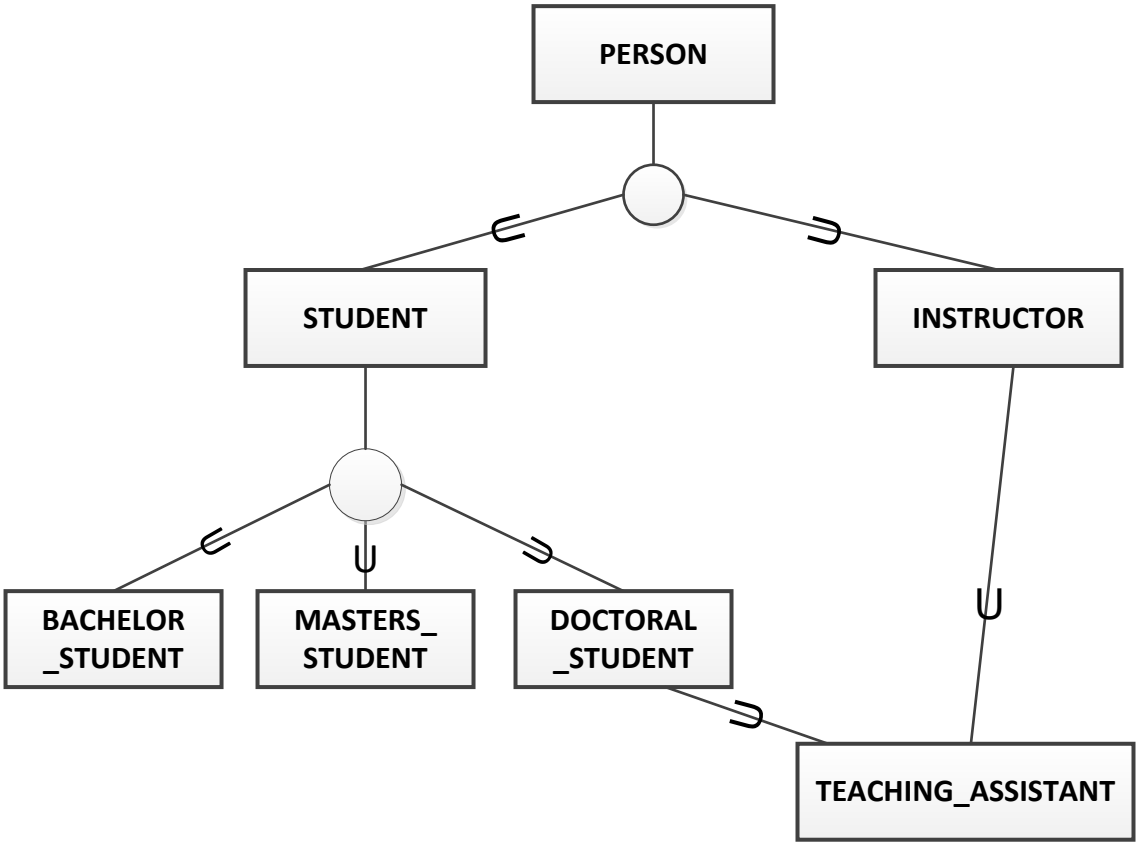


Figure 5.12: University System Lattice Example

From STUDENT path (PERSON → STUDENT → DOCTORAL_STUDENT):

- Student_ID (for academic records)
- Enrollment_date (doctoral program start)
- ECTS_completed (toward doctoral degree requirements)
- Dissertation_topic (doctoral research area)
- Defense_commitee
- Expected_defense_date

From INSTRUCTOR path (PERSON → INSTRUCTOR):

- Instructor_ID (for payroll and benefits)
- Salary (teaching assistant stipend)
- Hire_date (when assistantship began)
- Department_ID (department providing employment)
- Course_taught
- Teaching_schedule
- Student_evaluation_average (student ratings of teaching)

From TEACHING_ASSISTANT (local attributes combining both roles):

- Hours_per_week (assistantship time commitment)

- Semester_appointment (which semesters assistantship covers)
- TA_training_completed (required training status)
- Supervisor_ID (faculty supervisor for assistantship duties)

Multiple Inheritance Complexity

The lattice structure introduces complexity requiring careful handling. When TEACHING_ASSISTANT inherits from both DOCTORAL_STUDENT and INSTRUCTOR, both paths ultimately trace back to PERSON. The system must address several challenges:

Attribute Conflict Resolution: When the same attribute appears in multiple inheritance paths (e.g., SSN, Name, Birth_date, and Address from both INSTRUCTOR → PERSON and DOCTORAL_STUDENT → STUDENT → PERSON paths), the system treats them as the same attribute rather than creating duplicates. TEACHING_ASSISTANT entities have one SSN and one name, not two sets, even though these attributes are inherited through two different paths converging at PERSON.

Constraint Inheritance: Constraints from all ancestor paths must be satisfied. If INSTRUCTOR participates totally in works_for DEPARTMENT and STUDENT participates totally in belongs_to FACULTY, then TEACHING_ASSISTANT must satisfy both participation constraints: assigned to both a department (for employment) and a faculty (for academic studies). Similarly, the disjoint constraint on STUDENT specialization ensures that teaching assistants must be of type DOCTORAL_STUDENT entities, not bachelor's or master's students.

Relationship Participation: TEACHING_ASSISTANT participates in relationships inherited from all ancestors. Through the INSTRUCTOR path, they participate in works_for relationships with DEPARTMENT (receiving salary and benefits). Through the STUDENT→DOCTORAL_STUDENT path, they participate in enrolls_in relationships with doctoral seminars and advised_by relationships with dissertation supervisors. Additionally, they have specialized relationships, such as assists_in undergraduate COURSE sections. They participate in all these relationship types because they fulfill both organizational roles simultaneously, pursuing a doctoral degree and serving as university teaching staff.

Company System Lattice Example

To illustrate lattice structures within a company, Figure 5.13 shows an alternative employee classification structure in which employees perform multiple specialized roles. This example differs from the job function specialization (MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, ADMIN_EMPLOYEE) discussed in earlier sections. This lattice structure depicts technical sales managers, employees who manage sales teams while maintaining technical engineering expertise and personal sales responsibilities. Such individuals are common in modern technology companies where selling complex technical products requires deep technical knowledge combined with management and sales skills.

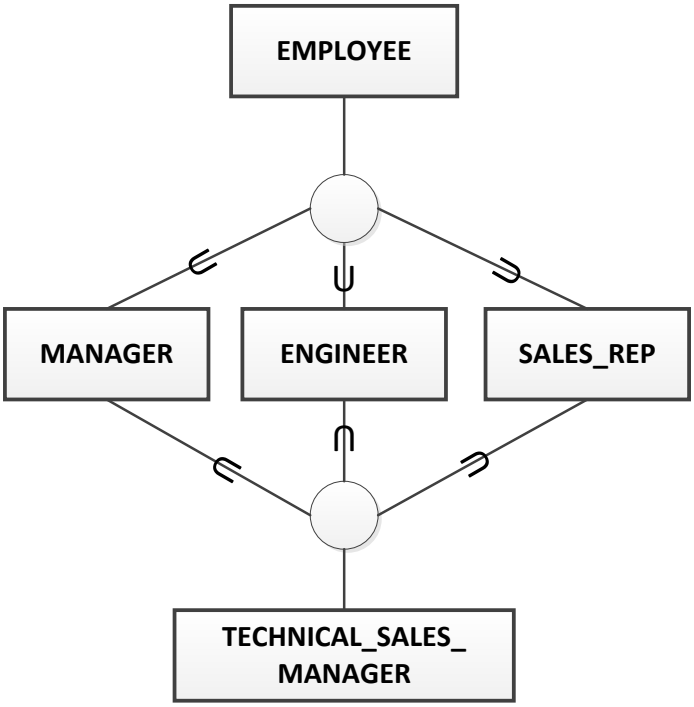


Figure 5.13: Company System Lattice Example

For clarity, Figure 5.13 shows only entity names and their inheritance relationships to illustrate the lattice structure. The specialized attributes from each inheritance path are described below.

Through multiple inheritance, a TECHNICAL_SALES_MANAGER accumulates attributes from three different specialization paths:

From MANAGER:

- Budget_authority (manages team budget)
- Team_size (number of direct reports)
- Management_level (organizational hierarchy position)
- Strategic_planning_responsibilities

From ENGINEER:

- Technical_certifications (product expertise)
- Engineering_specialization (technical domain)
- Product_development_experience
- Technical_publication_count

From SALES_REP:

- Sales_territory (geographic or market segment)
- Commission_rate (performance-based compensation)
- Customer_portfolio_size (number of active accounts)
- Monthly_sales_quota (revenue targets)

This employee is responsible for managing technical sales teams (MANAGER attributes), maintains extensive technical product expertise to advise customers (ENGINEER attributes), and personally manages key customer accounts (SALES_REP attributes). Forcing such employees into a single hierarchy would sacrifice semantic accuracy. They don't "primarily" serve as managers with some technical skills, nor as engineers who happen to sell; they combine all three specialized roles equally.

The lattice structure accurately models this organizational reality. When the company evaluates this employee's performance, it assesses management effectiveness (team performance, budget management), technical contributions (product expertise, quality of customer technical support), and personal sales achievement (revenue generated, customer satisfaction). All three evaluation dimensions are equally important; the employee genuinely fulfills all three roles simultaneously.

5.5.3 Hierarchies vs. Lattices

Choosing between hierarchical and lattice structures requires careful analysis of organizational requirements, implementation constraints, and semantic accuracy priorities. The decision depends on whether the requirements can be modeled with a single inheritance hierarchy or require multiple inheritance for accuracy.

5.5.3.1 When to Use Hierarchies

Hierarchies are appropriate when organizational classifications follow tree-like patterns:

Single Classification: Entities have one primary categorization that does not combine with other classifications. For example, under the Bologna Process, students pursue only one degree level at a time: they are either bachelor's students or master's students, not both simultaneously. This naturally creates a hierarchy.

No Overlapping Roles: Subclass memberships are mutually exclusive. When business rules enforce exclusive classification, for example, "employees have exactly one primary job function", hierarchies accurately model this constraint.

Implementation Simplicity: When project constraints prioritize simplicity, such as limited developer expertise, performance requirements, or maintenance considerations, hierarchies provide significantly simpler implementation.

Organizational Clarity Requirements: Business processes follow tree-like structures that stakeholders naturally understand. When organizational charts, reporting structures, and classification schemes all reflect hierarchical thinking, EER models that align with this mental model facilitate validation and reduce miscommunication with domain experts.

5.5.3.2 When to Use Lattices

Lattices become necessary when organizational reality resists hierarchical classification:

Complex Overlapping Roles: Entities naturally combine multiple classifications that cannot be reduced to a single primary categorization. For example, doctoral students who teach function as both students and instructors; they are not "primarily" one or the other, but both. Hierarchies cannot model this accurately without semantic distortion.

Multiple Valid Perspectives: Different stakeholders view entities through different classification lenses, and no single perspective takes precedence. The registrar views teaching assistants as students (for tracking degree progress), while human resources views them as instructors (for managing payroll and benefits). Both perspectives are valid and necessary, requiring a lattice structure that accommodates multiple simultaneous classifications.

Semantic Accuracy Importance: When modeling accuracy outweighs implementation convenience, lattices provide the structural flexibility needed to faithfully capture organizational reality. For example, technical sales managers combine management, engineering, and sales roles; representing them hierarchically would sacrifice accuracy for convenience.

Domain Modeling Requirements: Complex domains with sophisticated business rules often demand the expressiveness that only lattices provide.

5.5.3.3 Implementation Complexity Trade-offs

The choice between hierarchies and lattices involves explicit trade-offs across multiple dimensions (see Table 5.4).

Table 5.4: Implementation Complexity Trade-offs

Aspect	*Hierarchy*	*Lattice*
Conceptual Clarity	Simpler, one path per entity	More complex, multiple inheritance paths
Attribute Inheritance	Straightforward, single path	Requires conflict resolution across multiple paths
Query Complexity	Lower, linear joins	Higher, multi-path joins and unions
Maintenance Effort	Easier, changes propagate predictably	More challenging, changes affect multiple paths
Semantic Accuracy	May sacrifice accuracy for simplicity	More accurate for complex domains
Database Schema	Simpler, fewer tables or simpler discriminators	More complex junction tables or complex mappings
Development Time	Faster implementation	Slower due to complexity
Runtime Performance	Better, simpler queries	Potentially worse, complex joins

These trade-offs significantly impact design decisions. When hierarchies accurately reflect organizational reality, their simplicity makes them the preferred option. When only lattices provide the necessary semantic accuracy, their complexity is justified by the requirement for correct system behavior. The trade-offs presented in Table 5.4 represent well-established considerations in EER modeling practice (Elmasri & Navathe, 2016; Teorey et al., 2011)

5.5.4 Depth Considerations in EER Structures

Beyond choosing between hierarchies and lattices, database designers must consider hierarchy depth—how many levels of specialization to model.

Based on practical experience and implementation considerations, hierarchies should generally be limited to 3-4 levels unless business requirements clearly justify deeper structures (Teorey et al., 2011; Elmasri & Navathe, 2016).

In our University example, we examined a four-level structure for illustration purposes:

Level 1: PERSON

Level 2: STUDENT

Level 3: BACHELOR_STUDENT

Level 4: FIRST_YEAR_BACHELOR

Visualized as:

PERSON → STUDENT → BACHELOR_STUDENT → FIRST_YEAR_BACHELOR

Four levels provide meaningful classification: people in general (PERSON), students specifically (STUDENT), bachelor's-level students (BACHELOR_STUDENT), and first-year bachelor's students (FIRST_YEAR_BACHELOR). Adding a fifth level, for example, specializing FIRST_YEAR_BACHELOR by major field, would likely add more complexity than value unless specific business processes require that granularity.

Some domains legitimately require deeper hierarchies. However, when hierarchies extend beyond 4 levels, database designers should consider whether alternative modeling approaches would be more appropriate. Attributes can often replace excessive depth in hierarchy. For example, in our University example, we created year-based subclasses (FIRST_YEAR_BACHELOR through FOURTH_YEAR_BACHELOR) to demonstrate multi-level hierarchies. However, an alternative approach would use a single BACHELOR_STUDENT entity with a Study_year attribute (First, Second, Third, Fourth), providing equivalent information with less structural complexity. The choice depends on whether year-specific attributes and relationships justify the additional hierarchy level.

Hierarchies, lattices, and depth considerations demonstrate that EER structural choices significantly impact both semantic accuracy and implementation efficiency. Hierarchies provide simplicity and clarity when organizational classifications naturally form trees. Lattices enable accurate modeling of complex overlapping roles when multiple inheritance becomes necessary. Moderate depth provides meaningful classification without excessive complexity.

In Section 5.6, we examine the formal Chen notation that precisely represents these structures in EER diagrams, completing our comprehensive understanding of EER modeling techniques.

5.6 Chen Notation for EER Constructs

Having developed a conceptual understanding of EER concepts through Sections 5.1-5.5, we now examine the **formal Chen notation** for accurately representing these constructs in EER diagrams. Throughout the previous sections, we've introduced EER concepts through illustrative diagrams. Now we formalize these representations using Chen notation, the standard graphical language for EER modeling that extends the Chen ER

notation from Chapter 4. This notation provides symbols specifically designed for specialization, generalization, and constraint representation (Hristoski & Dimovski, 2022; Elmasri & Navathe, 2016).

5.6.1 Basic Chen EER Symbols

Specialization/Generalization Circle

The central symbol in Chen EER notation is the **specialization circle**, which serves as the connection point between the superclass and its subclasses (illustrated in Figures 5.9 and 5.14).

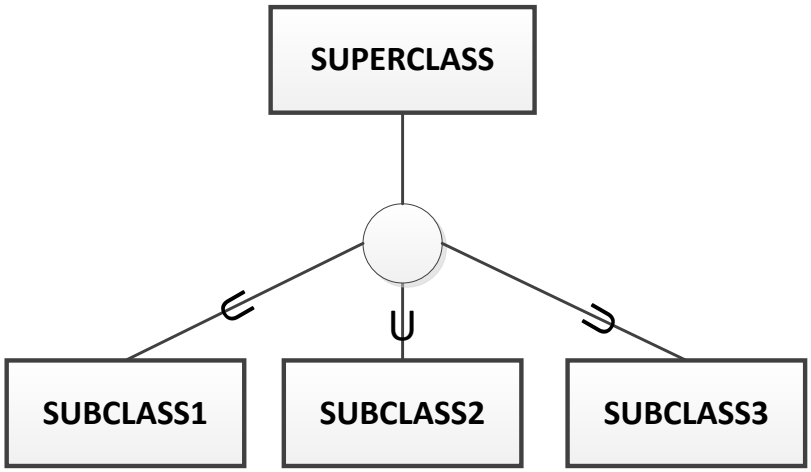


Figure 5.14: Unified Circle Notation

Key Components:

1. **Superclass Rectangle**: Standard entity rectangle from basic ER notation (Chapter 4), positioned above the specialization circle, containing the general entity type name
2. **Connection Line**: Single line (partial) or double line (total) connecting the superclass to the specialization circle
3. **Specialization Circle** (○): Central connection point containing constraint symbols: "d" (disjoint) or "o" (overlapping)
4. **Subset Symbols (⊆)**: Positioned on lines connecting the circle to each subclass, indicating that each subclass is a subset of the superclass ($S_i \subseteq C$)
5. **Subclass Rectangles**: Standard entity rectangles positioned below the circle, containing specialized entity type names and their local attributes (attributes inherited from the superclass are not repeated)

5.6.2 Constraint Representation in Chen Notation

Chen notation uses specific symbols to represent the disjointness and completeness constraints introduced in Section 5.4 (Hristoski & Dimovski, 2022; Elmasri & Navathe, 2016).

5.6.2.1 Disjointness Constraints

Disjoint Constraint (d)

Disjoint specializations are marked with lowercase "**d**" inside the circle (see Figure 5.15). This indicates that subclasses are mutually exclusive; an entity can belong to at most one subclass (Section 5.4.2.1).

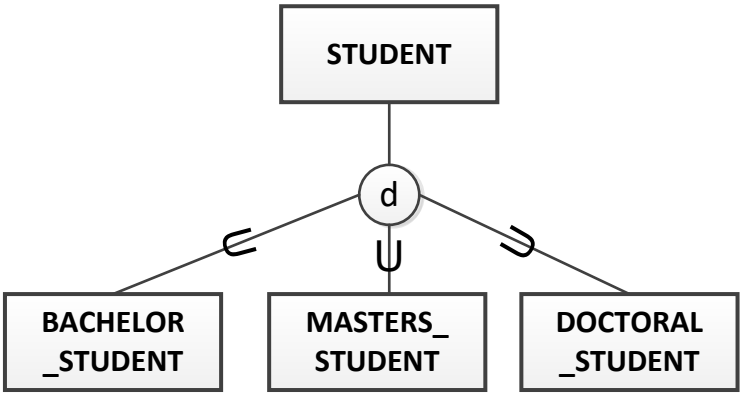


Figure 5.15: Disjoint Constraint Notation Example

Example: A student belongs to exactly one academic level (BACHELOR_STUDENT, MASTERS_STUDENT, or DOCTORAL_STUDENT), not multiple levels simultaneously.

Overlapping Constraint (o)

Overlapping specializations are marked with lowercase "**o**" inside the circle (see Figure 5.16). This indicates that entities can belong to multiple subclasses simultaneously (Section 5.4.2.2).

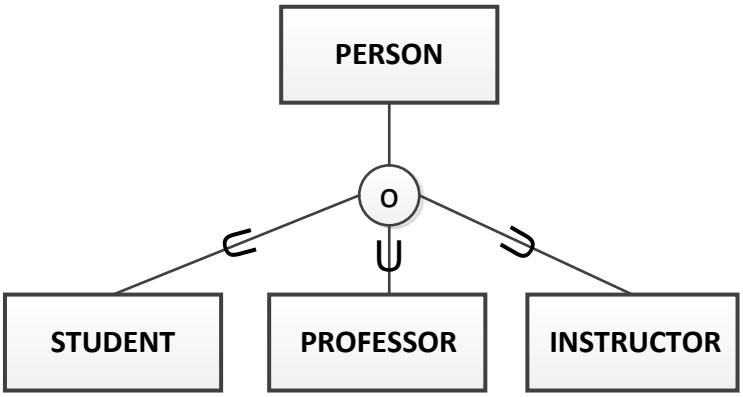


Figure 5.16: Overlapping Constraint Notation Example

Example: A person can hold multiple university roles simultaneously; a doctoral student who teaches is both STUDENT and INSTRUCTOR.

5.6.2.2 Completeness Constraints

Total Specialization (Double Line)

Total specialization is indicated by a **double line** from the superclass to the circle (see Figure 5.17). This indicates that every superclass entity must belong to at least one subclass (Section 5.4.3.1).

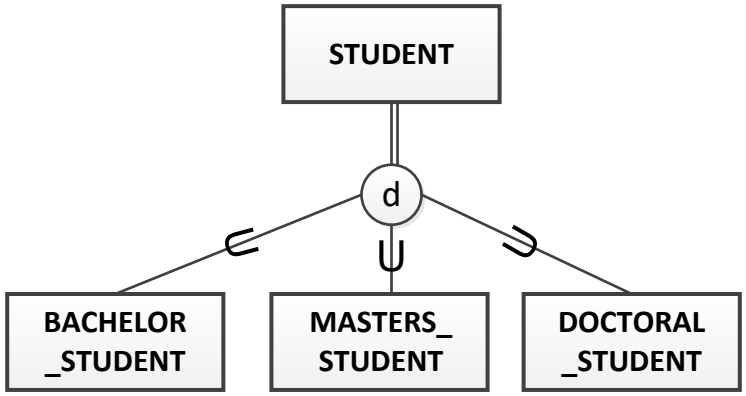


Figure 5.17: Total Participation Notation Example

Example: Every student must belong to one of the academic levels (BACHELOR_STUDENT, MASTERS_STUDENT, or DOCTORAL_STUDENT). There are no unclassified students.

Partial Specialization (Single Line)

Partial specialization is indicated by a **single line** from the superclass to the circle (see Figure 5.18). This indicates that some superclass entities may not belong to any subclass (Section 5.4.3.2).

Example: Some employees are specialized (MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, ADMIN_EMPLOYEE), but many are general employees who do not belong to any of these specialized categories.

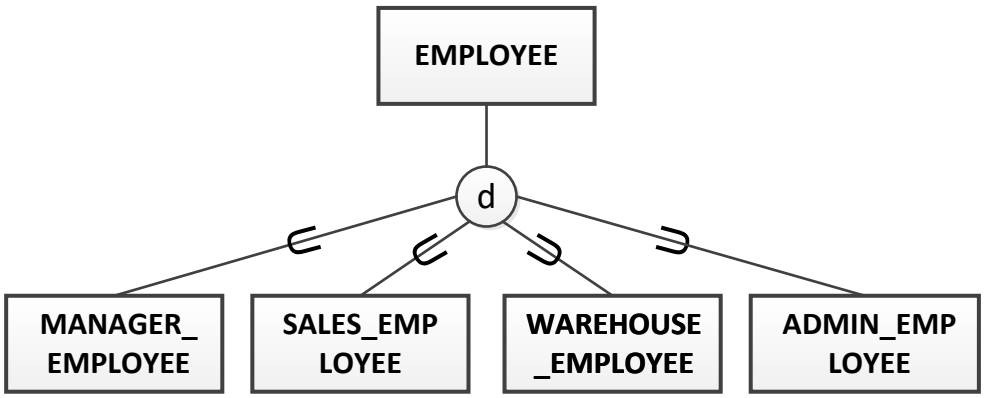


Figure 5.18: Partial Participation Notation Example

5.6.3 Integrated Notation Example: University System

This subsection demonstrates how the individual notation elements are combined into a cohesive EER diagram. Figure 5.19 applies all Chen notation concepts - specialization circles, constraint symbols (d/o), participation indicators (single/double lines), subset symbols (⊆), and domain constraints - to represent the key entity hierarchies from our University case study using Chen notation conventions (Hristoski & Dimovski, 2022; Elmasri & Navathe, 2016). While not exhaustive of all University entities and relationships from Chapter 4, this diagram illustrates the correct application of notation for multi-level hierarchies with multiple specialization paths.

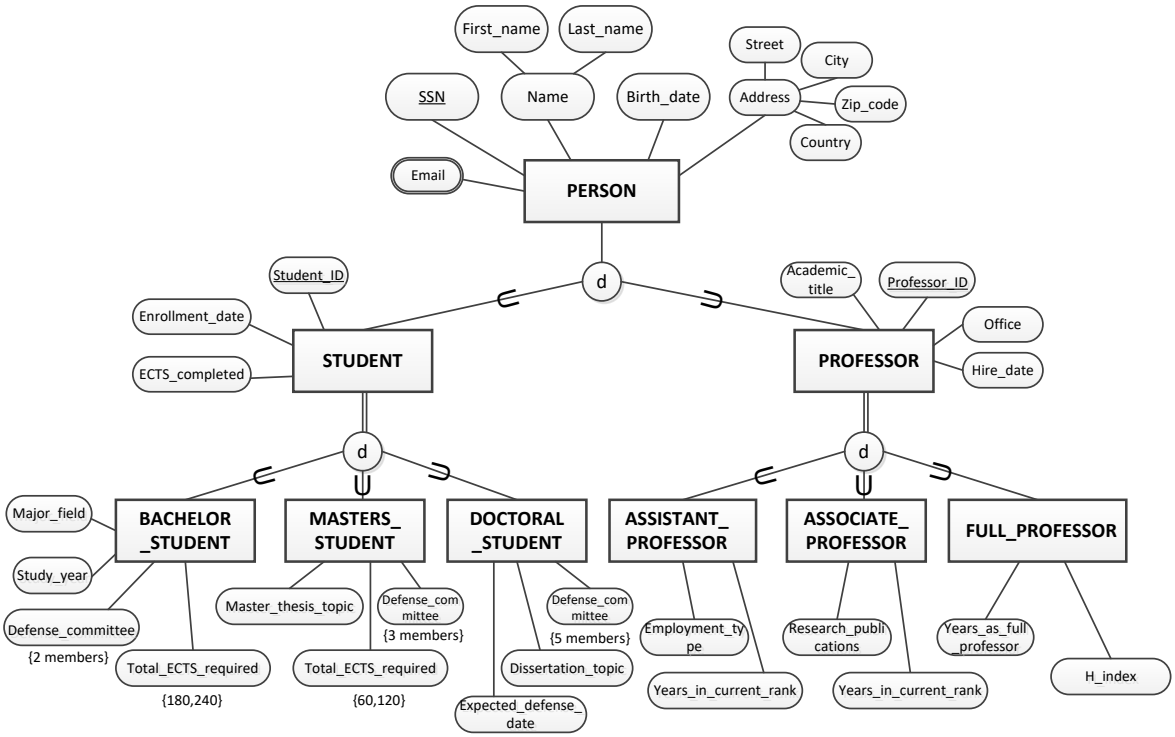


Figure 5.19: University EER Diagram with Chen Notation - Multi-Level Specialization and Composite Attributes

Chen Notation Elements Demonstrated:

1. **Composite Attributes**: Name (composed of First_name, Last_name) and Address (composed of Street, City, Zip_code, Country) shown with their component attributes connected to the composite attribute oval.
2. **Disjoint Partial Specialization** at the top level: PERSON specializes into STUDENT or PROFESSOR (marked with "d"), with partial participation (single line).
3. **Disjoint Total Specializations**: Both student academic levels and professor rank use disjoint constraints (marked with "d") and total participation (double lines).
4. **Multi-Level Hierarchy**: Three levels from PERSON to leaf entities, demonstrating inheritance across multiple levels.
5. **Proper Attribute Placement**:
    - Universal attributes (SSN, Name, Birth_date, Address, Email) at PERSON level, shared by all university people
    - Intermediate-level attributes (Student_ID, ECTS_completed, Enrollment_date for STUDENT; Professor_ID, Office, Hire_date, Academic_title for PROFESSOR)
    - Specialized attributes at subclass levels:
    - BACHELOR_STUDENT: Major_field, Study_year, Total_ECTS_required, Defense_committee
    - MASTERS_STUDENT: Master_thesis_topic, Total_ECTS_required, Defense_committee

- DOCTORAL_STUDENT: Dissertation_topic, Defense_committee, Expected_defense_date
- ASSISTANT_PROFESSOR: Employment_type, Years_in_current_rank
- ASSOCIATE_PROFESSOR: Research_publications, Years_in_current_rank
- FULL_PROFESSOR: Years_as_full_professor, H_index

6. **Domain Constraints:** Curly brackets { } specify allowed attribute values (detailed below)

Domain Constraint Notation

Domain constraints appear in curly brackets { } near the attribute they constrain. This mathematical set notation precisely specifies valid attribute values:

Discrete Value Sets:

- {180, 240}: Bachelor students must complete either 180 or 240 ECTS credits, representing 3-year or 4-year programs under the Bologna Process (European Commission, 2015)
- {60, 120}: Master's students must complete either 60 or 120 ECTS credits, representing 1-year or 2-year programs

Single Required Values:

- {2 members}: Bachelor's defense committees have exactly 2 members
- {3 members}: Master's defense committees have exactly 3 members
- {5 members}: Doctoral defense committees have exactly 5 members

Other Constraint Types (Examples):

- Range constraints: {18..65} for age limits
- Enumerated types: {Full-time, Part-time, Contract} for employment status

This notation enables database designers to translate business rules directly into implementation constraints (such as CHECK constraints in SQL DDL, discussed in Chapter 9). This integrated example demonstrates how Chen notation enables precise communication of entity hierarchies, inheritance patterns, and business rules. Section 5.7 presents complete EER diagrams for both the University and the Company case studies.

5.6.4 Integrated Notation Example: Company System

This section demonstrates Chen notation applied to our Company case study, illustrating specialization with **partial participation** and **disjoint constraints** (see Figure 5.21). Unlike the University example, where every student must have an academic level (total participation), the Company system recognizes that many employees perform general roles without specialized job function classification. This example illustrates proper notation for modeling organizational structures where specialization is selective rather than universal.

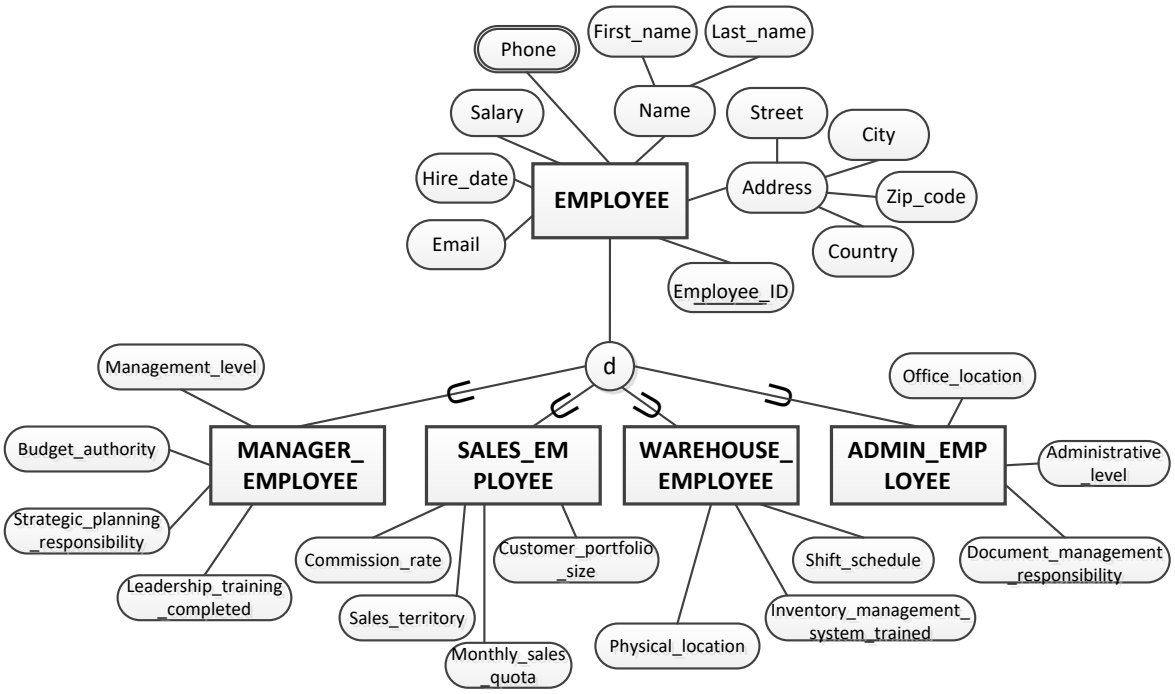


Figure 5.21: Company Employee Specialization with Chen Notation

For diagram clarity, Figure 5.21 shows representative specialized attributes for each subclass. The complete attribute sets defined in Section 5.3.1.2 include additional attributes not shown in this diagram but maintained in the detailed model.

Chen Notation Elements Demonstrated:

1. **Disjoint, Partial Specialization (d, p)**: The combination of:
    - **Disjoint (d)**: An employee can belong to at most one specialized job function category
    - **Partial (single line)**: Not every employee must belong to a specialized category

This model's business reality is based on our Company data:

- 3 employees (E1, E3, E5) are MANAGER_EMPLOYEE with specialized management responsibilities

- 4 employees (E2, E6, E7, E8) are WAREHOUSE_EMPLOYEE with warehouse assignments

- 1 employee (E4: Anita Toshevska) is a general employee performing marketing functions without requiring specialized subclass classification

2. **Four Specialized Subclasses**:
    - **MANAGER_EMPLOYEE**: Supervisory roles with budget authority (e.g., E1 managing Sales department D1, E3 managing Marketing department D2, E5 managing HR department D3)
    - **SALES_EMPLOYEE**: Customer-facing roles with territory assignments and commission structures

- **WAREHOUSE_EMPLOYEE**: Physical inventory management with location and shift assignments (E2 at W1 Skopje, E6 at W3 Ohrid, E7 at W1 Skopje, E8 at W2 Bitola)
- **ADMIN_EMPLOYEE**: Administrative support roles with office-type designations and department support areas

3. **Composite Attributes**: Name (composed of First_name, Last_name) and Address (composed of Street, City, Zip_code, Country) shown with their component attributes
4. **Multivalued Attribute**: Phone shown with double-line oval connecting to EMPLOYEE
5. **Specialized Attributes Shown** (representative subset):
   - MANAGER_EMPLOYEE: Management_level, Budget_authority, Strategic_planning_responsibility, Leadership_training_completed
   - SALES_EMPLOYEE: Commission_rate, Sales_territory, Monthly_sales_quota, Customer_portfolio_size
   - WAREHOUSE_EMPLOYEE: Physical_location, Shift_schedule, Inventory_management_system_trained
   - ADMIN_EMPLOYEE: Office_location, Administrative_level, Document_management_responsibility

   Note: Additional specialized attributes defined in Section 5.3.1.2 (such as Number_of_direct_reports for managers, Year_to_date_sales for sales employees, etc.) are part of the complete model but not shown in this diagram for visual clarity.
6. **Proper Attribute Placement**:
   - Universal attributes (Employee_ID, Name, Salary, Hire_date, Email, Phone, Address) at the EMPLOYEE level, shared by all employees
   - Specialized attributes at subclass levels, applying only to employees in those categories

Understanding Partial Participation

The partial participation constraint models an important business reality: **not all employees fit into predefined specialized categories**. In our Company data (Chapter 11):

- 3 employees (E1: Zoran Pejovski, E3: Marija Naumoska, E5: Naum Stojanoski) have Manages_Dept_ID values, classifying them as MANAGER_EMPLOYEE
- 4 employees (E2: Nikola Delcevski, E6: Hristina Miloseska, E7: Dimitar Tocko, E8: Bijana Osmani) have Warehouse_ID assignments, classifying them as WAREHOUSE_EMPLOYEE
- 1 employee (E4: Anita Toshevska, Marketing department D2, Salary: 45000) is a general employee without a specialized classification

General employees like E4 still have all common EMPLOYEE attributes (Employee_ID, Name, Salary, Hire_date, Email, Phone, Address) and participate in standard relationships (works_for DEPARTMENT, processes ORDERS), but they don't require specialized subclass

modeling. This partial participation prevents over-classification, maintains model simplicity, and accurately represents organizational structure.

Contrast with University Total Participation:

Compare this Company example with the University STUDENT specialization (Figure 5.19):

- **Company (d, p)**: Partial, general employees like E4 exist without specialized classification
- **University (d, t)**: Total, every student must belong to an academic level (BACHELOR_STUDENT, MASTERS_STUDENT, or DOCTORAL_STUDENT)

This difference reflects domain requirements: universities must classify all students by academic level for academic administration under the Bologna Process (European Commission, 2015), while companies often have diverse general employee populations performing valuable work without requiring specialized categorization. This notation framework demonstrates how Chen EER diagrams precisely capture organizational classification policies through appropriate constraint selection, enabling database designs that accurately reflect business rules and operational realities.

5.6.5 Attribute-Defined Specialization Notation

When specialization depends on the value of a specific attribute in the superclass, we use **attribute-defined specialization** (Elmasri & Navathe, 2016). This modeling approach automatically determines subclass membership based on attribute values, eliminating the need for manual classification decisions and ensuring consistent entity placement across the database.

5.6.5.1 Conceptual Foundation

Consider a company that categorizes employees by employment type: full-time, part-time, or contract. Instead of manually assigning each employee to a specific category, the Employment_type attribute automatically determines subclass membership. An employee with Employment_type = 'Full-time' automatically belongs to the FULL_TIME_EMPLOYEE subclass, one with Employment_type = 'Part-time' belongs to PART_TIME_EMPLOYEE, and one with Employment_type = 'Contract' belongs to CONTRACT_EMPLOYEE.

This approach offers several advantages:

- **Automatic Classification**: Subclass membership follows directly from attribute values, eliminating manual classification errors
- **Consistency**: The same attribute value always leads to the same subclass membership
- **Simplicity**: Database users simply set the attribute value; the system handles classification
- **Disjoint Enforcement**: Attribute-defined specialization using single-valued attributes inherently enforces disjoint constraints

5.6.5.2 Chen Notation for Attribute-Defined Specialization

Chen notation demonstrates attribute-defined specialization by displaying the **attribute** name on the line connecting the superclass to the specialization circle. Each subclass then shows its **defining predicate**, which is the specific condition that determines membership in that subclass.

Figure 5.22 illustrates the complete notation structure.

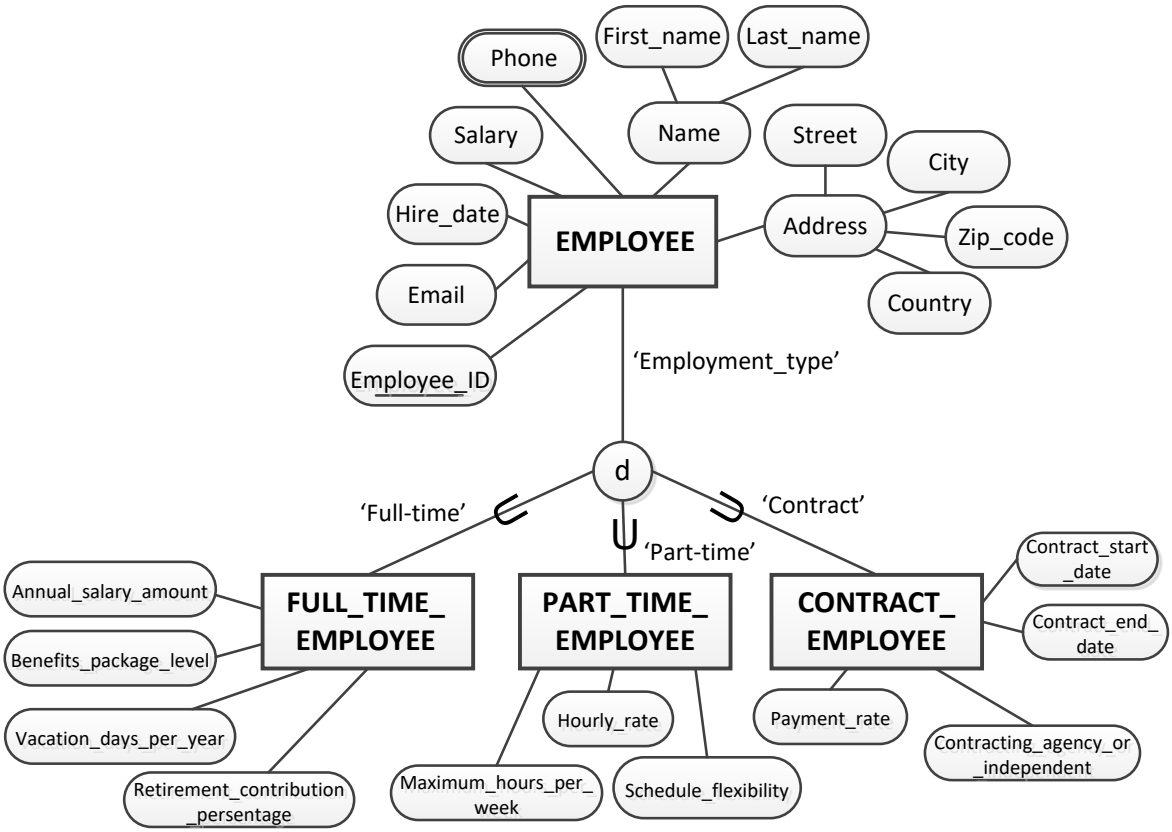


Figure 5.22: Attribute-Defined Specialization in Company Database

For diagram clarity, Figure 5.22 shows representative specialized attributes for each subclass. The complete attribute sets defined in Section 5.3.1.2 include additional attributes not shown in this diagram but maintained in the detailed model.

Notation Elements Explained

1. **Defining Attribute Label**: The attribute name (Employment_type) appears next to the line connecting EMPLOYEE to the specialization circle, indicating that this attribute's value determines subclass membership for all subclasses in this specialization.
2. **Disjoint Constraint**: The letter d inside the circle indicates disjoint specialization. Attribute-defined specializations that use single-valued attributes are inherently disjoint. Each employee can have only one Employment_type value, so it belongs to exactly one subclass.
3. **Defining Predicates**: Below each subclass, we specify the condition that determines membership: Employment_type = 'Full-time', Employment_type = 'Part-time', or

Employment_type = 'Contract'. These predicates make the classification rules explicit and unambiguous.

4. **Subset Symbols (⊆)**: The subset symbols connecting the specialization circle to each subclass indicate that subclass entities are subsets of the superclass entity set.

Semantic Interpretation

The complete diagram conveys precise semantics:

Automatic Membership Determination: When we insert a new employee with Employment_type = 'Full-time', the database system automatically recognizes this employee as a member of the FULL_TIME_EMPLOYEE subclass. No manual classification operation is needed; the attribute value determines subclass membership.

Mutual Exclusivity: Because Employment_type is a single-valued attribute, each employee has exactly one employment type value. An employee cannot be both full-time and part-time at the same time. This single-valued characteristic of the attribute ensures the disjoint constraint.

Complete Constraint Definition: Each entity with Employment_type = 'Full-time' belongs to FULL_TIME_EMPLOYEE; each entity with Employment_type = 'Part-time' belongs to PART_TIME_EMPLOYEE; and each entity with Employment_type = 'Contract' belongs to CONTRACT_EMPLOYEE. The attribute value fully and automatically determines subclass membership.

Consistent Classification: Unlike user-defined specialization, where database administrators manually assign entities to subclasses, attribute-defined specialization guarantees consistent classification. Two employees with the same Employment_type value will always belong to the same subclass.

Contrast with User-Defined Specialization

Understanding attribute-defined specialization becomes clearer when contrasted with user-defined specialization:

User-Defined Specialization: Subclass membership depends on explicit user actions during entity creation or update. There is no automatic rule dictating which entities belong to which subclasses. For example, identifying which employees are managers might involve examining multiple factors, such as experience, responsibilities, and organizational structure, that cannot be captured in a single attribute.

Attribute-Defined Specialization: Subclass membership is automatically determined by attribute value. A clear, explicit rule manages classification. Any employee with Employment_type = 'Full-time' automatically becomes a FULL_TIME_EMPLOYEE without requiring manual intervention.

The choice between user-defined and attribute-defined specialization depends on whether classification can be captured by a single attribute's value or requires more complex business logic.

Participation Constraint Considerations

The diagram shown represents **partial participation** (indicated by the single line from EMPLOYEE to the specialization circle). This means some employees might have Employment_type values not included in our specialization, or NULL Employment_type values.

For **total participation**, we would use a double line from EMPLOYEE to the circle, indicating that:

- Every employee must have an Employment_type value
- That value must be one of {'Full-time', 'Part-time', 'Contract'}
- Therefore, every employee belongs to exactly one of the three subclasses

Total participation in attribute-defined specialization requires careful domain definition: the possible values of the defining attribute must exactly match the set of defined subclasses.

5.6.6 Multi-Level Specialization Representation

Multi-level specialization hierarchies and lattices require careful notation to show constraint specifications at each level and to clearly indicate shared subclass connections. Figure 5.23 demonstrates proper Chen notation for the University teaching assistant structure discussed conceptually in Section 5.5.2, illustrating how notation represents both hierarchical and lattice structures within a single diagram.

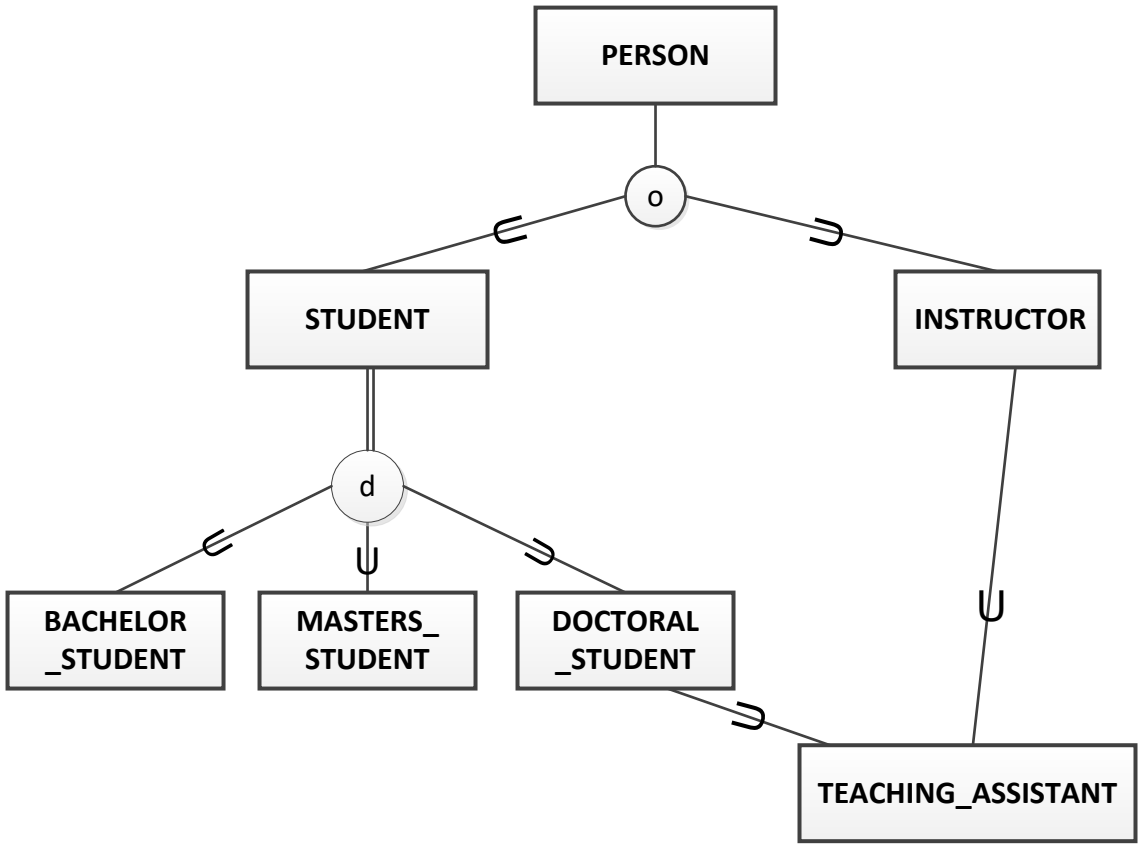


Figure 5.23: Multi-Level Hierarchy with Lattice Structure

Chen Notation Elements Demonstrated:

1. **First-Level Overlapping Partial Specialization (o, t)**:
   - PERSON is specialized into STUDENT and INSTRUCTOR
   - Overlapping (o): A person can hold both roles simultaneously (doctoral students who teach)
   - Partial (single line): Not every person belongs to the shown subclasses
     Note: This diagram shows only two of the five PERSON subclasses from the complete University model. Recall from Section 5.4.2.2 that PERSON fully specializes into {STUDENT, PROFESSOR, INSTRUCTOR, STAFF, ALUMNUS} with overlapping and total participation (o, t). Figure 5.23 focuses on STUDENT and INSTRUCTOR branches to illustrate the lattice structure that includes TEACHING_ASSISTANT. The partial participation (single line) reflects this selective representation - PROFESSOR, STAFF, and ALUMNUS subclasses exist but are not shown in this diagram.
2. **Second-Level Disjoint Total Specialization (d, t)**:
   - STUDENT specializes into BACHELOR_STUDENT, MASTERS_STUDENT, and DOCTORAL_STUDENT
   - Disjoint (d): Students pursue only one degree level at a time (mutually exclusive academic programs)
   - Total (double line): Every student must be enrolled in a specific degree program (no unclassified students under Bologna Process requirements)
3. **Lattice Structure - Shared Subclass**:
   - TEACHING_ASSISTANT appears at the bottom with two subset symbol (⊆) connections:
     - One line connects from DOCTORAL_STUDENT to TEACHING_ASSISTANT
     - One line connects from INSTRUCTOR to TEACHING_ASSISTANT
   - This dual connection indicates multiple inheritance: TEACHING_ASSISTANT simultaneously inherits from both DOCTORAL_STUDENT and INSTRUCTOR
   - No specialization circle connects these paths because TEACHING_ASSISTANT is not a specialization of either parent; it's a shared subclass combining both roles
4. **Multiple Inheritance Paths**:
   - Path 1: PERSON → STUDENT → DOCTORAL_STUDENT → TEACHING_ASSISTANT
   - Path 2: PERSON → INSTRUCTOR → TEACHING_ASSISTANT
   - Both paths lead to PERSON, creating the diamond inheritance pattern discussed in Section 5.5.2.3

Interpreting the Complete Structure:

This diagram captures the organizational reality that doctoral students who teach hold dual roles:

- As DOCTORAL_STUDENT: They inherit student attributes (Student_ID, ECTS_completed, Dissertation_topic, Defense_committee, Expected_defense_date) through the student hierarchy

- As INSTRUCTOR: They inherit teaching attributes (Instructor_ID, Salary, Hire_date, Department_ID, Course_taught, Teaching_schedule, Student_evaluation_average) through the instructor path
- As TEACHING_ASSISTANT: They have specialized attributes combining both roles (Hours_per_week, Semester_appointment, TA_training_completed, Supervisor_ID)

The notation precisely represents the constraint that teaching assistants must be doctoral students (not bachelor's or master's students) while simultaneously functioning as instructors. This semantic precision, which shows exactly which student type can be a teaching assistant, would be impossible to represent accurately using only hierarchical structures.

Contrast with Pure Hierarchy:

Compare this lattice structure (Figure 5.23) with the pure hierarchy (Figure 5.10):

- Figure 5.10: Each entity has exactly one parent (tree structure, single inheritance)
- Figure 5.23: TEACHING_ASSISTANT has two parents (lattice structure, multiple inheritance)

The lattice structure accurately models organizational reality: teaching assistants genuinely fulfill both the roles of doctoral students and instructors simultaneously. Forcing this into a pure hierarchy would sacrifice semantic accuracy; they are not "primarily" students with teaching duties, nor "primarily" instructors who happen to study, but authentically both.

This notation framework allows database designers to accurately represent complex organizational relationships that are difficult to classify hierarchically, ensuring EER models properly reflect business needs.

5.6.7 Chen Notation Best Practices

Effective EER diagrams require consistent use of Chen notation conventions (Hristoski & Dimovski, 2022; Elmasri & Navathe, 2016). This subsection outlines essential guidelines for creating clear, professional diagrams that accurately communicate database structure.

5.6.7.1 Essential Notation Guidelines

Apply these principles consistently throughout your EER diagrams:

1. **Explicit Constraint Specification**: Mark every specialization circle with (d) or (o), and use double lines (‖) for total participation or single lines (|) for partial participation. Unmarked constraints introduce ambiguity, undermining the diagram's precision.
2. **Complete Subset Symbols**: Place subset symbols (⊆) on all connections from specialization circles to subclasses. These symbols explicitly indicate the fundamental subclass ⊆ superclass relationship.
3. **Hierarchical Attribute Placement**: Position inherited attributes near superclass entities and local attributes near their respective subclasses. This spatial organization helps readers understand inheritance patterns at a glance.

4. **Level-Based Spatial Organization**: Arrange entities at the same hierarchy level with horizontal alignment. This alignment clarifies the hierarchical structure and makes multi-level specializations easier to interpret.
5. **Line Crossing Minimization**: Plan diagram layout to minimize line crossings between entities and specialization structures. Strategic placement of entities significantly improves diagram readability.

5.7 Complete EER Case Studies: University and Company Systems

Having established EER modeling concepts, constraints, hierarchies, and notation conventions in Sections 5.1-5.6, we now systematically apply these techniques to create complete EER diagrams for our case studies. These detailed models extend the basic ER designs from Chapter 4 (Figures 4.62 and 4.88) by including specialization hierarchies, constraint specifications, and attribute inheritance patterns.

5.7.1 Extended University System: Complete EER Model

5.7.1.1 EER Extension Requirements

Building on our University ER diagram from Chapter 4 (Figure 4.62), we address additional requirements that demand EER modeling features:

Bologna Process Academic Structure: The university implements the three-tier Bologna system (European Commission, 2015) with distinct student categories: Bachelor's students (180-240 ECTS over 3-4 years), Master's students (60-120 ECTS over 1-2 years), and Doctoral students (180 ECTS over 3 years). Each category has different thesis requirements: Bachelor's thesis with 2-member committees, Master's thesis with 3-member committees, and Doctoral dissertation with 5-member committees.

Academic Rank Classification: Faculty members are classified by academic rank (Assistant Professor, Associate Professor, Full Professor), each with distinct teaching loads, research expectations, and administrative responsibilities.

Dual-Status Teaching Assistants: Some doctoral students serve as teaching assistants, holding both STUDENT and INSTRUCTOR status. They remain enrolled in doctoral programs while teaching undergraduate courses and receiving instructor compensation.

These requirements necessitate EER modeling capabilities beyond those provided by basic ER diagrams.

5.7.1.2 Systematic Modeling Process

Step 1: Identify Specialization Opportunities

Requirements analysis reveals the following entities requiring specialization:

Root Entity:

- **PERSON**: Root superclass for all university individuals, with five primary role classifications

Student Hierarchy:

- **STUDENT**: Specialization by academic level (Bachelor, Master, Doctoral)

Faculty and Staff Entities:

- **PROFESSOR**: Specialization by academic rank (Assistant, Associate, Full)
- **INSTRUCTOR**: Teaching staff, including teaching assistants, lecturers, and adjuncts
- **STAFF**: Administrative and support personnel
- **ALUMNUS**: Former students who graduated from the university

Shared Subclass:

- **TEACHING_ASSISTANT**: Shared subclass inheriting from both DOCTORAL_STUDENT and INSTRUCTOR

Step 2: Define Hierarchies and Lattice Structures

The University system contains two independent hierarchies for students and professors, plus one lattice structure for teaching assistants:

Hierarchy 1: Student Academic Levels

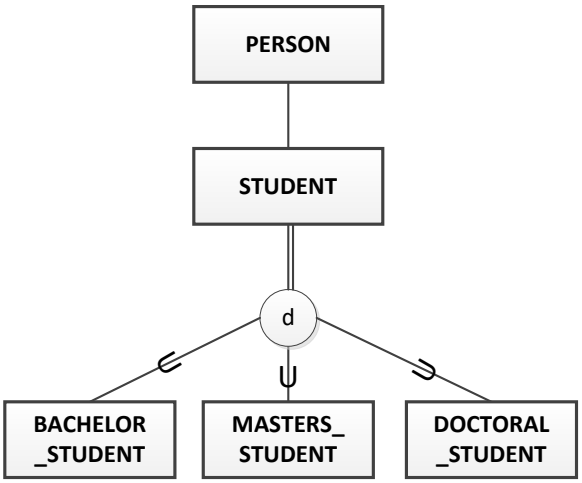


Figure 5.24: Hierarchy 1: Student Academic Levels

Hierarchy 2: Professor Academic Ranks

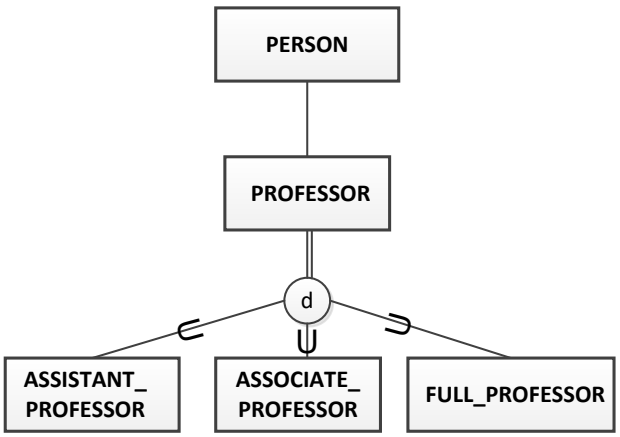


Figure 5.25: Hierarchy 2: Professor Academic Ranks

Lattice: Teaching Assistant Multiple Inheritance

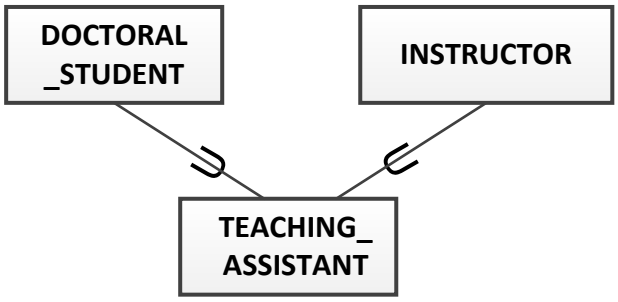


Figure 5.26: Lattice: Teaching Assistant Multiple Inheritance

Step 3: Assign Specialized Attributes

Following the attribute placement principles from Section 5.3.1.2, we assign attributes hierarchically from general to specific:

PERSON (Root Superclass):

- SSN (Key), Birth_date, Email
- Name (Composite: First_name, Last_name)
- Address (Composite: Street, City, Country, Zip_code)

STUDENT Level (Intermediate):

- Student_ID (Additional key), Enrollment_date, ECTS_completed
- Inherits: All PERSON attributes

BACHELOR_STUDENT (Specialized):

- Study_year, Major_field, Total_ECTS_required {180, 240}, Defense_committee {2 members}
- Inherits: All STUDENT and PERSON attributes

MASTERS_STUDENT (Specialized):

- Total_ECTS_required {60, 120}, Master_thesis_topic, Defense_committee {3 members}
- Inherits: All STUDENT and PERSON attributes

DOCTORAL_STUDENT (Specialized):

- Dissertation_topic, Defense_committee {5 members}, Expected_defense_date
- Inherits: All STUDENT and PERSON attributes

PROFESSOR Level (Intermediate):

- Professor_ID (Key), Hire_date, Office, Academic_title
- Inherits: All PERSON attributes

ASSISTANT_PROFESSOR (Specialized):

- Employment_type, Years_in_current_rank

- Inherits: All PROFESSOR and PERSON attributes

ASSOCIATE_PROFESSOR (Specialized):

- Research_publications, Years_in_current_rank
- Inherits: All PROFESSOR and PERSON attributes

FULL_PROFESSOR (Specialized):

- Years_as_full_professor, H_index
- Inherits: All PROFESSOR and PERSON attributes

INSTRUCTOR Level (Intermediate):

- Instructor_ID (Key), Salary, Hire_date, Department_ID
- Teaching_schedule, Course_taught, Student_evaluation_average
- Inherits: All PERSON attributes

STAFF Level (Intermediate):

- Staff_ID (Key), Hire_date, Department_assigned
- Job_title, Employment_status (Full-time, Part-time, Contract)
- Office_location
- Inherits: All PERSON attributes

ALUMNUS Level (Intermediate):

- Alumni_ID (Key), Graduation_date, Degree_earned, Current_occupation
- Inherits: All PERSON attributes

TEACHING_ASSISTANT (Shared Subclass):

- Inherits from both DOCTORAL_STUDENT and INSTRUCTOR paths
- From DOCTORAL_STUDENT path: Student_ID, Enrollment_date, ECTS_completed, Dissertation_topic, Defense_committee, Expected_defense_date
- From INSTRUCTOR path: Instructor_ID, Salary, Hire_date, Department_ID, Teaching_schedule, Course_taught, Student_evaluation_average
- Local attributes: Hours_per_week, Semester_appointment, TA_training_completed, Supervisor_ID

Domain Constraint Notation in Figure 5.27:

Several attributes use domain constraint notation (curly brackets) to specify allowed values:

- Total_ECTS_required {180, 240}: Bachelor students complete either 180 or 240 ECTS credits
- Total_ECTS_required {60, 120}: Master's students complete either 60 or 120 ECTS credits

- Defense_committee {2 members}, {3 members}, {5 members}: Specifies exact committee sizes for each degree level
- Employment_status {Full-time, Part-time, Contract}: Enumerated employment types for staff

This notation, introduced in Section 5.6.3, translates directly to CHECK constraints in SQL DDL (Chapter 9).

Step 4: Specify Constraints

STUDENT Specialization: (d, t) - Disjoint, Total

- **Disjoint**: Students cannot simultaneously pursue multiple degree levels (no concurrent Bachelor's-Master's enrollment under the Bologna Process structure)
- **Total**: Every student must be enrolled in a specific degree program with defined ECTS requirements
- **Business Rule**: Bologna Process (European Commission, 2015) mandates classification for credit tracking and degree requirements

PROFESSOR Specialization: (d, t) - Disjoint, Total

- **Disjoint**: Professors hold one academic rank at any given time
- **Total**: Every professor must have an assigned rank for salary determination and responsibility allocation
- **Business Rule**: Academic promotions move professors between ranks sequentially (Assistant → Associate → Full)

PERSON Specialization: (o, t) - Overlapping, Total

- **Overlapping**: Individuals can hold multiple roles simultaneously (doctoral students who teach are both STUDENT and INSTRUCTOR; professors who graduated from the university are both PROFESSOR and ALUMNUS; staff members pursuing degrees are both STAFF and STUDENT)
- **Total**: Every person in the university system must have at least one institutional role (as STUDENT, PROFESSOR, INSTRUCTOR, STAFF, or ALUMNUS)
- **Business Rule**: The university only tracks people with active or past institutional relationships; dual or multiple roles are common in academic environments

5.7.1.3 Validation Through Business Scenarios

The following scenarios demonstrate how the specialization hierarchies integrate with the base ER relationships from Chapter 4 (Figure 4.62). Relationships such as enrolls_in, belongs_to, advised_by, and teaches are inherited from the original ER model and are part of the complete EER diagram (Figure 5.27). The scenarios illustrate how specialized entities maintain all relationships defined for their superclasses through inheritance.

Scenario 1: Bachelor Student Enrollment and Progression

Student S001 (Sarah Smith) enrolls as BACHELOR_STUDENT in Computer Science:

- Inherits PERSON attributes: SSN, Name (First_name: Sarah, Last_name: Smith), Birth_date (25.08.1995), Address (Jane Sandanski 32, Ohrid, Macedonia, 6000), Email (sarah.smith@university.edu)
- Receives STUDENT attributes: Student_ID: S001, Enrollment_date (01.10.2025), ECTS_Completed
- Assigned BACHELOR_STUDENT attributes: Major_field: Computer Science, Study_year, Total_ECTS_required: 240, Defense_committee (2 members)
- Participates in inherited relationships: enrolls_in CS303, belongs_to Faculty F001 (Computer Science)

  Note: These relationships are defined in the base ER model (Chapter 4, Figure 4.62) and inherited by all STUDENT entities, including specialized BACHELOR_STUDENT entities
- Satisfies constraints: (d, t) - belongs to exactly one academic level

Scenario 2: Doctoral Student as Teaching Assistant

A doctoral student assumes teaching assistant responsibilities, demonstrating the lattice structure:

- Exists as DOCTORAL_STUDENT through inheritance: PERSON → STUDENT → DOCTORAL_STUDENT
- Simultaneously exists as TEACHING_ASSISTANT (shared subclass)
- Multiple inheritance accumulates attributes from both paths:
  - Student path: Student_ID, Enrollment_date, ECTS_Completed, Dissertation_topic, Defense_committee, Expected_defense_date
  - Instructor path: Instructor_ID, Salary, Hire_date, Department_ID, Teaching_schedule, Course_taught, Student_evaluation_average
  - Local TA attributes: Hours_per_week, Semester_appointment, TA_training_completed, Supervisor_ID
- Participates in student relationships: enrolls_in doctoral seminars, supervised_by dissertation committee
- Participates in instructor relationships: teaches undergraduate courses (CS101, CS201)

Scenario 3: Professor Promotion Through Academic Ranks

Professor P006 (Richard Johnson) receives promotion from Assistant Professor to Associate Professor:

- Initial state: PERSON → PROFESSOR → ASSISTANT_PROFESSOR
- Has ASSISTANT_PROFESSOR attributes: Employment_type, Years_in_current_rank
- Upon promotion: Membership changes from ASSISTANT_PROFESSOR to ASSOCIATE_PROFESSOR

- Retains inherited attributes: from PERSON (SSN, Name (First_name: Richard, Last_name: Johnson), Birth_date, Address, Email: richard.johnson@university.edu), from PROFESSOR (Professor_ID: P006, Office: E206, Hire_date: 02.09.2022) remain unchanged
- Gains ASSOCIATE_PROFESSOR attributes: Research_publications, Years_in_current_rank
- Loses ASSISTANT_PROFESSOR-specific attributes: Employment_type
- Constraint maintained: (d, t) - belongs to exactly one rank at any given time

Scenario 4: Alumnus Maintaining Multiple Roles

Alumnus A015 (Elena Mihajlovska) graduated with a Bachelor's degree in 2020 and now works as administrative staff while pursuing a Master's degree:

- Primary role as ALUMNUS: Alumni_ID: A015, Graduation_date: 15.07.2020, Degree_earned: Bachelor of Computer Science
- Simultaneously STAFF: Staff_ID: ST012, Job_title: IT Support Specialist, Department_assigned: IT Services
- Simultaneously STUDENT (MASTERS_STUDENT): Student_ID: S125, enrolled in Master's program in Information Systems
- Demonstrates overlapping participation: One person holds three roles (ALUMNUS, STAFF, MASTERS_STUDENT)
- All roles share PERSON attributes (SSN, Name, Birth_date, Address, Email) without duplication

5.7.1.4 Complete Model Summary

Hierarchical Structure:

- Root: PERSON
- Level 2: STUDENT, PROFESSOR, INSTRUCTOR, STAFF, ALUMNUS
- Level 3: BACHELOR_STUDENT, MASTERS_STUDENT, DOCTORAL_STUDENT (student hierarchy); ASSISTANT_PROFESSOR, ASSOCIATE_PROFESSOR, FULL_PROFESSOR (professor hierarchy)
- Shared Subclass: TEACHING_ASSISTANT (inherits from DOCTORAL_STUDENT and INSTRUCTOR)

Inheritance Paths:

- Single inheritance: BACHELOR_STUDENT inherits through PERSON → STUDENT → BACHELOR_STUDENT
- Single inheritance: ASSOCIATE_PROFESSOR inherits through PERSON → PROFESSOR → ASSOCIATE_PROFESSOR
- Multiple inheritance: TEACHING_ASSISTANT inherits through both PERSON → STUDENT → DOCTORAL_STUDENT and PERSON → INSTRUCTOR

Constraint Summary:

- (o, t): PERSON → {STUDENT, PROFESSOR, INSTRUCTOR, STAFF, ALUMNUS}
- (d, t): STUDENT → {BACHELOR_STUDENT, MASTERS_STUDENT, DOCTORAL_STUDENT}
- (d, t): PROFESSOR → {ASSISTANT_PROFESSOR, ASSOCIATE_PROFESSOR, FULL_PROFESSOR}

Attribute Distribution:

- PERSON level: 5 attributes (SSN, Name, Birth_date, Address, Email) - inherited by all subclasses
- Intermediate levels: 3-5 attributes per entity
- Specialized levels: 2-4 attributes per entity
- Total specialization depth: 3 levels maximum (PERSON → STUDENT → BACHELOR_STUDENT)

Figure 5.27 displays the complete University EER model with all hierarchies and shared subclasses.

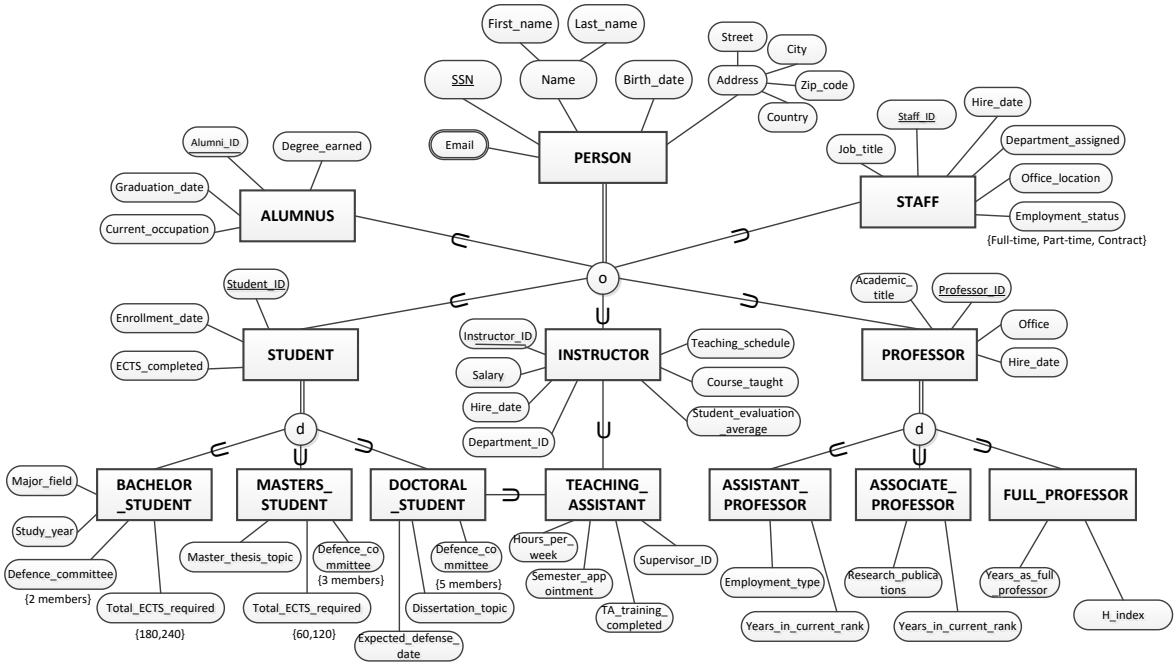


Figure 5.27: Complete University EER Model with All Hierarchies and Shared Subclass

5.7.1.5 Integrated EER and ER Model

Figure 5.28 demonstrates how the specialization hierarchies from Figure 5.27 integrate with the base ER relationships from Chapter 4 (Figure 4.62), creating a complete operational database model.

Integration of Specialization with Relationships:

The integrated model shows how specialized entities maintain all relationship capabilities of their superclasses:

STUDENT Entity Relationships:

- enrolls_in: All STUDENT entities (BACHELOR_STUDENT, MASTERS_STUDENT, DOCTORAL_STUDENT) enroll in courses with grades and enrollment dates
- belongs_to: All students belong to a specific FACULTY (Computer Science, Electrical Engineering, etc.)
- attends: Students attend CLASS_SESSION entities for their enrolled courses

PROFESSOR Entity Relationships:

- teaches: All PROFESSOR entities (regardless of rank: ASSISTANT_PROFESSOR, ASSOCIATE_PROFESSOR, FULL_PROFESSOR) teach courses
- affiliated_with: Professors are affiliated with specific faculties
- heads: Senior professors may head faculties (administrative relationship)
- assignment: Ternary relationship - professors assign specific textbooks to their courses

COURSE Entity Relationships:

- prerequisite: Self-referencing relationship showing course dependencies
- conducts: Links courses to their scheduled CLASS_SESSION entities
- assignment: Ternary relationship connecting courses to required textbooks as assigned by professors
- offers: FACULTY entities offer specific courses

TEACHING_ASSISTANT Multiple Inheritance Integration:

The TEACHING_ASSISTANT shared subclass demonstrates complex relationship inheritance:

- Through DOCTORAL_STUDENT path: Inherits enrolls_in (for doctoral seminars), belongs_to (faculty affiliation)
- Through the INSTRUCTOR path: Inherits teaches relationship (for undergraduate courses they teach)
- Direct relationships: All doctoral students have supervised_by relationship with dissertation supervisors (professors); teaching assistants may also have assists_in relationships for supporting professors in large lecture courses

Attribute and Relationship Inheritance Example:

Consider a DOCTORAL_STUDENT entity (D001) who is also a TEACHING_ASSISTANT:

From PERSON (via inheritance):

- SSN: 123-45-6789
- Name: Ana Petrovska
- Email: ana.petrovska@university.edu

From STUDENT (via DOCTORAL_STUDENT):

- Student_ID: D001
- Enrollment_date: 01.10.2022
- enrolls_in: DB601 (Doctoral Database Seminar), DB602 (Research Methods)
- belongs_to: Faculty F001 (Computer Science)

From DOCTORAL_STUDENT:

- Dissertation_topic: "Graph Databases for Biological Networks"
- Defense_committee: 5 members
- Expected_defense_date: 15.06.2026

From INSTRUCTOR (via TEACHING_ASSISTANT):

- Instructor_ID: I025
- teaches: CS101 (Introduction to Databases - undergraduate course)
- Salary: Teaching assistant stipend

From TEACHING_ASSISTANT (local):

- Hours_per_week: 20
- Semester_appointment: Fall 2025, Spring 2026
- Supervisor_ID: P006 (Professor Richard Johnson)

This entity simultaneously participates in student activities (enrolling in doctoral seminars, working toward a dissertation) and instructor activities (teaching undergraduate courses), demonstrating how multiple inheritance enables accurate modeling of complex organizational roles.

Complete Model Capabilities:

The integrated model in Figure 5.28 supports comprehensive university operations:

- Academic Administration: Track students through degree programs with specialized requirements (ECTS credits, defense committees) while managing course enrollments and faculty assignments
- Teaching Operations: Manage course offerings, prerequisite chains, class sessions, and textbook assignments across all faculty ranks
- Dual-Role Support: Handle teaching assistants who simultaneously function as students and instructors with appropriate compensation and responsibilities
- Faculty Management: Track professor ranks, research output (publications, H-index), and administrative roles (department heads)
- Alumni Relations: Maintain connections with graduates who may return as staff or students, enabling comprehensive lifecycle tracking

This integration demonstrates a fundamental database design principle: specialization hierarchies extend existing data models rather than replacing them. The EER model adds semantic precision (students have specific degree levels, professors have specific ranks) while preserving all operational relationships (course enrollment, teaching assignments, faculty affiliation) that make the database functional.

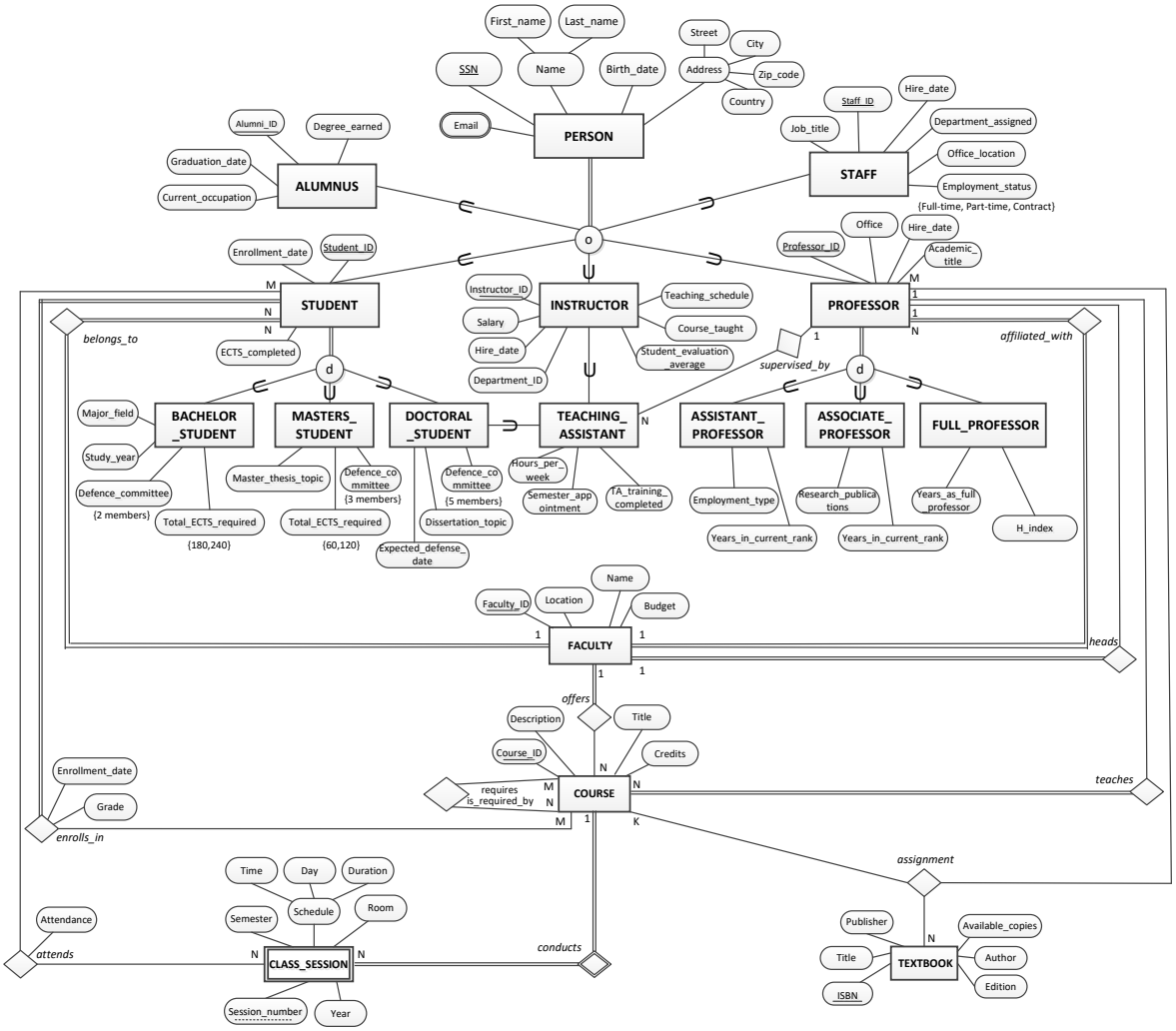


Figure 5.28: Complete Integrated University EER Model Combining Specialization Hierarchies with Base ER Relationships

5.7.2 Extended Company System: Complete EER Model

5.7.2.1 EER Extension Requirements

Building on our Company ER diagram from Chapter 4 (Figure 4.88), we include requirements that show two independent specializations:

Job Function Classification: The company categorizes employees by primary job function. For example, *managers* have budget authority and staff supervision responsibilities, such as employee E1 (Zoran Pejovski) managing Sales D1, E3 (Marija Naumoska) managing Marketing D2, and E5 (Naum Stojanoski) managing Human Resources D3. *Sales employees* work with customer territories and earn commissions. *Warehouse employees* work in physical inventory locations with shift schedules. *Administrative employees* support departmental operations.

However, not all employees belong to specialized categories. There are many general employees who perform standard functions without specialized classification, including E2 (Nikola Delcevski), E4 (Anita Toshevska), E6 (Hristina Miloseska), E7 (Dimitar Tocko), and E8 (Bijana Osmani).

Employment Type Classification: Employees are also classified by employment type: *full-time* employees with annual salaries and standard benefits, *part-time* employees with reduced work hours and limited benefits, or *contract employees* with fixed-term agreements and project-based compensation.

These requirements require modeling **multiple independent specializations** of the same EMPLOYEE superclass, which is a distinct EER pattern from the University's lattice structure.

5.7.2.2 Systematic Modeling Process

Step 1: Identify Multiple Specialization Dimensions

Analysis reveals EMPLOYEE is involved in two separate specializations:

- **By Job Function**: Four subclasses are identified: MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, ADMIN_EMPLOYEE.
- **By Employment Type**: Three subclasses are identified: FULL_TIME_EMPLOYEE, PART_TIME_EMPLOYEE, CONTRACT_EMPLOYEE

Step 2: Define Independent Specialization Hierarchies

The Company system contains two independent specialization hierarchies:

Specialization 1: Job Function (Primary Classification)

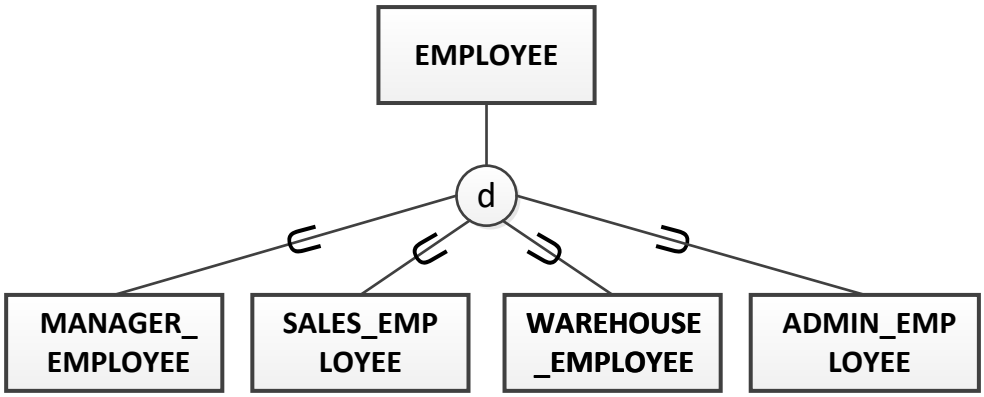


Figure 5.29: Job Function (Primary Classification)

Specialization 2: Employment Type (Secondary Classification)

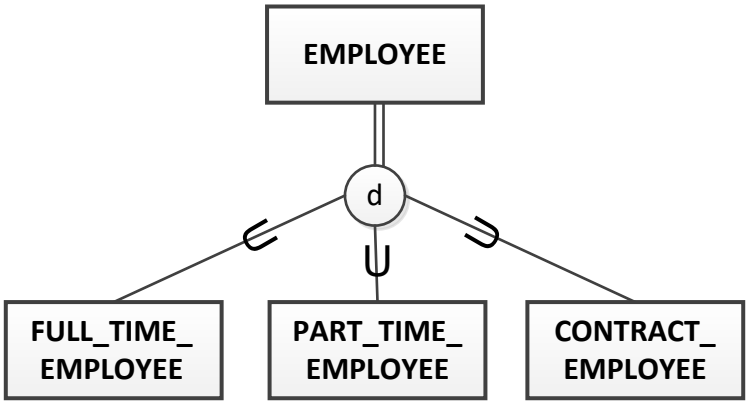


Figure 5.30: Employment Type (Secondary Classification)

Step 3: Assign Specialized Attributes

Following the attribute placement principles from Section 5.3.1.2, we assign attributes hierarchically from general to specific.

EMPLOYEE (Superclass):

- Employee_ID (Key), Email, Salary, Hire_date,
- Name (Composite: First_name, Last_name)
- Address (Composite: Street, City, Country, Zip_code)
- Phone (Multivalued)
- Job_function, Employment_type
- Department_ID (FK), Warehouse_ID (FK, optional)

Job Function Specializations:

MANAGER_EMPLOYEE:

- Manages_Department_ID (FK), Management_level, Budget_authority, Strategic_planning_responsibility, Leadership_training_complete
- Examples: E1 (manages D1, Salary: 60000), E3 (manages D2, Salary: 50000), E5 (manages D3, Salary: 40000)

SALES_EMPLOYEE:

- Sales_territory, Commission_rate, Monthly_sales_quota, Customer_portfolio_size

WAREHOUSE_EMPLOYEE:

- Primary_warehouse_ID (FK), Shift_schedule, Physical_location, Inventory_management_system_trained

ADMIN_EMPLOYEE:

- Office_location, Administrative_level, Department_management_responsibility

Employment Type Specializations:

FULL_TIME_EMPLOYEE:

- Benefits_package_level, Retirement_contribution_percentage, Vacation_days_per_year,

PART_TIME_EMPLOYEE:

- Hourly_rate, Maximum_hours_per_week, Schedule_flexibility

CONTRACT_EMPLOYEE:

- Contract_start_date, Contract_end_date, Contracting_agency_or_independent, Payment_rate

Step 4: Specify Constraints

Job Function Specialization: (d, p) - Disjoint, Partial

This specialization is *partial* because not all employees are specialized; many remain general employees.

- **Disjoint**: An employee has one primary job function (E1 is MANAGER, not simultaneously SALES)
- **Partial**: Many employees (E2, E4, E6, E7, E8) are general employees without specialized classification
- **Business Rule**: Not all positions require specialized classification; many employees perform standard operational duties

Employment Type Specialization: (d, t) - Disjoint, Total

- **Disjoint**: An employee has one employment type (cannot be both full-time and part-time)
- **Total**: Every employee must have an employment type for payroll and benefits administration
- **Business Rule**: HR requires employment type classification for all employees to determine compensation structure

5.7.2.3 Validation Through Business Scenarios

Scenario 1: Manager with Dual Specialization Membership

Employee E1 (Zoran Pejovski) serves as Sales Department manager:

- EMPLOYEE attributes: Employee_ID: E1, Name: Zoran Pejovski, Salary: 60000, Hire_date: 15.01.2020, Department_ID: D1
- MANAGER_EMPLOYEE attributes: Manages_Dept_ID: D1, Budget_authority
- FULL_TIME_EMPLOYEE attributes: Benefits_package_level, Vacation_days_per_year: 26
- Participates in two independent specializations simultaneously:
  - Job function: E1 ∈ MANAGER_EMPLOYEE
  - Employment type: E1 ∈ FULL_TIME_EMPLOYEE
- Each specialization has independent constraint rules

Scenario 2: General Employee (Partial Specialization)

Employee E2 (Nikola Delcevski) works in the Sales department without a specialized role:

- EMPLOYEE attributes: Employee_ID: E2, Department_ID: D1, Salary: 55000
- Does NOT belong to any job function subclass (demonstrates partial specialization)
- DOES belong to FULL_TIME_EMPLOYEE
- Satisfies (d, p) constraint: Unspecialized employees are explicitly permitted

- Demonstrates that partial specialization accommodates a general workforce

Scenario 3: Multiple Independent Specializations

The same employee simultaneously belongs to subclasses from different specializations:

- E1 belongs to both MANAGER_EMPLOYEE (job function) and FULL_TIME_EMPLOYEE (employment type)
- E2 belongs to no job function subclass but belongs to FULL_TIME_EMPLOYEE
- Possible combinations: Up to 4 × 3 = 12 theoretical combinations (though not all exist in practice)
- Each specialization dimension operates independently with its own constraints

5.7.2.4 Complete EER Model Summary

The Company EER model demonstrates an important pattern distinct from the University lattice:

Key Distinction:

- **University model**: Lattice with shared subclass (TEACHING_ASSISTANT inherits from multiple parents through multiple paths)
- **Company model**: Single superclass participating in multiple independent specializations (different classification dimensions)

The EER diagram in Figure 5.31 illustrates both independent specialization hierarchies, demonstrating how a single superclass can participate in multiple separate classification dimensions: Job Function Specialization and Employment Type Specialization.

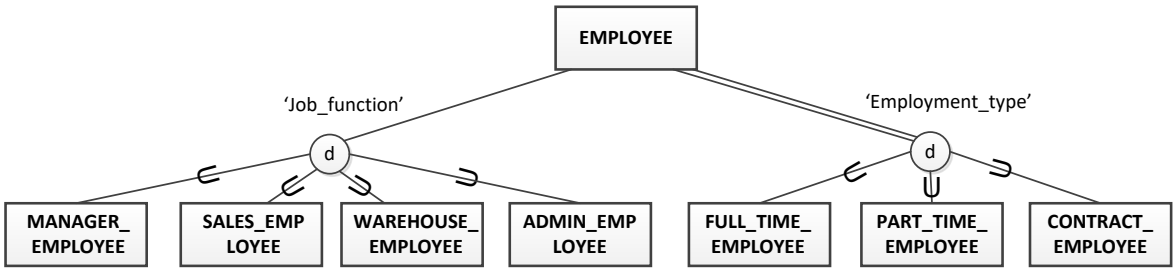


Figure 5.31: Independent Specializations Structure

Each employee potentially belongs to:

- Zero or one job function subclass (partial)
- Exactly one employment type subclass (total)

Constraint Summary:

- (d, p): EMPLOYEE → {MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, ADMIN_EMPLOYEE} by job function
- (d, t): EMPLOYEE → {FULL_TIME_EMPLOYEE, PART_TIME_EMPLOYEE, CONTRACT_EMPLOYEE} by employment type

Total Entities in Model:

- Superclass: EMPLOYEE
- Job function specialization: 4 subclasses (MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, ADMIN_EMPLOYEE)
- Employment type specialization: 3 subclasses (FULL_TIME_EMPLOYEE, PART_TIME_EMPLOYEE, CONTRACT_EMPLOYEE)

Figure 5.32 displays the complete Company EER model with both independent specialization hierarchies.

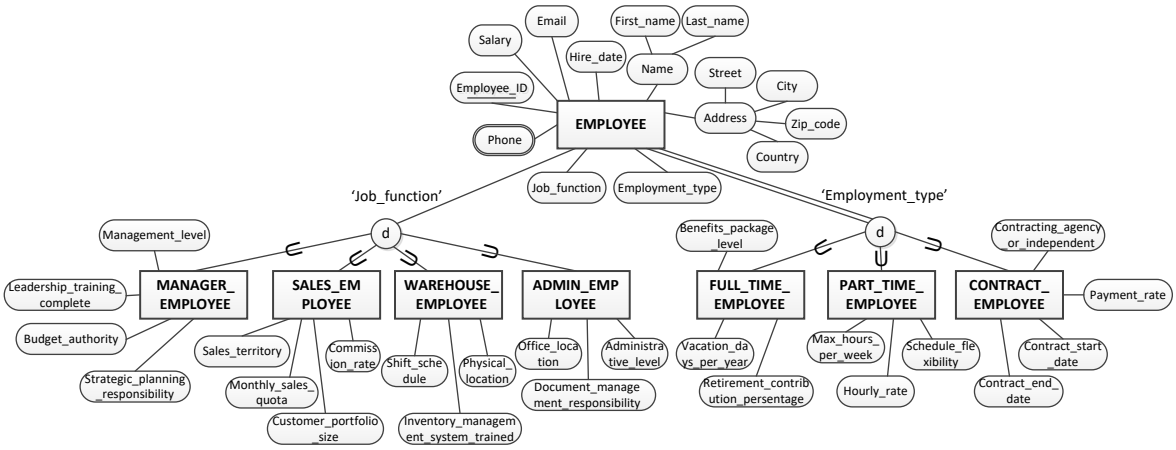


Figure 5.32: The Complete Company EER diagram

5.7.2.5 Integrated EER and ER Model

Figure 5.33 demonstrates how the independent specialization hierarchies from Figure 5.32 integrate with the base ER relationships from Chapter 4 (Figure 4.88) to create a complete operational database model.

Integration of Specialization with Relationships

The integrated model shows how specialized employees maintain all relationship capabilities of the EMPLOYEE superclass while adding specialized attributes:

EMPLOYEE Entity Relationships

- **works_for:** All EMPLOYEE entities (regardless of specialization) work for specific departments
- **processes:** Employees process customer orders
- **manages:** MANAGER_EMPLOYEE entities manage departments (1:1 relationship - each manager manages exactly one department)
- **works_at:** WAREHOUSE_EMPLOYEE entities work at specific warehouses (optional relationship for employees assigned to warehouse duties)

DEPARTMENT Entity Relationships

- **manages:** Each department is managed by one MANAGER_EMPLOYEE (inverse of employee manages relationship)

- **works_for:** Multiple employees work for each department
- **oversees:** Departments oversee warehouse operations

ORDER and CUSTOMER Relationships

- **places:** Customers place orders with order dates and status tracking
- **processes:** Employees process orders (handling, fulfillment, shipping)
- **contains:** Orders contain multiple ORDER_LINE items with quantities and pricing
- **appears_in:** Products appear in order lines with unit prices and discounts

PRODUCT and WAREHOUSE Relationships

- **stored_in:** Products are stored in warehouses with Quantity_on_hand and Reorder_level tracking
- **supplies:** Suppliers supply products with supply dates and unit costs
- **appears_in:** Products appear in order lines for customer orders

Independent Specialization Integration Example

Consider Employee E1 (Zoran Pejovski), who simultaneously belongs to two independent specializations:

From EMPLOYEE (superclass)

- Employee_ID: E1
- Name: Zoran Pejovski
- Email: zoran.pejovski@email.com
- Salary: 60000
- Hire_date: 15.01.2020
- Department_ID: D1 (Sales)
- **works_for:** DEPARTMENT D1
- **processes:** ORDER entities (manages order processing for Sales department)

From MANAGER_EMPLOYEE (Job Function specialization)

- Manages_Department_ID: D1
- Management_level: Department_Level
- Budget_authority: 500000
- Strategic_planning_responsibility: Yes
- Leadership_training_complete: Yes
- **manages:** DEPARTMENT D1 (bidirectional relationship with works_for)

From FULL_TIME_EMPLOYEE (Employment Type specialization)

- Benefits_package_level: Premium

- Vacation_days_per_year: 26
- Retirement_contribution_percentage: 18,8%

This demonstrates how an employee can simultaneously participate in two independent specialization dimensions:

- **Job Function dimension:** E1 ∈ MANAGER_EMPLOYEE (specialized managerial role)
- **Employment Type dimension:** E1 ∈ FULL_TIME_EMPLOYEE (full-time employment classification)

Each specialization operates independently with its own constraints:

- **Job Function (d, p):** E1 is a manager, not a sales, warehouse, or admin employee (disjoint); some employees have no specialized job function (partial)
- **Employment Type (d, t):** E1 is full-time, not part-time or contract (disjoint); every employee must have an employment type (total)

General Employee Example

Employee E2 (Nikola Delcevski) demonstrates partial participation in job function specialization:

From EMPLOYEE

- Employee_ID: E2
- Name: Nikola Delcevski
- Salary: 55000
- Department_ID: D1 (Sales)
- Warehouse_ID: W1 (Skopje)
- **works_for:** DEPARTMENT D1
- **works_at:** WAREHOUSE W1
- **processes:** ORDER entities

From Employment Type (required by total constraint)

- **Belongs to:** FULL_TIME_EMPLOYEE
- Benefits_package_level: Standard
- Vacation_days_per_year: 22

From Job Function (optional due to partial constraint)

- **Does NOT belong** to any specialized job function subclass (MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, or ADMIN_EMPLOYEE)
- Remains a general employee performing standard warehouse and sales support duties

This illustrates the **(d, p)** constraint on Job Function specialization: E2 participates in employment-type classification (total) but not in job-function classification (partial). E2 performs valuable work without requiring specialized job function modeling.

Warehouse Employee Example

Employee E6 (Hristina Miloseska) demonstrates specialized job function:

From EMPLOYEE

- Employee_ID: E6
- Salary: 35000
- Department_ID: D3 (Human Resources)
- Warehouse_ID: W3 (Ohrid)
- **works_for:** DEPARTMENT D3
- **works_at:** WAREHOUSE W3

From WAREHOUSE_EMPLOYEE (Job Function)

- Primary_warehouse_ID: W3
- Shift_schedule: Morning
- Physical_location: Ohrid
- Inventory_management_system_trained: Yes

From FULL_TIME_EMPLOYEE (Employment Type)

- Benefits_package_level: Basic
- Vacation_days_per_year: 20

E6 demonstrates how employees can have specialized job functions (warehouse operations) while maintaining a standard employment-type classification.

Complete Model Capabilities

The integrated model shown in Figure 5.33 supports comprehensive company operations:

- **Employee Management:** Track employees through dual independent classifications (job function and employment type) while managing department assignments, warehouse assignments, and payroll
- **Order Processing:** Manage customer orders from placement through fulfillment, tracking which employees process which orders
- **Inventory Control:** Track product storage across multiple warehouses with quantity monitoring, reorder levels, and warehouse employee assignments
- **Department Operations:** Manage departmental budgets, locations, and manager assignments with clear reporting structures
- **Supplier Management:** Track product suppliers, supply relationships, supply dates, and unit costs

- **Warehouse Operations:** Coordinate warehouse employees, product storage, capacity management, and inventory tracking
- **Customer Relations:** Maintain customer information, order history, and product preferences

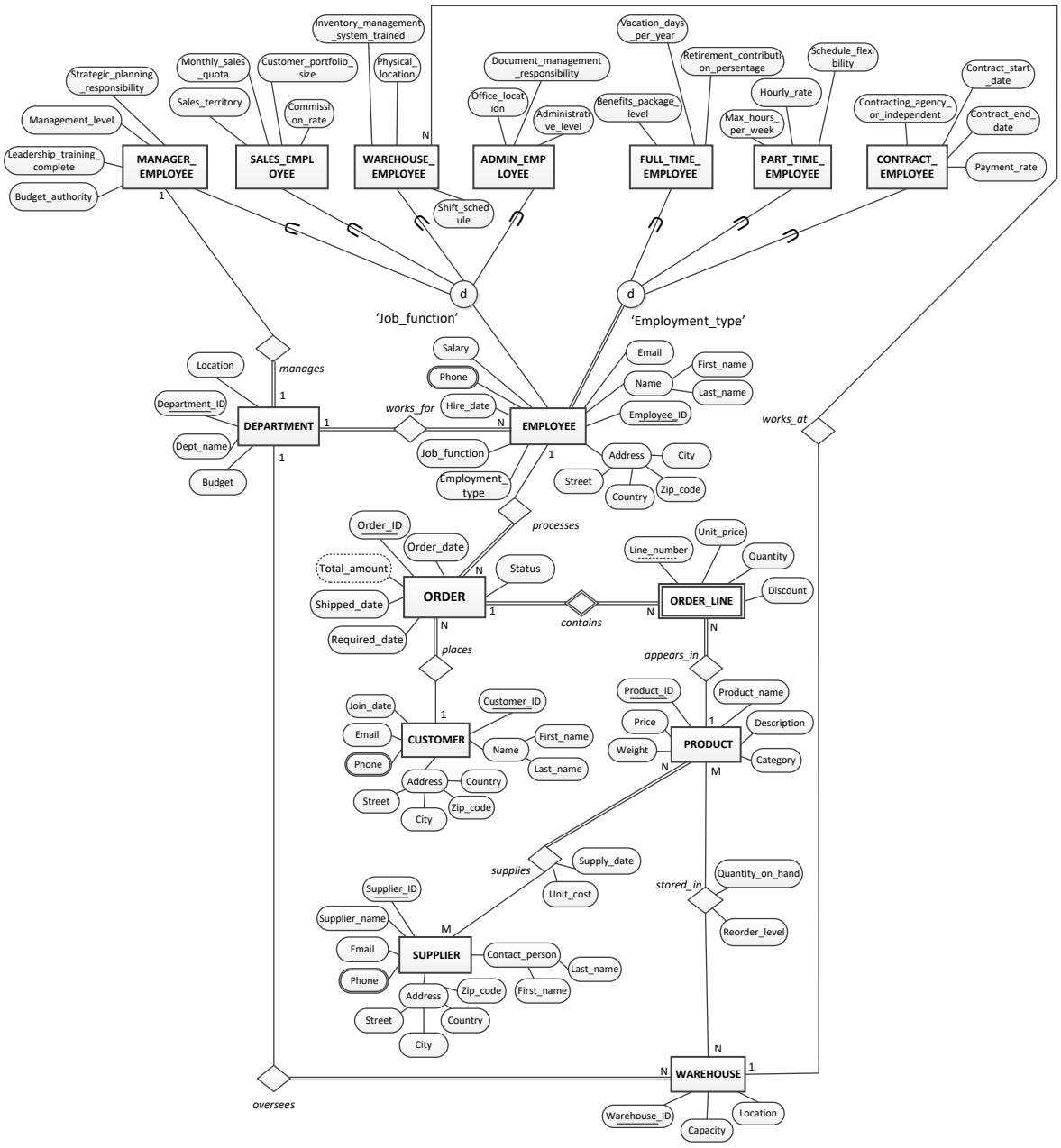


Figure 5.33: Complete Integrated Company EER Model Combining Specialization Hierarchies with Base ER Relationships

5.7.3 EER Design Lessons from Case Studies

The University and Company EER models demonstrate complementary approaches to handling organizational complexity through specialization hierarchies. The University model showcases deep hierarchical structures with lattice patterns for dual-role entities, while the

Company model illustrates multiple independent specializations operating simultaneously on the same superclass. Together, these case studies reveal essential principles for effective EER modeling.

5.7.3.1 Contrasting Patterns: Lattice vs. Multiple Independent Specializations

There are two fundamental differences between the two EER patterns:

University Lattice Pattern (TEACHING_ASSISTANT)

- **Structure:** Shared subclass with multiple inheritance
- **Inheritance paths:** Two distinct paths to root (PERSON → STUDENT → DOCTORAL_STUDENT → TEACHING_ASSISTANT and PERSON → INSTRUCTOR → TEACHING_ASSISTANT)
- **Attribute inheritance:** Inherits from multiple parents, creating a diamond pattern
- **Constraint interaction:** Constraints from both paths must be satisfied
- **Use case:** Entity genuinely IS both parent types (doctoral student AND instructor)

Company Independent Specializations Pattern

- **Structure:** Single superclass with multiple independent specialization dimensions
- **Inheritance paths:** Each specialization is independent - no shared subclasses
- **Attribute inheritance:** Simple single inheritance within each dimension
- **Constraint interaction:** Each specialization has independent constraints
- **Use case:** Entity classified along multiple orthogonal dimensions (job function AND employment type)

The Company pattern is simpler to implement than lattices because:

- No diamond inheritance - no attribute conflict resolution needed
- No multiple paths to root superclass
- Each specialization can be managed independently
- Constraints don't interact across specializations

Both patterns are valid EER modeling approaches, chosen based on business requirements:

- **Use lattices** when entities genuinely combine characteristics from multiple parent types
- **Use independent specializations** when entities are classified along multiple orthogonal dimensions

5.8 Chapter Summary

In this chapter, we extended the basic ER modeling concepts from Chapter 4 with Enhanced Entity-Relationship (EER) modeling techniques that capture the semantic complexity of organizations. We began by introducing the fundamental concepts of

subclasses and superclasses, showing how specialized entity types inherit attributes and relationships from more general types while adding unique characteristics. We explored both specialization (top-down identification of subclasses) and generalization (bottom-up synthesis of superclasses), demonstrating that these complementary processes produce identical hierarchical structures through different analytical approaches.

We examined the constraint mechanisms that govern EER models: disjoint versus overlapping constraints that determine whether entities can belong to multiple subclasses, and total versus partial participation that specifies whether all superclass entities must belong to subclasses. We showed how these constraints combine into four distinct patterns: (d,t), (d,p), (o,t), and (o,p), each capturing different organizational classification rules. We distinguished between simple hierarchies with single inheritance and lattices with shared subclasses that inherit from multiple parents, illustrating when each structure type appropriately models organizational reality.

The chapter provided comprehensive coverage of Chen notation for EER diagrams, demonstrating proper use of specialization circles, subset symbols, constraint indicators, and line styles to represent inheritance hierarchies without ambiguity. We emphasized that precise notation is essential for professional database design, ensuring consistent interpretation among designers, stakeholders, and implementers.

Finally, we demonstrated the complete development of the EER model through two comprehensive case studies. The University system illustrated multi-level hierarchies using lattice structures, modeling Bologna Process requirements through PERSON specializing into STUDENT, PROFESSOR, INSTRUCTOR, STAFF, and ALUMNUS hierarchies, with TEACHING_ASSISTANT as a shared subclass that captures doctoral students' dual status. The Company system demonstrated multiple independent specializations, showing how EMPLOYEE is classified simultaneously by job function and employment type with different constraints. We validated both models against representative organizational instances, confirming that our conceptual structures accommodate real instances and accurately reflect business rules.

These conceptual models establish the foundation for transformation into implementable relational database schemas.

Review Questions

5.1. What is Enhanced Entity-Relationship (EER) modeling? Explain how it extends basic ER modeling concepts and describe situations where EER modeling is necessary versus when basic ER modeling suffices.

5.2. Define the following EER concepts and explain their relationship to each other: a) Subclasses and superclasses, b) Specialization and generalization, c) Inheritance

5.3. Explain the four constraint combinations in EER modeling: (d, t), (d, p), (o, t), and (o, p). Provide specific examples from organizational contexts for each combination.

5.4. What is the difference between disjoint and overlapping constraints? Provide examples showing when each is appropriate based on business rules.

5.5. Compare total and partial participation constraints in specialization hierarchies. How do these constraints differ from participation constraints in relationships from Chapter 4?

5.6. Distinguish between specialization and generalization processes. Although they produce identical structures, why is understanding both processes valuable for database designers?

5.7. What is a hierarchy versus a lattice in EER modeling? Provide examples of each and explain when multiple inheritance through lattices becomes necessary.

5.8. Explain shared subclasses and multiple inheritance. Using the TEACHING_ASSISTANT example from this chapter, describe the challenges and benefits of lattice structures.

5.9. Describe Chen notation conventions for EER diagrams, including: a) Specialization circles, b) Subset symbols (⊆), c) Constraint indicators (d, o), d) Line styles (single vs. double) for participation

5.10. What are multiple independent specializations? How does this pattern differ from hierarchies and lattices? Provide the Company EMPLOYEE example illustrating this concept.

5.11. Distinguish between attribute-defined and user-defined specialization. When should discriminating attributes be made explicit in the superclass?

5.12. Explain the systematic methodology for developing EER models presented in Section 5.7. What are the key steps from requirements analysis through validation?

5.13. Compare the University lattice structure with the Company's multiple independent specializations. What different organizational realities does each pattern capture?

5.14. How do you validate an EER model against business requirements? Describe the validation process using scenarios with actual data.

■ ■ ■ ■ ■ ■

Exercises

5.1. Subclass and Superclass Identification

Consider the following requirements for a Vehicle Rental System:

"A rental company has vehicles available for rent. Vehicles include cars, motorcycles, and trucks. All vehicles have registration numbers, makes, models, years, and current mileage. Cars have seating capacity and trunk size. Motorcycles have an engine size and whether they require a special license. Trucks have cargo capacity and the number of axles."

Tasks: a) Identify the superclass and subclasses in this scenario. b) List inherited attributes versus specialized attributes. c) Determine whether this should be a disjoint or

overlapping specialization. d) Determine whether this should be a total or partial specialization. e) Draw the EER diagram using proper Chen notation

5.2. Constraint Analysis

For each of the following business scenarios, determine the appropriate constraints (disjoint/overlapping and total/partial):

a) "Bank accounts are classified as savings accounts or checking accounts. An account cannot be both types simultaneously, and every account must be one of these types."

b) "University community members can be students, professors, staff, or alumni. A person can hold multiple roles simultaneously (e.g., a staff member pursuing graduate studies)."

c) "Employees may be classified by job function as managers, engineers, or salespeople, but many employees perform general duties without specialized classification."

d) "Products in inventory may be classified as premium items, clearance items, or seasonal items. Some products don't fit any of these categories. A product might be both premium quality AND on seasonal promotion simultaneously."

For each scenario:

- Specify the constraint combination: (d, t), (d, p), (o, t), or (o, p)
- Justify your choice based on business rules
- Draw the specialization using Chen notation

5.3. Specialization vs. Generalization Practice

Consider these entity types for a Healthcare System:

Current entities: GENERAL_PRACTITIONER, SPECIALIST, SURGEON, NURSE_PRACTITIONER

All have: employee_id, name, license_number, hire_date, specialty_area

Tasks: a) Apply **generalization** (bottom-up): Identify common characteristics and create appropriate superclass(es). b) Now start fresh with a MEDICAL_STAFF superclass and apply **specialization** (top-down): Identify subclasses based on distinguishing characteristics. c) Compare the two approaches - do they produce identical structures? d) Draw both approaches using EER diagrams. e) Specify appropriate constraints.

5.4. Hierarchy vs. Lattice Design

Design EER models for these two scenarios:

Scenario A - Product Hierarchy: "A publishing house produces three types of publications: print books, e-books, and audiobooks. Print books have ISBN numbers, page counts, binding types (hardcover, paperback), and warehouse locations. E-books have file formats (PDF, EPUB, MOBI), file sizes, and DRM protection status. Audiobooks have narrators, durations, and audio formats (MP3, AAC)."

Scenario B - Author-Editor Lattice: "The publishing house employs content creators in different roles. Authors write manuscripts and have pen names, genres of expertise, and

publication histories. Editors review manuscripts and have editing specializations (developmental, copy, technical) and manuscripts currently under review. Some senior professionals work as author-editors, both writing their own books and editing others' work. They have attributes from both roles: their own publication history AND their editing specialization and current manuscript review load."

Tasks: a) Identify which scenario requires a hierarchy and which requires a lattice. b) For each scenario, draw the complete EER diagram with proper notation. c) Explain why the lattice is necessary in Scenario B (why the author-editor cannot be modeled as a simple hierarchy). d) Specify all constraints appropriately. e) Describe attribute inheritance in the lattice structure for an author-editor who writes mystery novels and edits technical manuals.

5.5. Multiple Independent Specializations

Design an EER model for a Hospital Personnel System:

"Hospital employees are classified in two independent ways. First, by professional role: physicians, nurses, technicians, or administrative staff. Each role has specialized attributes. Physicians have medical license numbers and specializations; nurses have certifications and shift preferences; technicians have equipment certifications; and administrative staff have office assignments.

Second, all employees are classified by employment status: full-time, part-time, or contract. Full-time employees have annual salaries and benefits packages. Part-time employees have hourly rates and maximum weekly hours. Contract employees have contract start/end dates and contract rates."

Tasks: a) Identify the two independent specialization dimensions. b) Determine constraints for each specialization (disjoint/overlapping, total/partial). c) Determine if discriminating attributes should be made explicit. d) Draw the complete EER diagram showing both independent specializations. e) Explain how a single employee (e.g., a part-time nurse) belongs to subclasses from both dimensions. f) List all possible combinations of role and employment status. g) Explain why this pattern (independent specializations) is different from the lattice pattern and when each is appropriate.

5.6. Attribute-Defined Specialization

Consider this requirements statement:

"Customers are classified by membership level: Bronze, Silver, Gold, or Platinum, based on their annual purchase amount. The membership_level attribute determines classification. Bronze members (purchases < $500) receive standard service. Silver members ($500-$1,999) receive priority shipping. Gold members ($2,000-$4,999) receive priority shipping plus dedicated support. Platinum members (≥$5,000) receive all benefits plus exclusive offers."

Tasks: a) Identify the defining attribute and its role. b) Design the EER structure with proper attribute-defined notation. c) Specify the domain constraints for the defining attribute. d) Show how the defining attribute appears in Chen notation. e) List specialized attributes for each membership level. f) Explain how this differs from user-defined specialization.

5.7. Complete EER Diagram - Zoo Management System

Design a comprehensive EER model for a City Zoo:

"The zoo maintains diverse animal collections and employs various staff members. All persons in the system have an SSN, name, birth date, and contact information.

Animals are classified by taxonomy: mammals, birds, reptiles, amphibians, and fish. All animals have animal IDs, common names, scientific names, birth dates, and current health status. Mammals have different fur types, gestation periods, and temperature-regulation methods. Birds have wing spans, migration patterns, and nesting behaviors. Reptiles have different types of scales and temperature preferences. Amphibians have metamorphosis stages. Fish have water-type requirements (freshwater, saltwater, brackish) and preferred swimming depths.

Animals are also independently classified by conservation status: not threatened, vulnerable, endangered, or critically endangered. Each status has different breeding program requirements and habitat protection protocols.

Zoo staff members have different roles. Zookeepers care for animals, have assigned animal sections, and receive feeding certifications and training in animal handling. Veterinarians provide medical care and hold medical license numbers, surgical certifications, and specializations (e.g., large animals, exotic species, avian medicine). Education staff conduct tours and programs, have education certifications, and are assigned to school groups. Administrative staff manage operations and have office assignments and departmental responsibilities.

Some veterinarians also work as researchers, conducting studies on animal behavior, breeding programs, or disease prevention. They have research projects, grant funding, and publication records in addition to their veterinary duties. These vet-researchers participate in both clinical care AND active research."

Tasks:

a) Identify all superclass/subclass structures (animals, staff, vet-researchers)
b) Determine appropriate constraints for each specialization
c) Identify where hierarchies versus lattices are needed
d) Design the complete EER diagram with proper Chen notation
e) Show all inheritance paths
f) Validate your design with specific scenarios:
- A critically endangered Sumatran tiger (mammal)
- A veterinarian-researcher specializing in avian medicine while studying migration patterns
- A zookeeper assigned to the reptile section
- An education staff member conducting school tours

5.8. Complex EER Diagram - Financial Services

Design a comprehensive EER model for a Financial Services Company:

"The company serves clients with various financial products. Clients have client IDs, names, addresses, phone numbers, email addresses, and financial profiles, including income level, risk tolerance, and investment experience.

Financial products include: deposit accounts (savings accounts, checking accounts, money market accounts), investment products (stocks, bonds, mutual funds, retirement accounts), and loan products (mortgages, personal loans, auto loans, business loans).

All financial products have account numbers, opening dates, current balances, and account status. Deposit accounts have interest rates and minimum balances. Savings accounts have withdrawal limits. Checking accounts have overdraft protection options and monthly fees. Money market accounts have tiered interest rates based on balance.

Investment products have portfolio managers and risk ratings. Stocks have ticker symbols, purchase prices, current prices, and quantities. Bonds have maturity dates, face values, and coupon rates. Mutual funds have fund names, expense ratios, and NAV (net asset value). Retirement accounts (IRAs, 401(k)s) have contribution limits, tax treatment types, and vesting schedules.

Loan products have interest rates, payment schedules, and collateral (if applicable). Mortgages have property addresses, property values, and loan-to-value ratios. Personal loans have purposes and co-signer information. Auto loans have vehicle information (VIN, make, model, year). Business loans have business information and business plans.

Clients can hold multiple products simultaneously. Clients are also classified by relationship type: retail clients (individual customers), private banking clients (high-net-worth individuals requiring specialized services), and institutional clients (corporations and organizations). Each type has different service levels and fee structures.

Financial advisors manage client relationships. Advisors have employee IDs, names, certifications (CFP, CFA), specializations, and client portfolios. Advisors specialize in different areas, such as retirement planning, wealth management, or lending. Some senior advisors specialize in multiple areas."

Tasks: a) Identify all superclass/subclass relationships. b) Determine whether each specialization should be disjoint or overlapping, total or partial. c) Distinguish between different specialization patterns (simple hierarchies, multiple levels, multiple independent specializations). d) Handle the many different product types with appropriate EER structures. e) Design the client classification system. f) Model advisor specializations appropriately. g) Draw the complete EER diagram with proper Chen notation. h) Specify all constraints clearly.

i) Validate your design against these scenarios:

- A retail client with a checking account, retirement IRA, and mortgage
- A private banking client with multiple investment products
- An institutional client with business loans
- A wealth management advisor specializing in retirement planning

5.9. Complex EER Diagram - Transportation Network

Design a comprehensive EER model for a Multi-Modal Transportation Network:

"The transportation network coordinates various vehicle types and routes. All vehicles have registration numbers, capacities, manufacturers, model years, and maintenance schedules.

Vehicles are specialized by type: buses, trains, ferries, and bicycles (for a bike-share program). Buses have bus numbers, seating capacities, accessibility features, and route assignments. Trains have train numbers, car counts, and track compatibility. Ferries have vessel names, passenger capacities, vehicle capacities, and dock assignments. Bicycles have bike numbers, lock codes, and GPS trackers.

Vehicles are independently classified by ownership: agency-owned (purchased by the transit authority), leased (rented from external companies), or contractor-operated (owned and operated by private companies under contract). Agency-owned vehicles have purchase dates and purchase prices. Leased vehicles have lease companies, lease start dates, lease end dates, and monthly lease payments. Contractor-operated vehicles have contractor company information and service contracts.

Transportation routes connect stations/stops. Stations have station IDs, names, locations, and facility types. Bus stops are simple locations with stop IDs. Train stations have platform numbers and accessibility features. Ferry terminals have dock numbers and vessel berthing information. Bike stations have docking capacities and available bike counts.

Riders use the system to pay fares. Riders have rider IDs, names, and contact information. Riders can be regular riders (frequent users with monthly passes), occasional riders (pay-per-ride), students (with student discounts), seniors (with senior discounts), or disabled riders (with accessibility requirements). These classifications overlap; a rider can be both a student and a regular rider, for example.

Transportation workers operate the system. Workers have employee IDs, names, hire dates, and certifications. Operators drive/pilot vehicles and have vehicle-type certifications: bus operators are certified for buses, train engineers for trains, and ferry captains for ferries. Maintenance workers have specialization areas (mechanical, electrical, structural) and certification levels. Administrative staff have office assignments and responsibilities.

The system tracks schedules. Each vehicle operates on scheduled runs with departure times, arrival times, and stop sequences. The system must track actual performance: scheduled vs. actual departure/arrival times, delays, and service interruptions."

Tasks: a) Identify all specialization opportunities (vehicles, riders, workers, stations). b) Determine which specializations are independent versus hierarchical. c) Design overlapping constraints for rider classifications. d) Handle the ownership classification of vehicles. e) Model worker specializations with appropriate constraints. f) Design the temporal aspects (scheduled vs. actual times). g) Draw the complete EER diagram showing all specialization patterns. h) Use proper Chen notation for all constraints. i) Validate your design with these scenarios:

- An agency-owned bus operated by a certified bus operator
- A student who is also a regular rider with a monthly pass
- A leased train operated on a scheduled route with delays

- A bike-share bicycle at a bike station
- A maintenance worker specializing in mechanical repairs for trains

5.10. Complex EER Diagram - Media Streaming Platform

Design a comprehensive EER model for a Media Streaming Platform:

"The platform delivers content to subscribers. Content items have content IDs, titles, descriptions, release dates, ratings, and view counts. Content is specialized by type: movies, TV series, documentaries, and podcasts.

Movies have runtimes, directors, and cast lists. TV series have season numbers, episode counts, and series status (ongoing/completed). Each episode belongs to one series and has an episode number, title, runtime, and air date. Documentaries have subjects, narrators, and educational categories. Podcasts have hosts, episode frequencies, and episode durations.

Content is also independently classified by content rating: G, PG, PG-13, R, or MA (mature audience), based on age appropriateness. Each rating has minimum age requirements and content warnings.

Content creators produce content for the platform. Creators have creator IDs, names, and profiles. Individual creators (independent producers) have portfolios and subscriber counts. Production companies have company names, founding years, and multiple productions. Networks have network names, programming schedules, and broadcast agreements.

Subscribers access content through subscriptions. Subscribers have subscriber IDs, names, email addresses, and payment information. Subscriptions are classified by tier: Basic (standard definition, single-screen), Standard (high-definition, two simultaneous screens), and Premium (ultra-HD, four simultaneous screens, downloadable content). Each tier has different pricing and features.

Subscribers are also classified by subscription status: active (currently paying), trial (free trial period), or paused (temporarily suspended). This is independent of tier; a Premium subscriber might pause their subscription.

The platform has viewing profiles within subscriptions. Each profile has a profile name, age rating restrictions, viewing history, and personalized recommendations. Profiles can be designated as adult profiles (no restrictions), teen profiles (restricted content ratings), or child profiles (strictly limited content).

The platform tracks viewing activity. Each viewing session records which profile watched which content, the start time, the duration watched, the completion status, and the device used. The platform uses this data for recommendations.

Content is organized into collections. Collections have collection IDs, names, themes, and curator information. Collections include: genre-based collections (Action, Comedy, Drama), featured collections (Staff Picks, Trending Now), and personalized collections (Because You Watched, Top Picks for You). A content item can appear in multiple collections simultaneously."

Tasks: a) Identify all specialization hierarchies (content types, creator types, profile types). b) Identify multiple independent specializations (subscription tier vs. subscription status). c) Determine appropriate constraints for each specialization. d) Handle the weak entity relationship between TV series and episodes. e) Model the overlapping collection membership (content in multiple collections). f) Design the profile restrictions based on

content ratings. g) Handle the temporal tracking of viewing sessions. h) Draw the complete EER diagram with all specialization patterns. i) Use proper Chen notation, including all constraint specifications. j) Validate your design with these scenarios:

- A TV series with multiple episodes, some episodes rated PG, and others rated MA
- A Premium subscriber on trial status with three profiles (adult, teen, child)
- A production company that creates both movies and documentaries
- Viewing history tracking across different profiles for recommendations
- Content appearing in multiple collections (e.g., a movie in both "Action" and "Staff Picks")

References and Recommended Readings

Primary References

Hristoski, I., & Dimovski, T. (2022). *Dizajn na relaciski bazi na podatoci, Tom 1* [Design of Relational Databases, Volume 1]. University "St. Kliment Ohridski", Bitola. (In Macedonian)

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of Database Systems* (7th ed.). Pearson.

Teorey, T. J., Yang, D., & Fry, J. P. (1986). A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys, 18*(2), 197-222.

Batini, C., Ceri, S., & Navathe, S. B. (1992). *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings.

Additional Textbooks

Date, C. J. (2003). *An Introduction to Database Systems* (8th ed.). Addison-Wesley.

Connolly, T. M., & Begg, C. E. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson.

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database System Concepts* (7th ed.). McGraw-Hill.

Historical and Theoretical Foundations

Chen, P. P. (1976). The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems, 1*(1), 9-36.

Scheuermann, P., Scheffner, G., & Weber, H. (1980). Abstraction capabilities and invariant properties modelling within the entity-relationship approach. In *Entity-Relationship Approach to Systems Analysis and Design* (pp. 121-140). North-Holland.

Smith, J. M., & Smith, D. C. (1977). Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems, 2*(2), 105-133.

Constraint Specifications and Semantics

Lenzerini, M., & Santucci, G. (1983). Cardinality constraints in the entity-relationship model. In *Entity-Relationship Approach to Software Engineering* (pp. 529-549). North-Holland.

Gogolla, M., & Hohenstein, U. (1991). Towards a semantic view of an extended entity-relationship model. *ACM Transactions on Database Systems, 16*(3), 369-416.

Practical Design Methodologies

Teorey, T. J., Lightstone, S. S., Nadeau, T., & Jagadish, H. V. (2011). *Database Modeling and Design: Logical Design* (5th ed.). Morgan Kaufmann.

Hernandez, M. J. (2013). *Database Design for Mere Mortals: A Hands-On Guide to Relational Database Design* (3rd ed.). Addison-Wesley.

Implementation and Transformation

Markowitz, V. M., & Makowsky, J. A. (1990). Identifying extended entity-relationship object structures in relational schemas. *IEEE Transactions on Software Engineering, 16*(8), 777-790.

Premerlani, W. J., & Blaha, M. R. (1994). An approach for reverse engineering of relational databases. *Communications of the ACM, 37*(5), 42-49.

Advanced Topics

Hull, R., & King, R. (1987). Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys, 19*(3), 201-260.

Abiteboul, S., Hull, R., & Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

Contemporary Perspectives and Tools

Ambler, S. W., & Holub, A. (2006). *The Object Primer: Agile Model-Driven Development with UML 2.0* (3rd ed.). Cambridge University Press.

Song, I. Y., Evans, M., & Park, E. K. (1995). A comparative analysis of entity-relationship diagrams. *Journal of Computer and Software Engineering, 3*(4), 427-459.

Case Studies and Applications

Badia, A., & Lemire, D. (2011). A call to arms: Revisiting database design. *ACM SIGMOD Record, 40*(3), 61-69.

Simsion, G., & Witt, G. (2004). *Data Modeling Essentials* (3rd ed.). Morgan Kaufmann.

Standards and Guidelines

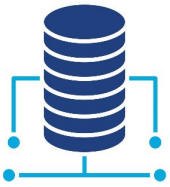
European Commission. (2015). ECTS Users' Guide. Publications Office of the European Union.

Part 3

Logical Design & Implementation

Relational Model

Relational Algebra



Learning Objectives:

By the end of Part 3, you will be able to:

- Transform conceptual ER and EER models into structured relational database schemas
- Apply systematic mapping rules for entities, relationships, and advanced EER constructs
- Understand the mathematical foundations of the relational model and its practical implications
- Master relational algebra operations for expressing complex database queries
- Implement complete database systems using SQL DDL and DML statements
- Navigate DBMS-specific syntax variations and make informed implementation decisions
- Create production-ready databases that maintain data integrity and support organizational operations
- Validate logical designs through normalization and constraint specification

***Chapter Overview*:**

Chapter 6: Logical Design and Implementation introduces the transition from conceptual models to implementable database structures. It establishes the relational model as the target for our ER/EER designs and previews the systematic transformation process that bridges conceptual understanding with physical implementation.

Chapter 7: Relational Model presents the formal foundations of relational databases, including mathematical definitions, key concepts, integrity constraints, and the systematic mapping rules for transforming ER/EER diagrams into relational schemas. It demonstrates complete transformations using both University and Company case studies.

Chapter 8: Relational Algebra explores the theoretical query language underlying all database operations. It covers fundamental operations (selection, projection, joins, set operations) and demonstrates how complex information requirements translate into formal algebraic expressions using practical examples from both case studies.

Chapter 9: SQL Implementation provides comprehensive coverage of SQL as the industry-standard implementation language. It demonstrates DDL statements for creating database structures with constraints, DML operations for data manipulation, and addresses DBMS-specific variations between SQL Server and MySQL using complete implementations of both case studies.

Our Comprehensive Case Study Approach

Throughout Part 3, we continue using our established University Management System and Company/Business Database case studies, now transforming them into working database implementations. You'll see how the ER and EER diagrams you created in Part 2 become concrete relational schemas, how those schemas support queries through relational algebra, and finally, how everything is implemented in actual SQL code. This continuity ensures that you understand not just isolated techniques, but the complete journey from business requirements to functioning database systems.

Chapter 6

6. Logical Design and Implementation

6.1 Introduction: From Vision to Reality

Imagine you have just finished designing your dream home with an architect. The blueprints are beautiful—showing rooms, doorways, windows, and how everything connects together. Your architect understands your family's needs perfectly, and the *conceptual design* captures exactly what you want. But now comes the next challenge: how do you transform those blueprints into an *actual house*? You need to specify exact dimensions, choose materials, determine structural supports, and create detailed plans that builders can actually follow. The vision must become reality.

This transition represents exactly where we stand in our database design journey. In Part 2, we mastered **conceptual design**, learning to create precise ER and EER diagrams that capture organizational requirements using Chen notation. Our University Management System model represents students, courses, professors, and their complex relationships. Our Company Database represents employees, departments, customers, orders, and products with well-defined hierarchies. These conceptual models are elegant and expressive, effectively communicating with both technical teams and business stakeholders.

But conceptual models, like architectural blueprints, cannot function on their own. Database Management Systems do not understand ER diagrams. They work with *tables*, *columns*, *keys*, and carefully structured data. The journey from conceptual design to a working database requires a crucial intermediate step: **logical design**.

6.2 Understanding Logical Database Design

Logical design transforms abstract conceptual models into structured, implementable database specifications based on a specific data model (Teorey et al., 2011). For relational databases, the dominant database technology for over four decades, logical design means converting ER and EER diagrams into **relational schemas**: precise specifications of tables with their columns, keys, constraints, and relationships.

> **Definition: Logical Database Design**
>
> Logical database design is the process of transforming a conceptual data model (ER or EER diagram) into a logical data model consistent with a specific database technology, while remaining independent of any particular Database Management System (Coronel

> & Morris, 2017; Hristoski & Dimovski, 2022). The result is a structured specification that can be implemented in any DBMS supporting that technology.

This definition reveals several key aspects of logical design:

6.2.1 Key Aspects of Logical Design

Technology-Specific but DBMS-Independent: Logical design commits to the relational model, but doesn't yet choose between SQL Server, MySQL, PostgreSQL, or Oracle. The resulting relational schema works with any relational DBMS (Elmasri & Navathe, 2016).

Structured and Formal: Unlike conceptual models that use intuitive visual notation, logical models employ precise mathematical structures. Tables have specific column types, keys follow formal definitions, and constraints express exact rules (Date, 2003).

Implementation-Ready: Logical designs translate directly into database creation statements. The gap between logical design and physical implementation is narrow and straightforward (Connolly & Begg, 2015).

Optimized for Data Integrity: Logical design incorporates mechanisms—primary keys, foreign keys, constraints—that enforce business rules and maintain data consistency automatically (Codd, 1970).

6.3 Why Logical Design Matters

Students sometimes wonder why we need this intermediate step. Why not jump directly from ER diagrams to SQL code? Several crucial reasons justify this separate phase (Batini et al., 1992; Teorey et al., 2011):

6.3.1 Why Separate Logical Design Phase?

Clear Separation of Concerns: Conceptual design focuses on understanding business requirements; logical design focuses on data structure optimization; physical design focuses on performance tuning. Separating these concerns allows designers to concentrate fully on each challenge without distraction (Simsion & Witt, 2005).

Technology Evaluation: Organizations sometimes need to evaluate different database technologies. A well-designed logical model can guide implementation in multiple database systems, supporting technology comparisons and migration planning.

Design Validation: The logical design provides an opportunity to verify that our conceptual model transforms correctly into implementable structures. We can check for normalization issues, identify missing constraints, and refine our design before investing in physical implementation (Hernandez, 2013).

Team Communication: Logical schemas provide a precise technical specification that development teams can implement consistently. While ER diagrams communicate well with business users, relational schemas communicate effectively with database developers and administrators.

Foundation for Evolution: Databases evolve continually as organizations grow and requirements change. A solid logical design provides the foundation for systematic

evolution, helping teams understand how changes propagate through the system (Ambler & Holub, 2006).

6.4 The Logical Design Process

Logical design for relational databases proceeds through several systematic steps, each transforming our conceptual understanding into increasingly concrete specifications (Teorey et al., 1986; Markowitz & Makowsky, 1990).

> **Logical Design Process: Five Essential Steps**
>
> **Step 1: Entity Mapping** — Convert each entity from ER diagrams into relations (tables)
>
> **Step 2: Attribute Specification** — Transform attributes into columns with appropriate data types
>
> **Step 3: Relationship Transformation** — Implement relationships through foreign keys or junction tables
>
> **Step 4: EER Construct Mapping** — Transform specialization hierarchies and advanced constructs
>
> **Step 5: Constraint Specification** — Formalize all business rules as database constraints

Step 1: Entity Mapping begins the transformation by converting each entity from our ER diagram into a relation (table). **Strong entities** become base tables with their attributes becoming columns. The entity's **key attribute** becomes the **primary key** (Elmasri & Navathe, 2016). Our STUDENT entity becomes a STUDENT table; COURSE becomes a COURSE table; EMPLOYEE becomes an EMPLOYEE table.

Weak entities require special handling, with the owner entity's primary key included in their **composite primary key**. The weak entity CLASS_SESSION in our University model depends on COURSE for its identification — it cannot be uniquely identified by its own attributes alone. Its composite primary key therefore combines the owner's primary key (Course_ID, included as a foreign key) with its own partial key attributes (Session_number and Year). This reflects the real-world logic that a class session is only meaningful in the context of the course it belongs to, and that the same session number can recur across different courses and academic years.

Step 2: Attribute Specification refines our understanding of each column. *Simple attributes* translate directly into columns. *Composite attributes* require decisions: should we store Name as a single column, or decompose it into First_name, Middle_initial, and Last_name? The answer depends on anticipated queries and business requirements. *Multivalued attributes* cannot exist in relational tables; we must create separate tables to accommodate them. *Derived attributes* typically don't appear in base tables; they're calculated when needed.

Step 3: Relationship Transformation represents the most complex aspect of logical design (Teorey et al., 1986). The transformation approach depends entirely on **relationship cardinality**:

One-to-one relationships typically consolidate into a single table, or are implemented as two tables with a foreign key linking them. The choice depends on *participation constraints* and the distribution of attributes.

One-to-many relationships are implemented through foreign keys. The "many" side receives the primary key from the "one" side as a foreign key. In our University model, the 1:N relationship between FACULTY and PROFESSOR means the PROFESSOR table receives Faculty_ID as a foreign key.

Many-to-many relationships require new tables—called **junction tables**, **bridge tables**, or **associative tables** (Elmasri & Navathe, 2016). The STUDENT-COURSE enrollment relationship in our University model creates an ENROLLS_IN table with foreign keys referencing both STUDENT and COURSE, along with any *relationship attributes* like Grade and Enrollment_date.

Step 4: EER Construct Mapping extends our basic transformations to handle **specialization hierarchies**, **shared subclasses**, and **categories** (Markowitz & Makowsky, 1990). Multiple mapping strategies exist for EER constructs, each with different trade-offs regarding *query performance*, *storage efficiency*, and *constraint enforcement*. We must choose approaches that match our specific requirements.

Step 5: Constraint Specification completes our logical design by formalizing all business rules as database constraints (Date, 2019). **Entity integrity** ensures primary keys are never null. **Referential integrity** maintains consistency between related tables. **Domain constraints** limit attribute values to valid ranges. **Check constraints** enforce complex business rules directly at the database level.

6.5 The Relational Model: Our Foundation

The relational model, introduced by Dr. E.F. Codd in 1970, provides the mathematical foundation for logical design (Codd, 1970). Understanding its core principles helps us make better design decisions.

> **Relational Model: Core Concepts**
>
> **Relations (Tables)**: Organize data into rows (tuples) and columns (attributes), with each column drawing values from a specific domain.
>
> **Keys**: Superkeys uniquely identify tuples; candidate keys are minimal superkeys; primary keys are chosen candidate keys that definitively identify each tuple; foreign keys link tables together (Date, 2019).
>
> **Integrity Constraints**: Entity integrity requires primary keys to be non-null and unique; referential integrity ensures foreign keys reference existing primary keys or contain null values (Elmasri & Navathe, 2016).
>
> **Simplicity and Power**: All data structures are tables, all operations produce tables, and complex database operations are composed from simple algebraic operations (Codd, 1970).

Relations (tables) organize data into **rows (tuples)** and **columns (attributes)**, with each column drawing values from a specific domain. The relational model's elegance lies in its

simplicity: all data structures are tables, all operations produce tables, and complex database operations are composed from simple algebraic operations (Date, 2003).

Keys play a fundamental role in the relational model. A **superkey** is any set of attributes that uniquely identifies tuples. A **candidate key** is a *minimal superkey*—removing any attribute destroys its uniqueness property. The **primary key** is the chosen candidate key that definitively identifies each tuple. **Foreign keys** link tables together, creating the relationships that give relational databases their power.

Integrity constraints maintain data consistency. **Entity integrity** requires primary keys to be *non-null* and *unique*. **Referential integrity** ensures foreign keys reference existing primary keys or contain null values. These constraints, enforced automatically by the DBMS, prevent the data inconsistencies that plagued earlier database systems.

6.6 Relational Algebra: The Query Language

Before implementing databases in SQL, we must understand **relational algebra**—the formal query language underlying all database operations (Codd, 1970). Relational algebra provides operations for extracting information from relations: **selection** filters tuples, **projection** extracts columns, **joins** combine related tables, and **set operations** (union, intersection, difference) manipulate table collections.

Understanding relational algebra helps us think precisely about queries (Date, 2003). When stakeholders request "all students enrolled in Computer Science courses with grades above 7.5," relational algebra expresses this as a sequence of formal operations: *join* STUDENT with ENROLLS_IN, *join* with COURSE, *select* where Department equals 'CS' and Grade exceeds 7.5, and *project* the desired attributes.

This formal foundation guides SQL implementation (Silberschatz et al., 2019). Every SQL query ultimately translates into relational algebra operations, which the DBMS optimizer then transforms into *efficient execution plans*. Designers who understand relational algebra write better queries and understand performance implications more deeply.

6.7 SQL Implementation: Making It Real

The final step in Part 3 transforms our logical designs into working database systems using **SQL (Structured Query Language)**. SQL combines **Data Definition Language (DDL)** for creating structures with **Data Manipulation Language (DML)** for managing content (Elmasri & Navathe, 2016).

DDL statements create our database structures (Connolly & Begg, 2015). **CREATE TABLE** definitions specify table names, column definitions with data types, and constraints that enforce business rules. Our logical design translates directly into CREATE TABLE statements, with each relation becoming a table, each attribute becoming a column, and each key becoming a constraint.

DML statements populate and modify our databases. **INSERT** adds new data, **UPDATE** modifies existing records, and **DELETE** removes obsolete information. More importantly, **SELECT** queries extract information, combining tables through *joins* and filtering results to answer specific questions.

DBMS-specific considerations emerge during implementation (Coronel & Morris, 2017). SQL Server and MySQL, while both implementing the SQL standard, offer different *data types*, *constraint syntax variations*, and unique features. Professional database developers must navigate these variations, choosing approaches that work across platforms when possible and adapting to platform-specific capabilities when necessary.

6.8 Our Journey Through Part 3

The upcoming chapters systematically develop the logical design and implementation skills through our established case studies. **Chapter 7** presents the relational model comprehensively, demonstrating complete transformations of our University and Company ER/EER diagrams into relational schemas. You'll learn the *mapping rules*, understand the rationale behind each transformation, and see how complex conceptual structures become elegant relational designs.

Chapter 8 explores relational algebra, showing how formal operations express complex queries. You'll learn to think *algebraically* about information requirements, decomposing questions into fundamental operations that combine to produce answers. The University and Company examples demonstrate how real business questions translate into algebraic expressions.

Chapter 9 brings everything together through SQL implementation. You'll see complete database creation scripts for both case studies, learn to write complex queries that answer real business questions, and understand how to adapt your implementations for different DBMS platforms. By the end, you'll have *functioning databases* ready for deployment.

6.9 Success Principles for Logical Design

As you progress through Part 3, keep these principles in mind:

Start with Solid Conceptual Foundations: Quality logical design depends on quality conceptual design (Batini et al., 1992). Review your ER and EER diagrams carefully before beginning transformation. Correct conceptual errors now prevent complicated logical patches later.

Follow Systematic Processes: Logical design isn't a creative art. It's systematic engineering (Teorey et al., 2011). Follow the mapping rules methodically, validating each transformation. Systematic approaches produce consistent, maintainable results.

Think About Queries: Design decisions should consider how users will query your database. Will they frequently join specific tables? Plan key structures accordingly. Will they filter on particular attributes? Consider constraint specifications that support those queries (Hernandez, 2013).

Validate Relentlessly: Check every foreign key reference, validate every constraint, verify every relationship. Small logical errors cascade into serious implementation problems. Thorough validation during logical design saves countless hours debugging production systems (Simsion & Witt, 2005).

Document Thoroughly: Your logical design documentation guides implementation teams and supports future maintenance (DAMA International, 2017). Document design decisions, explain mapping choices for complex EER constructs, and note constraint rationale. Your future self will thank you.

6.10 The Path Forward

Logical design represents the crucial bridge between *conceptual understanding* and *physical reality*. The abstract entities and relationships in your ER diagrams are about to become concrete tables you can populate with real data and query with actual SQL statements. The transformation requires care, precision, and systematic application of well-established principles.

But do not let the formality intimidate you. The **mapping rules** you will learn in Chapter 7 are straightforward once understood. The **relational algebra** in Chapter 8 provides powerful conceptual tools for thinking about queries. The **SQL** in Chapter 9 brings immediate satisfaction; your designs come alive as functioning database systems.

The conceptual design skills from Part 2 provided your foundation. The logical design and implementation techniques in Part 3 transform that foundation into reality. Let us begin the transformation with Chapter 7, where our University and Company conceptual models become precise relational schemas ready for implementation.

6.11 Summary: Preparing for Transformation

This chapter introduced logical design as the essential bridge between conceptual models and physical databases. We have seen how logical design transforms ER and EER diagrams into structured relational schemas, providing precise specifications that guide database implementation.

Key Insights: Logical Design Fundamentals

Logical design is systematic and rule-based, following established transformation patterns that ensure consistent results (Teorey et al., 1986).

The relational model provides a mathematical foundation for efficiently organizing data while maintaining data integrity (Codd, 1970).

Logical designs remain DBMS-independent, allowing flexibility in technology choices while providing implementation-ready specifications (Coronel & Morris, 2017).

The design process progresses systematically from entities to attributes to relationships to constraints, each step building on previous transformations (Elmasri & Navathe, 2016).

Part 3 develops these concepts thoroughly. Chapter 7 presents complete mapping rules with comprehensive examples. Chapter 8 demonstrates how relational algebra formally expresses complex queries. Chapter 9 shows how everything is implemented in SQL, creating databases ready for production deployment.

Your conceptual design expertise from Part 2, combined with the logical design skills you will develop in Part 3, creates a complete database development capability. The journey continues. Let us transform conceptual vision into relational reality.

Review Questions

Conceptual Understanding Questions

6.1. What is logical database design? Explain how it differs from conceptual design (covered in Chapters 3-5) and physical design (covered later in the course).

6.2. Explain why logical design is described as "technology-specific but DBMS-independent." What does this mean in practical terms?

6.3. List and briefly explain the five essential steps in the logical design process. Why is the order of these steps important?

6.4. Describe the relationship between conceptual design (ER/EER diagrams) and logical design (relational schemas). How does one inform the other?

6.5. What are the four key aspects of logical design mentioned in this chapter? Explain why each is important for successful database development.

Process and Methodology Questions

6.6. Why do we need a separate logical design phase? Couldn't we jump directly from ER diagrams to SQL implementation? Discuss at least three reasons for maintaining this intermediate step.

6.7. Explain the difference between entity mapping (Step 1) and relationship transformation (Step 3) in the logical design process. Why are these treated as separate steps?

6.8. How does the treatment of weak entities differ from that of strong entities during logical design? Provide an example from the University or Company case studies.

6.9. Describe how the three types of relationships (1:1, 1:N, M:N) are transformed differently during logical design. What factors determine which transformation approach to use?

Relational Model Questions

6.10. What is a foreign key? How do foreign keys implement relationships in relational databases? Provide examples from either the University or the Company case study.

6.11. Define entity integrity and referential integrity. Why are these constraints fundamental to relational database design?

6.12. Dr. Codd introduced the relational model in 1970. What characteristics of this model have made it successful for over 50 years? What problems did it solve compared to earlier database models?

Application and Analysis Questions

6.13. In the Company case study, EMPLOYEE participates in a 1:N relationship with DEPARTMENT (each employee works for one department, each department has many employees). Explain how this relationship would be transformed during logical design, including specific details about foreign keys and constraints.

6.14. Why might design decisions made during logical design affect query performance later? Provide specific examples of design choices that have performance implications.

6.15. Explain how properly specified constraints during logical design automatically maintain data integrity. What types of problems would occur without these constraints?

Integration and Synthesis Questions

6.16. This chapter describes logical design as a "bridge" between conceptual design and physical implementation. Explain this metaphor using specific examples from the transformation process.

6.17. Compare the communication effectiveness of ER diagrams versus relational schemas for different audiences (business stakeholders, database developers, system administrators). When is each most appropriate?

6.18. The chapter lists five success principles for logical design. Choose three and explain how following each principle contributes to successful database development using specific examples.

6.19. How does understanding relational algebra (previewed in this chapter, detailed in Chapter 8) help with logical design decisions? Provide specific examples.

6.20. Trace the complete journey of a business requirement from the initial stakeholder interview through conceptual and logical design. Use a specific example, such as "professors teach courses," from the University system to illustrate each transformation step.

Critical Thinking Questions

6.21. The chapter states that logical design "remains DBMS-independent" while being "technology-specific" to the relational model. Some organizations now use NoSQL databases (document stores, graph databases) instead of relational systems. How would logical design differ for these alternative technologies?

6.22. How might the logical design process need to adapt for very large databases (big data scenarios) compared to traditional database applications? What additional considerations arise?

6.23. Consider a situation where an organization must migrate an existing database from one DBMS to another (e.g., from SQL Server to MySQL). How does having a well-documented logical design facilitate this migration?

6.24. Based on what you learned in this chapter, what skills and knowledge do you now have that prepare you for Chapters 7, 8, and 9? What connections do you anticipate between logical design principles and the detailed transformation techniques you'll learn next?

References and Recommended Readings

Primary References

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM, 13*(6), 377-387.

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of Database Systems* (7th ed.). Pearson.

Hristoski, I., & Dimovski, T. (2022). *Dizajn na relaciski bazi na podatoci, Tom 1* [Design of Relational Databases, Volume 1]. University "St. Kliment Ohridski", Bitola. (In Macedonian)

Teorey, T. J., Lightstone, S. S., Nadeau, T., & Jagadish, H. V. (2011). *Database Modeling and Design: Logical Design* (5th ed.). Morgan Kaufmann.

Historical and Theoretical Foundations

Teorey, T. J., Yang, D., & Fry, J. P. (1986). A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys, 18*(2), 197-222.

Date, C. J. (2003). *An Introduction to Database Systems* (8th ed.). Addison-Wesley.

Practical Design Methodologies

Batini, C., Ceri, S., & Navathe, S. B. (1992). *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings.

Simsion, G. C., & Witt, G. C. (2005). *Data Modeling Essentials* (3rd ed.). Morgan Kaufmann.

Hernandez, M. J. (2013). *Database Design for Mere Mortals: A Hands-On Guide to Relational Database Design* (3rd ed.). Addison-Wesley.

Implementation and Transformation

Markowitz, V. M., & Makowsky, J. A. (1990). Identifying extended entity-relationship object structures in relational schemas. *IEEE Transactions on Software Engineering, 16*(8), 777-790.

Coronel, C., & Morris, S. (2017). *Database Systems: Design, Implementation, & Management* (12th ed.). Cengage Learning.

Connolly, T. M., & Begg, C. E. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson.

Contemporary Perspectives and Tools

Date, C. J. (2019). *Database Design and Relational Theory: Normal Forms and All That Jazz* (2nd ed.). O'Reilly Media.

Ambler, S. W., & Holub, A. (2006). *The Object Primer: Agile Model-Driven Development with UML 2.0* (3rd ed.). Cambridge University Press.

DAMA International. (2017). *DAMA-DMBOK: Data Management Body of Knowledge* (2nd ed.). Technics Publications.

Relational Model Foundations

Codd, E. F. (1990). *The Relational Model for Database Management: Version 2*. Addison-Wesley.

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database System Concepts* (7th ed.). McGraw-Hill.

Chapter 7

7. Relational Model

7.1 Introduction: Relational Model

The **relational model** is essential for effectively modeling a University Management System as it provides a **structured** and **logical** way to represent the complex relationships among academic entities. By organizing **entities** such as students, courses, and professors into distinct **tables** and defining their connections through **foreign keys**, the model ensures data **integrity** and **consistency**. For example, students are connected to their enrolled courses, and courses are linked to their professors, through relationships expressed using foreign keys. This arrangement reduces data **redundancy** and enables efficient information **querying** and management. The model's rules ensure data **accuracy** and **reliability**, which is essential for managing vital academic records.

Having completed the conceptual database design, capturing data requirements and the essence of organizational operations through the Entity-Relationship (ER) and Enhanced Entity-Relationship (EER) modeling process, we now move to the logical design stage. Our high-level ER and EER diagrams serve as precise blueprints, detailing entities, their attributes, and key relationships. **Logical design and implementation** are used to develop a structured specification that can be implemented in any DBMS supporting the relational model, as this impacts the quality of subsequent database design stages.

The journey from abstract requirements to a concrete, implementable database continues with the **relational model**. The next crucial stage in database design involves transforming our rich conceptual ER and EER models into a structured logical format (Connolly & Begg, 2015). In this phase, the relational model organizes data into two-dimensional tables, formally known as **relations**. Each relation consists of **rows (tuples)** and **columns (attributes)**, providing an explicit, systematic, and mathematically sound framework for storing and managing academic and organizational data (Date, 2003). This transformation is essential, bridging the gap between our human-centric understanding of data and the rigorous structure required by modern DBMS (Simsion & Witt, 2005). The elegance and efficiency of our database implementation of the University Management System and Company/Business Database **relational models** will directly reflect the foundational strength of the established ER models.

In this chapter, we'll have a comprehensive exploration of the relational model, the cornerstone of modern database systems. This chapter examines the essential principles of data representation through tables, known as relations, and develops the important skill of identifying different types of keys, including candidate, primary, alternate, and foreign keys. These keys are essential for maintaining data integrity within the database. Additionally, we will cover critical concepts such as entity integrity and referential integrity, highlighting their roles in ensuring data consistency.

7.2 What is the Relational Model?

The **relational model** is built on solid mathematical foundations, specifically set theory and predicate logic, which allows for a rigorous definition of data manipulation operations (Codd, 1970; Date, 2003). It provides a concise and powerful way to organize data into two-dimensional tables, known as ***relations***. Each relation consists of named columns, called attributes, and rows, called tuples (Elmasri & Navathe, 2016). This tabular structure makes data easy to understand, manage, and query (Silberschatz et al., 2019).

The Relational Model Characteristics

- **Logical Structure**: The relational model organizes data into two-dimensional tables called **relations**. Each relation consists of rows, known as **tuples**, and columns, known as **attributes**. The structure of the relations is defined by a **schema**, which specifies the name of the relation, its attributes, and their data types.
- **Data Integrity**: A key principle is that all data within a relation must be **atomic**, meaning each cell contains a single value. The model also enforces data **integrity** through keys and constraints, ensuring data consistency and validity. A **primary key** uniquely identifies each tuple, while a **foreign key** establishes a link to a tuple in another relation, representing a relationship between tables.
- **Widespread Importance**: The model's simplicity and flexibility have made it the backbone of modern database management systems, enabling efficient **storage**, **retrieval**, and **manipulation** of vast amounts of information while ensuring data **consistency**. While physical implementation and optimization have advanced, the relational model's logical abstraction remains a robust framework for data management.

Furthermore, as we will see in Chapter 8, the **relational algebra** operations, such as selection, projection, and join, enable powerful data manipulation and retrieval. The relational model's simplicity, mathematical foundation, and flexibility made it the dominant model for relational databases, enabling them to manage large datasets efficiently and ensure data consistency. Behind this straightforward structure is a solid theoretical foundation that explains the **relational data model's** terminology and basic structural concepts (Codd, 1970; Date, 2003; Silberschatz et al., 2019).

The relational **Database Management System** (DBMS) has become today's leading data-processing software. The broader DBMS market is experiencing significant growth, projected to reach USD 225.17 billion by 2032, with a compound annual growth rate (CAGR) (Investopedia, 2025) of 11.41% from 2024 to 2032 (SNS Insider, 2025) and a CAGR of 12.50% from 2025 to 2034 (Market Research Future) (Singh, 2025). The popularity of open-source relational databases like **MySQL** (which leads with around 42% market share) and

PostgreSQL (which is experiencing significant growth at around 17% market share) (6sense, 2025) may affect commercial license sales. However, they will ultimately contribute to the overall usage and significance of relational databases. A major driver of this trend is the shift toward **cloud-based DBMS** solutions. Cloud deployments offer scalability, flexibility, and often different licensing models (e.g., pay-as-you-go), which could impact traditional license sales but increase overall spending on relational database technology. Cloud-based relational databases are expected to maintain their strong growth trajectory.

Ongoing digital transformation initiatives across industries necessitate robust, reliable data management solutions, where relational databases remain a strong choice for structured data. The growing importance of data analytics and business intelligence drives demand for efficient, scalable relational database systems to support complex queries and data warehousing (Ambler & Holub, 2006).

Learning Approach: Our examples throughout this chapter use a comprehensive University Management System and Company/Business Database models that demonstrate the relational model. These will develop a **logical relational data structure** and discuss domains, essential rules for representing relationships, weak entities, composite attributes, and multi-valued attributes. Every concept is illustrated with specific examples from both case studies, showing how abstract principles apply to real-world scenarios.

7.2.1 History of the Relational Model

The relational model, proposed by Edgar **F. Codd** in his seminal 1970 paper “A relational model of data for large shared data banks” (Codd, 1970), laid the groundwork for the dominant paradigm in database management. This paper is now widely regarded as a landmark in database systems. Mathematical relations serve as its formal foundation for structuring, manipulating, and accessing data. The relational model's emphasis on high data independence remains a cornerstone of modern database systems. Application programs must be protected from changes in data storage and organization, including how files are arranged, the order of records, and the ways data is accessed. Keeping these programs insulated from such changes helps organizations adapt quickly and maintain their systems effectively in today's fast-changing data environment. To provide robust foundations for managing data semantics, ensuring consistency, and minimizing redundancy, Codd introduced the concept of **normalized relations**, specifically defining relations free from repeating groups. This normalization, while foundational, has since been expanded and refined with various normal forms to address increasingly complex data dependencies and optimize query performance in modern relational Database Management Systems (DBMS).

Three key projects drove the initial interest in the relational model. IBM's **System R**, developed in the late 1970s (Astrahan et al., 1976), demonstrated the practical viability of the relational model and addressed key implementation challenges, including transaction management, concurrency, and query optimization. System R led to the creation of **Structured Query Language (SQL)**, now the standard for interacting with relational databases. It also spurred the development of early commercial relational DBMS, such as IBM's **DB2** and **Oracle**, laying the foundation for a multi-billion-dollar industry.

At the same time, the **INGRES** project at UC Berkeley pursued similar goals (Stonebraker et al., 1976), contributing to the broader adoption of relational concepts and leading to commercial Ingres-based products. Lastly, the **Peterlee Relational Test Vehicle** in the UK focused on theoretical research, influencing modern query optimization techniques (Todd, 1976).

By the late 1970s and early 1980s, the commercial relational DBMS market grew rapidly, with many systems still in use today, including **cloud-native** databases like Amazon Aurora and Google Cloud Spanner. The core principles established by System R, INGRES, and Peterlee continue to shape relational database design, even in the era of **NoSQL** databases.

7.3 Elements of the Relational Model

The **relational model** provides a powerful and logical framework for representing data. At its core, data is organized into two-dimensional tables, officially known as **relations**. Each relation consists of rows, called tuples, which represent a single record, and columns, known as attributes, that define the properties of that record. A fundamental principle of this model is that all data within a relation must be **atomic**, meaning each cell contains only a single value. The model also enforces **data integrity** through primary and **foreign keys**, which are discussed in detail in Section 7.4. Primary keys uniquely identify each tuple within a relation, while foreign keys establish links between relations, accurately representing the complex relationships between different data sets. This structured approach ensures data consistency and facilitates efficient storage, retrieval, and manipulation of information.

Table 7.1: Fundamental Terminology in Relational Model

ER Model Term	Relational Model Term	Description
Entity	Table (Relation)	*Collection of related data*
Entity Instance	Row (Tuple)	*Single record in relation*
Attribute	Column (Attribute)	*Property or characteristic*
	Domain	*Set of allowed values for an attribute*

The relational model stores every piece of information in tables, where each table corresponds to a single entity or concept.

> **Relation**: A relation is a two-dimensional table consisting of named columns (attributes) and rows (tuples), representing a collection of related data about a single entity or concept (Codd, 1970; Elmasri & Navathe, 2016).

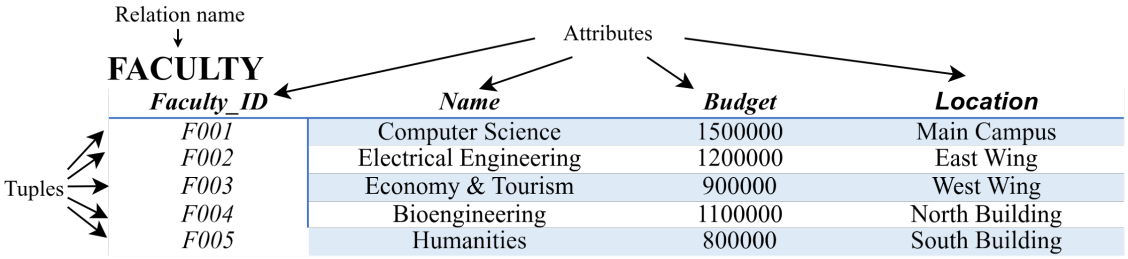
The concept of a **relation** in DBMS provides a structured, logical way to organize data based on mathematical principles. This structure ensures data integrity, reduces redundancy (through normalization), and allows for powerful and flexible data manipulation and retrieval using relational algebra and SQL. The "table with columns and rows" is the user-friendly physical manifestation of this more formal mathematical concept.

University System Example

In Case Study 1: University Database System (**section 4.9.2**), six entities were identified (STUDENT, COURSE, PROFESSOR, FACULTY, TEXTBOOK, CLASS_SESSION). Using the **relational model**, they represent six base **relations**, one for each entity.

Table 7.2 presents the FACULTY relation data as an example from the University Database System. The name of the relation is FACULTY, with the key attribute: Faculty_ID and simple attributes: Name, Budget, and Location.

Table 7.2: University Database System - FACULTY relation data



Faculty_ID	*Name*	*Budget*	*Location*
F001	Computer Science	1500000	Main Campus
F002	Electrical Engineering	1200000	East Wing
F003	Economy & Tourism	900000	West Wing
F004	Bioengineering	1100000	North Building
F005	Humanities	800000	South Building

Company System Example

In Case Study 2 Company Database System **(section 4.9.5)**, eight entities (EMPLOYEE, DEPARTMENT, WAREHOUSE, ORDER, CUSTOMER, PRODUCT, SUPPLIER, ORDER_LINE) were identified. Using the **relational model**, they represent eight base **relations,** one for each entity.

Table 7.3 presents the DEPARTMENT relation data as an example from the Company Database System.

Table 7.3: Company Database System - DEPARTMENT relation data

Department_ID	Dept_name	Location	Budget
D1	Sales	Skopje	500000
D2	Marketing	Bitola	300000
D3	Human Resources	Ohrid	200000

7.3.1 Attribute in Relational Model

An Attribute is a fundamental component of the relational model, serving as a named column within a relation (table). It defines a specific characteristic or property that describes the entities or relationships represented by the relation (Elmasri & Navathe, 2016).

> **Attribute**: An attribute is a named column of a relation (Codd, 1970; Elmasri & Navathe, 2016).

Think of an attribute as a category of information. For instance, the complete attribute structure of the PROFESSOR entity, shown in **Figure 4.48**, includes Professor_ID as a key, the composite attribute Name (First_name, Last_name), and simple attributes Hire_date, Email, Office, and Academic_title. These attributes describe all the necessary information to identify the professor's record in the relation. In essence, attributes are the building blocks that allow us to structure and categorize the data within our relational database, making it organized and manageable.

Attributes are crucial for:

- **Defining Structure**: They establish the schema of a relation, dictating what kind of data each row will hold.
- **Data Representation**: They hold the individual pieces of data that collectively form a tuple (row).
- **Uniqueness and Relationships**: Attributes, or combinations of them, can serve as keys to uniquely identify rows and link different relations together.

University System Example

Each table represents a **relation** (e.g., STUDENT, COURSE) or a **relationship type** (e.g., ENROLLS_IN). The columns of the table correspond to the **attributes** of that entity or relationship, and each row represents a unique instance (or record) of that entity or relationship. Here are some examples from the University Database System:

- Based on the **STUDENT** relation in Figure 4.41, it contains columns like Student_ID, First_name, Last_name, Birth_date, and address details (Street, City, Zip_code, and Country).
- An **ENROLLS_IN** relation, based on Figure 4.45, representing the relationship between STUDENT and COURSE, contains columns like Student_ID, Course_ID, Enrollment_date, and Grade, as given in Table 7.4.

Table 7.4: University Database System - ENROLLS_IN relation data

Student_ID	Course_ID	Enrollment_date	Grade
S001	CS303	01.10.2025	9
S002	EE201	01.10.2025	7
S002	EE301	01.10.2025	5
S003	MA301	01.10.2025	8
S003	MA401	01.10.2025	5
S004	PH101	01.10.2025	6
S004	PH201	01.10.2025	8
S005	HU201	01.10.2025	10
S005	HU301	01.10.2025	7
S006	CS101	01.10.2025	10
S006	CS201	01.10.2025	8
S006	CS303	01.10.2025	6

Company System Example

In the Company Database System, representing data using tables (relations) follows the same principles as in the University system: each significant entity or relationship becomes its own table.

The *product* list is stored in the **PRODUCT** relation, based on the entity shown in Figure 4.82. This relation contains all items sold by the company, with attributes such as Product_ID (the primary key), Product_name (e.g., "Ajvar"), Description, Category, Price, and Weight for each product, as shown in Table 7.5.

Table 7.5: Company Database System - PRODUCT relation data

Product_ID	Product_name	Description	Category	Price	Weight
1	Ajvar	Traditional Macedonian spread	Food	372	0.5
2	Tavce Gravce	Baked beans dish	Food	434	0.7
3	Rakija	Fruit brandy	Beverage	682	1
4	Mastika	Mastic-flavored liqueur	Beverage	930	1
5	Sirenje	White cheese	Dairy	465	0.4
6	Kajmak	Creamy dairy product	Dairy	620	0.3
7	Ohrid Pearl	Jewelry	Accessories	9300	0.1
8	Macedonian Ruby	Gemstone	Accessories	31000	0.05
9	Filigree	Traditional craft	Souvenir	4650	0.2
10	Galichnik Wedding Costume	Traditional clothing	Apparel	74400	2.5
11	Vranec Wine	Red wine	Beverage	1116	1.5
12	Selsko Meso	Village-style meat dish	Food	868	0.8

To track the storage location of each product, such as where "Ajvar" is stored, the **WAREHOUSE** relation is used. Based on the entity shown in Figure 4.70, this relation stores the Warehouse_ID (the primary key), Location (e.g., "Skopje"), and Capacity for each warehouse.

A single product can be stored in many warehouses, and a single warehouse can store many products. To track which specific product is in which specific warehouse, and how much of it is there, we need a third relation that links them.

- Based on Figure 4.84, this third relation is the **STORED_IN**. It combines the Product_ID from the **PRODUCT** relation and the Warehouse_ID from the **WAREHOUSE** relation.

Crucially, for each specific combination of a product and a warehouse, it also records:

- **Quantity_on_hand**: The number of units of that product currently available in that specific warehouse.
- **Reorder_level**: The minimum stock level for that product in that specific warehouse before a replenishment order is required.

Table 7.6: Company Database System - STORED_IN relation data

Product_ID	Warehouse_ID	Quantity_on_hand	Reorder_level
1	W1	57	10
2	W1	21	6
3	W3	18	4
4	W2	32	15
5	W1	14	6
6	W3	12	9
7	W1	9	7

8	W2	23	16
9	W3	6	1
10	W4	7	3
11	W2	72	10
12	W1	18	6

In this way, the **STORED_IN** relation acts as a bridge, connecting products to their storage locations and providing vital inventory details for each unique product-warehouse combination, as shown in Table 7.6.

7.3.2 The Role of Domain

> **Domain**: A domain is the set of allowable values for one or more attributes (Codd, 1970; Elmasri & Navathe, 2016).

Domains are a crucial feature of the relational model. Each attribute in a relation is defined by a domain, which is a set of permissible values for that attribute. For example, **Table 4.2** illustrates the domain specifications for various attributes in the University system, while **Table 4.3** provides corresponding domain examples from the Company system. This comparison highlights how operational needs of an organization can influence data formats and validation requirements.

The concept of a domain is significant since it enables users to establish a centralized definition for the meanings and origins of the values that attributes can possess. Consequently, the system has access to more information when performing relational operations, helping to prevent semantically incorrect operations. For example, it is not sensible to compare a Salary, from the EMPLOYEE relation, with a Price, from the PRODUCT relation, even though the domain definitions for both attributes are decimal values. On the other hand, the Email attribute from the EMPLOYEE relation and the Email attribute from the CUSTOMER relation are both character types, and they can be legally compared because the two values are from the same domain.

7.3.3 Tuple in Relational Model

> **Tuple**: A tuple is a row of a relation (Elmasri & Navathe, 2016).

A **tuple** in the relational model is a single row of a relation (or table) that represents a set of related data values. Each row in the table represents a tuple or a record in the relation. A tuple is an ordered set of values, one for each attribute (column) of the relation. The value in each column for a given row must originate from the domain of that attribute.

Tuple Definition and Characteristics (Codd, 1970; Elmasri & Navathe, 2016):

- **Relational Model**: In the relational model, data is organized into relations (tables). A relation is a set of tuples.
- **Row Representation**: A tuple is the formal term for what is commonly known as a row or record in a database table.

- **Uniqueness**: Although not always the case in practice (e.g., duplicates can exist in some relations), in a true mathematical relation, all tuples are unique.
- **Order**: The values within a tuple are ordered to correspond to the order of the attributes in the relation's schema.

University System Example

As described in Section **4.7,** attributes capture the descriptive details of the **STUDENT** relation. The STUDENT relation (table) stores information about individual students. Each row, or tuple, represents one specific student and contains values for all the attributes (columns) defined for that relation. The attributes in the sentence "Student Sarah Smith has ID S001, was born on August 25, 1995, and lives in Ohrid" have a specific purpose in the system.

STUDENT (Student_ID, First_name, Last_name, Birth_date, Age, Street, City, Country, Zip_code, Faculty_ID)

Example Tuple (Row):

('S001', 'Sarah', 'Smith', '1995-08-25', 30, 'Jane Sandanski 32', 'Ohrid', 'Macedonia', '6000', 'F001').

This entire row is a single tuple. It represents a student with Student_ID=S001, named Sarah Smith, born on August 25, 1995, with a current age of 30 (in 2025), living at Jane Sandanski 32 in Ohrid, Macedonia, with zip code 6000, belonging to Faculty F001.

Table 7.2 presents the FACULTY relation data from the University Database System, where each row represents a tuple**.** The FACULTY relation (table) stores information about each University faculty. Each row, or tuple, represents one specific faculty and contains values for all the attributes (columns) defined for that relation.

FACULTY (Faculty_ID, Name, Budget, Location)

Example Tuples (Rows):

('F001', 'Computer Science', 1500000.00, 'Main Campus').

This entire row is a single tuple. It represents a faculty with Faculty_ID=F001, named Computer Science, with a budget of 1500000.00, located at the Main Campus.

Company System Example:

In the Company Database System, the EMPLOYEE relation (table) stores information about individual employees. Each row, or tuple, represents one specific employee and contains values for all the attributes (columns) defined for that relation.

Relation EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Country, Zip, Department_ID, Warehouse_ID, Manages_Dept_ID)

Example Tuples (Rows):

('E1', 'Zoran', 'Pejovski', 'zoran.pejovski@email.com', 60000.00, '15.01.2020', 'Partizanska 106', 'Skopje', 'Macedonia', '1000', 'D1', 'W1', 'D1').

This entire row is a single tuple. It represents an employee with Employee_ID E1, named Zoran Pejovski, with email address zoran.pejovski@email.com, earning 60000.00, hired on 15 January 2020, living at Partizanska 106 in Skopje, zip code 1000, Macedonia, and working in Department D1, handling the warehouse W1 and managing Department D1.

Example Tuples (Rows) in the DEPARTMENT Relation

The DEPARTMENT relation (table) stores information about company departments. Each row, or tuple, represents one specific department and contains values for its defined attributes, as shown in Table 7.3:

Relation DEPARTMENT (Department_ID, Dept_name, Location, Budget)

Example Tuples (Rows):

('D1', 'Sales', 'Skopje', 500000.00),

This tuple represents the "Sales" department. Its Department_ID is D1, it's located in Skopje, and it has a budget of 500000.00.

7.3.4 The Degree of Relation

> **Degree**: The degree of a relation is the number of attributes it contains (Elmasri & Navathe, 2016).

The degree of a relation refers to the number of attributes (columns) defined in its schema. It signifies the structural complexity of the relation, indicating how many distinct pieces of information are being recorded for each entity or relationship represented by the table. A relation with a degree of one is called unary, two is binary, three is ternary, and so on (Elmasri & Navathe, 2016). Understanding the degree is crucial for database design, as it dictates the number of data points per tuple (row) and influences how the relation can be joined with others to retrieve comprehensive information. A higher degree can imply more complex entities or intricate relationships.

University System Example

In the *University* Database System, the degree of the STUDENT relation is 10, whereas the degree of the FACULTY relation is 4.

Company System Example

In the *Company* Database System, the degree of the EMPLOYEE relation is 13, whereas the degree of the DEPARTMENT relation is 4.

7.3.5 The Cardinality of Relation

> **Cardinality**: The cardinality of a relation is the number of tuples it contains (Elmasri & Navathe, 2016).

The cardinality of a relation refers to the number of tuples (rows) it contains at a specific point in time. It essentially describes the current size or the number of records in the table. Cardinality is dynamic; it changes as rows are inserted, deleted, or updated. A high

cardinality indicates a large number of unique records, while a low cardinality suggests fewer records. Understanding the cardinality is important for query optimization, as it helps the database system determine the most efficient way to retrieve and process data. It also plays a role in database design, influencing indexing strategies and join performance.

University System Example

In the *University* Database System, the **cardinality** of the FACULTY relation, as shown in Table 7.2, is 5. The cardinality of the ENROLLS_IN relation, as shown in Table 7.4, is 12.

Company System Example

In the *Company* Database System, the **cardinality** of the DEPARTMENT relation, as shown in Table 7.3, is 3. The cardinality of the PRODUCT relation, as shown in Table 7.5, is 12. Relation. The cardinality of the STORED_IN relation, as shown in Table 7.6, is 12.

7.3.6 The Relational Database

> **Relational database**: A collection of normalized relations with distinct relation names (Codd, 1970; Elmasri & Navathe, 2016).

A relational database is a structured collection of normalized relations, each identified by a unique name. Normalization is a crucial process that organizes data to minimize redundancy and improve data integrity by dividing larger tables into smaller, well-structured relations. Each relation (table) contains data about a specific entity or relationship, with rows representing individual instances (tuples) and columns representing attributes. Distinct naming ensures that each relation in the database can be uniquely referenced and manipulated. This structured, normalized organization enables efficient data storage, retrieval, and manipulation via relational algebra (Chapter 8) and SQL (Chapter 9), forming the foundation of a robust, consistent DBMS.

7.3.7 Characteristics of Relations

Ordering of Tuples in a Relation. A relation is defined as a set of tuples. Mathematically, elements of a set have no order among them; hence, tuples in a relation do not have any particular order (Elmasri & Navathe, 2016).

According to the preceding definition of a relation, an ***n-tuple* is an ordered list of n values**, so the ordering of values in a tuple, and hence of attributes in a relation schema, is important. However, at a more abstract level, the order of attributes and their values is not that important as long as the correspondence between attributes and values is maintained. This makes sense at an abstract level, since there really is no reason to prefer having one attribute value appear before another in a tuple.

University System Example:

The FACULTY relation (table) stores information by Faculty_ID, but it could be ordered by faculty Name, as shown in **Table 7.7.**

Table 7.7: University Database System - FACULTY relation data (sorted by Name)

Faculty_ID	Name	Budget	Location
F004	Bioengineering	1100000	North Building
F001	Computer Science	1500000	Main Campus
F003	Economy & Tourism	900000	West Wing
F002	Electrical Engineering	1200000	East Wing
F005	Humanities	800000	South Building

In the formal relational model, a relation is defined as a set of tuples. By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for all their attributes. Usually, there are other subsets of attributes of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for those attributes.

Values and NULLs in the Tuples. Each value in a tuple is atomic, meaning it cannot be split into components within the domain of the basic relational model.

An important concept is NULL values, which represent attribute values that may be unknown or not applicable to a tuple. A special value, called NULL, is used in these cases. For example, in Table 7.8, some PROFESSOR tuples have NULL for their Heads_Faculty_ID, because they do not head any faculty. For example, Faculty_ID F001 is headed by Professor_ID P004, and professors with Professor_ID P001 and P006 are only members of Faculty_ID F001, as indicated by NULL values in the Heads_Faculty_ID attribute. Professor_ID P002 is both a member of and heads Faculty_ID F002. There is a special case where Professor_ID P003 is a member of Faculty_ID F004 but does not head any faculty (Heads_Faculty_ID = NULL). Furthermore, since no other professor in the table has Heads_Faculty_ID = F004, Faculty F004 currently has no assigned head.

In general, NULL values can have several meanings, such as an unknown value, a value that exists but is unavailable, or an attribute that does not apply to this tuple (also known as value undefined).

Table 7.8: University System - PROFESSOR relation

Professor_ID	First_name	Last_name	Academic_title	Office	Email	Hire_date	Faculty_ID	Heads_Faculty_ID
P001	Kliment	Ohridski	Full Professor	A201	kliment.ohridski@univeristy.edu	01.08.2007	F001	NULL
P002	Naum	Ohridski	Associate Professor	B305	naum.ohridski@univeristy.edu	15.01.2004	F002	F002
P003	Kiril	Prosvetitel	Full Professor	C102	kiril.prosvetitel@univeristy.edu	25.09.2001	F004	NULL
P004	Metodij	Prosvetitel	Full Professor	D404	metodij.prosvetitel@univeristy.edu	13.09.2002	F001	F001
P005	Blaze	Koneski	Assistant Professor	E203	blaze.koneski@univeristy.edu	19.10.2012	F005	F005
P006	Richard	Johnson	Associate Professor	E206	richard.johnson@univeristy.edu	02.09.2022	F001	NULL

7.3.8 Relational Model Notation

The following notation for the relational model will be used (Elmasri & Navathe, 2016):

- A relation schema *R* of degree *n* is denoted by *R(A1, A2, ... , An).*
- The uppercase letters *R, S, C, E, D* denote relation names.
- The lowercase letters *q, r, s* denote relation states.
- The letters *x, y, u, v* denote tuples.

In general, the name of a relation schema, such as FACULTY, also indicates the current set of tuples in that relation, the current relation state, whereas FACULTY (Faculty_ID, Name, Budget, Location) refers only to the relation schema.

An attribute A can be qualified with the relation name R to which it belongs by using the **dot** notation R.A, for example, FACULTY.Name or FACULTY.Location. This is because the same name may be used for two attributes in different relations. However, all attribute names in a particular relation must be distinct.

An *n-tuple* t in a relation ***r(R)*** is denoted by *t* = <*v1, v2, ... , vn*>, where vi is the value corresponding to attribute *Ai*.

As an example, consider the tuple *t = <'F004', 'Bioengineering', 1100000, 'North Building'>* from the FACULTY relation in Table 7.2. We have *t[Name] = <'Bioengineering'>*, and *t[Budget, Location] = <1100000,'North Building'>.*

7.4 Database Relation

Database relations form the mathematical foundation of relational database management systems, establishing structured connections between data entities through precisely defined rules and constraints. At its core, a database relation is a formal structure in which data is organized into rows (tuples) and columns (attributes), governed by principles derived from set theory and predicate logic (Codd, 1970; Date, 2003). Mathematical relations are essential because they provide a rigorous, formal framework that ensures data consistency, eliminates ambiguity, and enables complex query operations (Date, 2003; Codd, 1990). This mathematical underpinning guarantees predictable behavior and reliable data manipulation across all database operations (Silberschatz et al., 2019).

Database relations possess several critical properties that maintain data integrity and coherence. Key attributes serve as unique identifiers, ensuring each tuple can be distinctly recognized within a relation (Elmasri & Navathe, 2016). Highlighting these constraints helps students see how systems prevent errors and maintain data coherence. Entity integrity enforces that primary keys remain non-null and unique, preserving the fundamental identity of each record (Silberschatz et al., 2019). Referential integrity maintains logical connections between related tables, preventing orphaned records and ensuring data dependencies remain valid (Connolly & Begg, 2015). Foreign keys establish inter-table relationships, creating a structured set of inter-table connections that ensure data remains consistent and meaningful across the entire database (Teorey et al., 2011; Hristoski & Dimovski, 2022).

7.4.1 Mathematical Relation

At the heart of relational database systems lies the concept of **mathematical relations**, which serve as the foundation for effectively organizing and managing data. In this framework, a relation is represented as a table, where each row, called a tuple, corresponds to an individual element. Each column, called an attribute, corresponds to a specific domain. Edgar Codd's work used set theory and predicate logic to define operations on these relations, which ultimately led to the development of relational algebra (Codd, 1970). Relational algebra, as described in Chapter 8, defines operations such as selection, projection, union, intersection, and join that are rooted in set theory, enabling robust, logical data manipulation. This mathematical foundation ensures data *integrity*, *consistency*, and precise *querying* capabilities in a relational database.

Relational database systems are based on the mathematical concept of a relation. In mathematics, a **relation** describes a connection or association between the elements of one or more sets. It defines how elements from one set (the domain) relate to elements in another set (the target domain). Essentially, a **relation** is a set of ordered pairs, where the first element comes from the domain and the second element comes from the target domain. The Cartesian product of these domains creates all possible combinations of these values, and a relation only includes a subset of these combinations. Each domain represents a set of atomic values (e.g., integers, strings, dates).

In a relational model, this concept of **mathematical relation** translates directly as:

- A relation (or table) is a collection of tuples (rows).
- Each tuple is an ordered list of values, where each value corresponds to an attribute (column).
- Each attribute draws its values from a specific domain (data type).

University System Example:

- Let S be the domain of Student_IDs = {S001, S006}
- Let C be the domain of Course_IDs = {CS101, CS303}

The Cartesian Product S x C would be

S x C = {(S001, CS101), (S001, CS303), (S006, CS101), (S006, CS303)}.

A relation, such as ENROLLS_IN, is then a subset of this Cartesian product. For instance, if we consider a subset

R1 = {(S001, CS303), (S006, CS101), (S006, CS303)},

this represents a relation where student S001 is enrolled in course CS303, and student S006 is enrolled in courses CS101 and CS303. If we observe that the common characteristic of all pairs where the second element is CS303, then we could write:

R1 = {(x, y) | x ∈ S, y ∈ C, and y = CS303} = {(S001, CS303), (S006, CS303)},

this notation means "the set of all ordered pairs (x, y) that are members of the Cartesian product S × C such that the second element, y, is equal to CS303."

Another example of a subset representing actual student-course pairings could be {(S001, CS303), (S006, CS101), (S006, CS303)}. This mathematical rigor ensures data within a relational database is structured, consistent, and adheres to well-defined rules, making relational databases robust and reliable.

Company System Example:

- Let E be the domain of Employee_ID= {E1, E2, E3, E4, E5, E6, E7, E8}
- Let D be the domain of Department_IDs = {D1, D2, D3}

The Cartesian product E × D would contain all 24 possible combinations (8 employees × 3 departments), for example:

E x D = {(E1,D1), (E1,D2), (E1,D3), (E2,D1), (E2,D2), (E2,D3), (E3,D1), (E3,D2), (E3,D3), ...}.

A relation, such as WORKS_FOR, is then a subset of this Cartesian product. For instance, if we consider a subset

R2 = {(E1, D1), (E2, D1), (E7, D1)},

this represents a relation in which employees E1, E2, and E7 all work in department D1. The ordered pairs belonging to a relation can be specified by stating the conditions for their selection. Observing that the common characteristic of all pairs in R2 is that their second element is D1, we could write R2 as:

R2 = {(x, y) | x ∈ E, y ∈ D, and y = D1},

This notation means "the set of all ordered pairs (x, y) that are members of the Cartesian product E × D such that the second element, y, is equal to D1."

7.4.2 Relational Database Schema

> **Relational database schema**: A named relation defined by a set of attribute-domain pairs (Elmasri & Navathe, 2016).

In the context of a relational DBMS, a **database relation** is a fundamental concept representing a two-dimensional table. Each relation stores a collection of unique records, or **tuples** (rows), that share a common structure. This structure is defined by its named columns, called **attributes** (Elmasri & Navathe, 2016). For instance, in a *University* Database System, the **STUDENT** relation holds all student records, while in a *Company* Database System, the **EMPLOYEE** relation contains all employee records. This tabular organization allows for clear, structured storage and retrieval of data, forming the backbone of relational databases.

The **relational database schema** defines the structure or blueprint of a relation (table). It specifies the name of the relation, along with the names and data types (domains) of all its attributes. Let A1, A2,…, An be the attributes with domains D1, D2,…, Dn. Then the set {A1 :D1, A2:D2,…, An:Dn} is a relation schema. A relation R defined by a relational database schema S is a set of mappings from the attribute names to their corresponding domains. Thus, relation R is a set of *n-tuples* (or rows).

University System Example

The relation schema defines the structure or blueprint of a relation (table). It specifies the name of the relation, along with the names and data types (domains) of all its attributes.

Based on the completed entity example in Figure 4.10, the **relational database schema** for a STUDENT relation is:

STUDENT (Student_ID: varchar, First_name: varchar, Last_name: varchar, Birth_date: date, Age: int, Street: varchar, City: varchar, Country: varchar, Zip_code: varchar, Faculty_ID: varchar)

Furthermore, based on the completed UNIVERSITY entity examples in subsection 4.9.3, the **relational database schemas** are:

FACULTY (Faculty_ID: varchar, Name: varchar, Budget: decimal, Location: varchar)

EMAIL (Student_ID: varchar, Email: varchar)

PROFESSOR (Professor_ID: varchar, First_name: varchar, Last_name: varchar, Office: varchar, Email: varchar, Hire_date: date, Academic_title: varchar, Faculty_ID: varchar, Heads_Faculty_ID: varchar)

COURSE (Course_ID: varchar, Title: varchar, Credits: int, Description: varchar, Faculty_ID: varchar, Professor_ID: varchar)

PREREQUISITE (Course_ID: varchar, Prerequisite_Course_ID: varchar)

ENROLLS_IN (Student_ID: varchar, Course_ID: varchar, Enrollment_date: date, Grade: int)

CLASS_SESSION (Course_ID: varchar, Session_number: int, Year: int, Room: varchar, Semester: varchar, Time: time, Day: varchar, Duration: int)

ATTENDS (Student_ID: varchar, Course_ID: varchar, Session_number: int, Year: int, Attendance: varchar)

TEXTBOOK (ISBN: varchar, Title: varchar, Author: varchar, Edition: varchar, Publisher: varchar, Available_copies: int)

ASSIGNMENT (Professor_ID: varchar, Course_ID: varchar, ISBN: varchar)

The detailed process for deriving these schemas from ER and EER diagrams is explained step by step in Section 7.5. The schemas above represent the complete University relational model, including relations for entities, multivalued attributes, and many-to-many relationships.

The schema ensures consistency by dictating the permissible values and the overall structure for all tuples in that relation.

Company System Example

Based on the completed entity example in Figure 4.10, the **relational database schema** for an EMPLOYEE relation is:

EMPLOYEE (Employee_ID: varchar, First_name: varchar, Last_name: varchar, Email: varchar, Salary: decimal, Hire_date: date, Street: varchar, City: varchar, Country: varchar, Zip_code: varchar, Department_ID: varchar, Warehouse_ID: varchar, Manages_Dept_ID: varchar)

Furthermore, based on the completed Company entity examples, in subsection 4.9.6, the **relational database schemas** are:

DEPARTMENT (Department_ID: varchar, Dept_name: varchar, Location: varchar, Budget: decimal)

WAREHOUSE (Warehouse_ID: varchar, Location: varchar, Capacity: int, Department_ID: varchar)

CUSTOMER (Customer_ID: varchar, First_name: varchar, Last_name: varchar, Email: varchar, Join_date: date, Street: varchar, City: varchar, Country: varchar, Zip_code: varchar)

PRODUCT (Product_ID: int, Product_name: varchar, Description: varchar, Category: varchar, Price: decimal, Weight: decimal)

SUPPLIER (Supplier_ID: varchar, Supplier_name: varchar, First_name: varchar, Last_name: varchar, Email: varchar, Street: varchar, City: varchar, Country: varchar, Zip_code: varchar)

ORDER (Order_ID: varchar, Status: varchar, Order_date: date, Required_date: date, Shipped_date: date, Employee_ID: varchar, Customer_ID: varchar)

ORDER_LINE (Order_ID: varchar, Line_number: varchar, Unit_price: decimal, Quantity: int, Discount: decimal, Product_ID: int)

EMPLOYEE_PHONE (Employee_ID: varchar, Phone_number: varchar)

CUSTOMER_PHONE (Customer_ID: varchar, Phone_number: varchar)

SUPPLIER_PHONE (Supplier_ID: varchar, Phone_number: varchar)

SUPPLIES (Product_ID: int, Supplier_ID: varchar, Supply_date: date, Unit_cost: decimal)

STORED_IN (Product_ID: int, Warehouse_ID: varchar, Quantity_on_hand: int, Reorder_level: int)

Note: The attributes First_name and Last_name in the SUPPLIER relation represent the components of the composite attribute Contact_person from the SUPPLIER entity in the ER diagram. In the complete EER relational schema presented in Section 7.7.5, these attributes are renamed to Contact_first_name and Contact_last_name to preserve the semantic clarity of the original composite attribute name.

The detailed process for deriving these schemas from ER and EER diagrams is explained step by step in Section 7.5. The schemas above represent the complete Company relational model, including relations for entities, multivalued attributes, and many-to-many relationships.

The schema ensures consistency by dictating the permissible values and the overall structure for all tuples in that relation.

7.4.3 Description of Relation Instance

A relation instance (or relation state) refers to the actual set of tuples (rows) present in a relation at a specific moment in time. Each element in an *n*-tuple consists of an attribute and a corresponding value for that attribute. According to the tuple definition, there is a mapping set of (<attribute>, <value>) pairs, where each pair specifies the mapping from attribute Ai to value vi in D_i. When we visualize a relation as a table, we typically list the attribute names as column headings and write out the tuples as rows, having the form $(d_1,d_2,...,d_n)$, where each value d_i is taken from the appropriate domain D_i of attribute A_i. In this way, we can think of a relation in the relational model as any subset of the Cartesian product of its attribute domains. A table is simply a physical representation of such a relation instance.

University System Example

For example, in a *University* Database System, the COURSE relation has attributes Course_ID, Title, Credits, and Description, each with its corresponding domain. A COURSE relation instance is any subset of the Cartesian product of these domains, or any set of four-tuples in which the first element is from the domain Course_ID, the second from Title, and so on.

One such four-tuple (representing a single course record) is:

(Course_ID: CS303, Title: Database, Credits: 6, Description: Database course covers database fundamentals, design, SQL, and practical applications, preparing students for data management challenges in various fields.)

The COURSE table is a convenient way to represent all the four-tuples that form the relation at a specific moment in time, which explains why table rows in the relational model are called tuples. In the same way that a relation has a schema, so too does the relational database.

Company System Example

For example, in a *Company* Database System, the **DEPARTMENT** relation has attributes Department_ID, Dept_name, Location, and Budget, each with its corresponding domain. A DEPARTMENT relation instance is any subset of the Cartesian product of these domains, or any set of four-tuples in which the first element is from the domain Department_ID, the second from Dept_name, and so on.

One such four-tuple (representing a single department record) is:

(Department_ID: D3, Dept_name: Human Resources, Location: Ohrid, Budget: 200000.00)

7.4.4 Properties of Database Relation

A database relation has several key properties (Codd, 1970; Elmasri & Navathe, 2016)**:**

- Each relation must have a **unique** name distinct from all other relation names in the relational schema.
- Each cell in the relation should contain exactly **one atomic** (single) value.
- Each attribute must have a distinct name, and the values of an attribute must all come from the same domain.
- Each tuple within the relation must be unique, meaning there can be no duplicate tuples.
- The order of attributes and tuples is not significant

To illustrate these restrictions, consider the FACULTY relation. Each cell should contain only one value; for example, storing two Name values for a single faculty record in one cell is not allowed. Relations cannot contain repeating groups; a relation that complies with this requirement is said to be in first normal form.

The column names at the top of each column refer to the attributes of the relation. The values in the Name attribute must come exclusively from the FACULTY Name domain; thus, a Location value should not appear in that column. Additionally, no tuples can be duplicates in a relation. For instance, the row (F001, Computer Science, 1500000, Main Campus) should appear only once.

As long as an attribute name is moved together with its corresponding values, the order of the columns in a table can be changed without altering the meaning of the relation. For example, the table would still represent the same relation even if the Location attribute is placed before the Budget attribute. However, for readability, it is often better to maintain a logical order. Similarly, tuples can be interchanged; for example, the records for F005 and F004 can be switched without changing the relation.

Most of the properties specified for relations derive from the characteristics of mathematical relations. When we derive the Cartesian product of sets with simple, single-valued elements, such as integers, each element in the tuples is single-valued. In this manner, each cell of a relation also contains only one value. However, it is important to note that a mathematical relation does not necessarily need to be normalized. E.F. Codd, who developed the relational model (Codd, 1970), prohibited repeating groups in order to simplify the relational data model.

In a relation, the possible values for a given position are determined by the set, or domain, corresponding to that position. Furthermore, the values in each column must originate from the same attribute domain. Since a set does not contain repeated elements, a relation must also have no duplicate tuples. Being a set, a relation disregards the order of its elements; therefore, the order of tuples is immaterial.

In contrast, within a mathematical relation, the order of elements in a tuple is significant. For example, the ordered pair (A1, A2) differs from (A2, A1). In the relational model, the order of attributes is not significant. The column headings indicate which attribute

corresponds to each value, meaning that the sequence of these headings does not affect the intension (the schema structure). However, once the structure of the relation is established, the order of elements within the tuples (the extension) (the actual data) must align with the order of the attribute names.

7.4.5 Relational Keys

A key attribute uniquely identifies each tuple in a relation. Therefore, we need attributes (called relational keys) that uniquely identify each tuple.

> **Key Definition**: A key is an attribute (or set of attributes) that uniquely identifies a tuple (row) in a table.

7.4.5.1 The Role of Keys in Relational Databases

Keys are fundamental components of relational databases that serve multiple critical functions. They uniquely identify each record within a table, ensuring no two rows are identical and preventing duplicate entries. Keys establish relationships between tables, enabling the connection of related data across the database. They maintain data integrity by enforcing rules that keep information accurate and consistent. Additionally, keys dramatically improve query performance by creating indexes that enable the database to quickly locate and retrieve specific records. Without keys, databases would lack organization, suffer from data anomalies, and experience slow performance. They form the backbone of efficient database design, ensuring reliable, scalable data management for applications of any size.

Table 7.9: Fundamental Types of Keys

Type of Key	Description
Candidate Key	*Minimal set of attributes that can uniquely identify a tuple*
Primary Key (PK)	*Selected candidate key to uniquely identify tuples*
Foreign Key (FK)	*An attribute that references the primary key in another table*
Composite Key	*Key consisting of multiple attributes*
Superkey	*Any set of one or more attributes that can uniquely identify a tuple in a relation*
Alternate Key	*Candidate keys not chosen as the primary key*

A **superkey** is an attribute, or set of attributes, that uniquely identifies a tuple within a relation (Elmasri & Navathe, 2016). While a superkey guarantees uniqueness, it may contain unnecessary attributes. For example, in the COURSE relation (University Database), (Course_ID, Title, Credits) is a superkey, but (Course_ID) alone is also unique. In the EMPLOYEE relation (Company Database), (Employee_ID, First_name, Last_name) is a superkey.

Superkey and Key Properties

- **Unique Property:** Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key.
- **Minimality Property:** It is a minimal superkey from which we cannot remove any attributes and still have the uniqueness constraint hold. This is required for a key but is optional for a superkey.

A key is determined by the meaning of the attributes, and the property is time-invariant: it must continue to hold even after we insert new tuples into the relation. For example, we should not designate only one of the Last_name attributes of the PROFESSOR relation in Table 7.8 as a key, because two professors with identical Last_name may exist, i.e. "Ohridski" in a valid state. In general, a relation schema may have more than one key. In this case, each key is called a **candidate key.**

A **candidate key** is a superkey such that no proper subset is also a superkey. It possesses both uniqueness (each tuple is uniquely identified) and irreducibility (no attribute can be removed without losing uniqueness) (Elmasri & Navathe, 2016).

University System Example

For the PROFESSOR relation, Professor_ID is a candidate key.

For FACULTY, both Faculty_ID and Name (if unique) could be candidate keys.

Company System Example

In the EMPLOYEE relation, Employee_ID and Email are candidate keys.

For DEPARTMENT, both Department_ID and Dept_name are candidate keys. However, it is common to designate one of the candidate keys as the primary key of the relation.

The **primary key** is the chosen candidate key that uniquely identifies tuples within a relation (Codd, 1970; Elmasri & Navathe, 2016). It cannot contain NULL values and is the principal identifier.

Notation: Primary keys are **underlined** in relational schemas

Primary Key Definition: An attribute (or combination of attributes) that uniquely identifies each row in a table.

A primary key is the cornerstone of table identification, governed by strict rules that ensure data reliability and integrity. Every primary key value must be unique across all rows in the table; this guarantees that each record can be distinctly identified without ambiguity. It cannot contain NULL values, as every record must have a definitive identifier. The primary key should remain stable over time, meaning its value should not change once assigned, preserving consistent references and relationships. Furthermore, each table should have exactly one primary key, which may consist of a single column or multiple columns forming a composite key. These rules work together to create a solid foundation for data organization, enabling efficient indexing, reliable relationships with other tables, and the maintenance of the database's overall structural integrity.

Primary Key Rules:

- Must be UNIQUE for each row
- Cannot be NULL (must have a value)
- Should be stable (value doesn't change)
- Each table should have exactly ONE primary key

University System Example

For COURSE, Course_ID would typically be chosen as the primary key. For FACULTY, Faculty_ID would be the primary key.

COURSE (Course_ID, …) **PROFESSOR** (Professor_ID, …)

Company System Example:

In the EMPLOYEE relation, Employee_ID is usually selected as the primary key. For DEPARTMENT, Department_ID is chosen.

DEPARTMENT (Department_ID, …) **EMPLOYEE (**Employee_ID, …)

A **foreign key** is an attribute, or set of attributes, within one relation that matches the candidate key of another (possibly the same) relation (Elmasri & Navathe, 2016). It establishes a link between tuples of two relations, enforcing referential integrity.

Notation: Foreign keys are marked with an asterisk (*)

Foreign Key: An attribute (or set of attributes) in one table that references **the primary key** of another table.

A foreign key is a crucial database constraint that establishes meaningful connections between tables by referencing the primary key of another table. Its primary purpose is to establish and maintain relationships between related data stored across different tables, enabling complex data structures while preventing inconsistencies. Foreign keys enforce referential integrity by ensuring that relationships between tables remain valid—a record cannot reference a non-existent entry in another table. This prevents orphaned records and maintains data accuracy throughout the database. By linking related information across multiple tables, foreign keys enable normalized database designs that efficiently organize data without redundancy. They allow queries to retrieve associated data from multiple tables simultaneously, supporting the relational model's core principle of connecting disparate information through well-defined relationships.

University System Example:

In the COURSE relation, Faculty_ID* is a foreign key referencing FACULTY.Faculty_ID and Professor_ID* is a foreign key referencing PROFESSOR. Professor_ID.

COURSE (Course_ID, …, Faculty_ID*, Professor_ID*)

In the PROFESSOR relation, Heads_Faculty_ID* is a foreign key referencing FACULTY.Faculty_ID and Faculty_ID* is a foreign key referencing FACULTY.Faculty_ID

PROFESSOR (Professor_ID, …, Faculty_ID*, Heads_Faculty_ID*)

Company System Example:

In the EMPLOYEE relation, Department_ID* is a Foreign Key referencing DEPARTMENT.Department_ID.

EMPLOYEE (Employee_ID, …, Department_ID*, Warehouse_ID*, Manages_Dept_ID*)

7.4.6 Entity Integrity, Referential Integrity, and Foreign Keys

The **entity integrity** constraint states that no primary key value can be NULL (Codd, 1970; Elmasri & Navathe, 2016). This is because the primary key value uniquely identifies each tuple in a relation. Having NULL values in the primary key means we cannot identify some tuples. For example, if two or more tuples had NULL for their primary keys, we may not be able to distinguish them if we try to reference them from other relations.

Key constraints and **entity integrity** constraints are specified on individual relations. The **referential integrity** constraint is specified between two relations and is used to maintain the consistency among tuples in the two relations (Elmasri & Navathe, 2016).

> **What is Referential Integrity?** A constraint that ensures relationships between tables remain consistent.

Referential integrity is a fundamental concept in relational databases. It means that if a row in one table refers to a row in another table, the referenced row in the second relation must exist. This concept is often explained with the term foreign key. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relations E and D. A set of attributes FK in relation schema E is a foreign key of E that references relation D if it satisfies the following rules:

1. The attributes in FK have the same domain(s) as the primary key attributes PK of D; the attributes FK are said to reference or refer to the relation D.

2. A value of FK in a tuple d1 of the current state r1(E) either occurs as a value of PK for some tuple d2 in the current state r2(D) or is NULL. In the former case, we have d1[FK] = d2[PK], and we say that **d1 references tuple d2**.

In this definition, E is called the **referencing relation**, and D is the **referenced relation**. If these two conditions hold, a referential integrity constraint from E to D is said to hold. In a database of many relations, there are usually many referential integrity constraints.

To specify these constraints, first, we must have a clear understanding of the meaning or role that each attribute or set of attributes plays in the various relations of the database. Referential integrity constraints typically arise from the relationships among the entities represented by the relation.

Referential integrity is a fundamental database principle that ensures relationships between tables remain valid and consistent throughout all operations. It prevents orphaned

records—situations in which a foreign key references a nonexistent primary key in the parent table, leaving the data disconnected or meaningless. By enforcing referential integrity, databases automatically maintain data consistency across related tables, rejecting operations that would violate established relationships. This creates a reliable framework where every foreign key value either matches an existing primary key or is NULL when permitted. Referential integrity safeguards against data anomalies that could corrupt the database structure, ensuring that deletions, updates, and insertions respect the logical connections between tables. Without it, databases would accumulate invalid references, leading to inaccurate queries, unreliable reports, and compromised data quality, undermining business decisions and application functionality.

When an operation threatens **referential integrity**, databases offer several strategies to resolve the conflict. CASCADE automatically propagates delete or update operations to all related records in child tables, maintaining consistency by removing or updating dependent data as the parent record is deleted or updated. RESTRICT prevents the operation entirely, blocking any deletion or update that would create orphaned records until the relationships are manually resolved. SET NULL assigns NULL values to the affected foreign keys, preserving child records while removing the specific relationship reference. SET DEFAULT replaces the foreign key with a predefined default value, maintaining the record while redirecting the relationship to a fallback entry. Each action serves different business requirements: CASCADE suits tightly coupled data, RESTRICT protects critical relationships, while SET NULL and SET DEFAULT offer flexible alternatives that preserve records while managing broken references appropriately.

Table 7.10: Database Actions on Violation

Type of Operation	Description
CASCADE	*Delete/update related records automatically*
RESTRICT	*Prevent the operation*
SET NULL	*Set FK to NULL*
SET DEFAULT	*Set FK to the default value*

Referential Integrity Rule: Every foreign key value must either **match** a primary key value in the referenced table, OR be **NULL** (if NULLs are allowed) (Codd, 1970; Elmasri & Navathe, 2016).

University System Example:

The Heads_Faculty_ID attribute in the PROFESSOR relation is a foreign key that references the FACULTY relation — specifically, FACULTY.Faculty_ID. It indicates which faculty a professor heads, if any. As shown in Table 7.8, the tuple for professor P004 (Metodij Prosvetitel) has Heads_Faculty_ID = 'F001', meaning that professor Metodij Prosvetitel is the head (dean) of Faculty F001 (Computer Science). Professors P001 (Kliment Ohridski) and P006 (Richard Johnson) have Heads_Faculty_ID = NULL, indicating that they do not head any faculty — they are members of Faculty F001 but hold no dean role.

Company System Example:

For example, consider the relationship between EMPLOYEE and DEPARTMENT, described in Case Study 2: Company Database System. In the EMPLOYEE relation, the Department_ID attribute refers to the department for which an employee works; therefore, we designate Department_ID as a foreign key in EMPLOYEE referencing the DEPARTMENT relation. For example, in Section 7.3.3, the tuple for the employee with Employee_ID= E1 references the tuple for the 'Sales' department, indicating that 'Zoran Pejovski' works for this department.

Entity integrity, referential integrity, and *foreign keys* are essential constraints that ensure accuracy and consistency within a relational model. *Entity integrity* requires that each table has a primary key and that no part of that key can be null, ensuring the **unique** identification of records. *Referential integrity* ensures that foreign keys in a table correctly reference **existing primary keys** in related tables, maintaining valid relationships between data. *Foreign keys*, therefore, enforce **links** between tables and prevent orphaned records. These constraints are defined in the database schema using Data Definition Language (DDL) (covered in Chapter 9), enabling the database system to automatically enforce rules and maintain data reliability. Together, they form the foundation of data integrity and support logical structure across the database system.

7.4.7 Summary of Relational Model Fundamentals

The **relational model** provides a robust foundation for structuring, manipulating, and maintaining data integrity in a database system (Elmasri & Navathe, 2016). It is based on formal modeling concepts, including domains, attributes, tuples, and relation schema. A **domain** defines the set of possible values an attribute can take. A **relation schema** is a list of attributes that describes the structure of a relation, while a tuple is an ordered list of attribute values that conforms to this schema. A relation (or relation state) is a set of tuples, which collectively form the data in a relation. Unlike ordinary tables or files, relations are characterized by their independence from the ordering of both tuples and attributes.

The model introduces **NULL** values to represent missing or unknown information, but it's best to use them sparingly to maintain data consistency. We ensure data integrity through constraints, which fall into three categories: model-based, schema-based, and semantic (business rule) constraints (Elmasri & Navathe, 2016). Key schema-based constraints include entity integrity, which requires that primary key attributes cannot be NULL, and referential integrity, which ensures that foreign key values in one relation match primary key values in another relation.

7.5 ER and EER to Relational Model Mapping

The database design process is a structured progression from abstract ideas to technical implementation, as shown in Figure 7.1. It begins with **requirements analysis**, where designers identify exactly "what" data needs to be stored and how users will interact with it.

As we learned in Chapter 6, just as a dream home cannot be built from blueprints alone, the database design process requires a structured progression from abstract ideas to

technical implementation. The process starts with **conceptual design**, resulting in an ER/EER diagram that visually represents entities and their relationships in a form easy for stakeholders to understand.

Next, **logical design** translates those visual concepts into a relational schema, defining the specific implementation rules, keys, and constraints. Emphasizing that each step is part of a clear, step-by-step process can help the audience feel reassured and confident in their ability to master database design. Finally, the **physical design** focuses on the actual database, where the schema is converted into database tables optimized for a specific storage system. Each step ensures that the final database is accurate, efficient, and aligned with user needs.

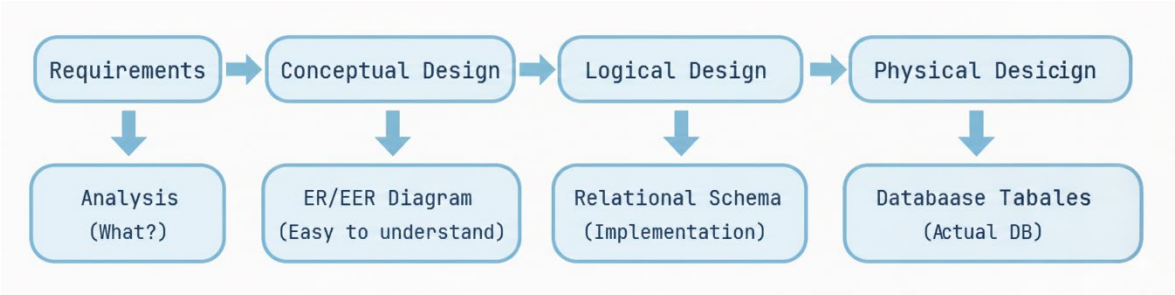


Figure 7.1: Complete Database Design Process

7.5.1 Why Do We Need Mapping?

The transition from a **logical design** to a functional database requires a formal bridge. Mapping is essential because it translates human-centric abstractions into database-implementable structures. While an Entity-Relationship (ER) model and an Enhanced Entity-Relationship (EER) model excel at capturing business requirements and data semantics, they cannot be directly processed by a relational DBMS. Mapping ensures that the rich associations and hierarchies, including specialization hierarchies from EER modeling, defined during the design phase are preserved during the logical design phase. Without a systematic mapping process, there is a significant risk of data redundancy, data integrity loss, or a schema that does not accurately reflect the original business logic. By following formal mapping procedures, database designers ensure that the resulting relational schema is normalized, supports complex queries, and maintains a consistent and authoritative representation of the data (Elmasri & Navathe, 2016).

Key concepts in mapping

- ER/EER Diagrams are conceptual - easy for humans to understand
- Relational model is logical - what databases actually implement
- Systematic transformation - ensures no information is lost
- Industry practice - standard approach worldwide

7.5.2 The Conceptual to Logical Transition

The core of database design lies in the movement from ER and EER diagrams to the relational model. ER and EER diagrams are strictly conceptual; they use symbols such as

rectangles and diamonds to provide a high-level representation of data that is easy for both technical and non-technical users to understand. This stage focuses on communication and on capturing the "big picture" of organizational data. However, relational DBMS software does not "understand" diamonds or lines. It operates on the relational model, which is logical, and data is organized into tuples and attributes within relations.

The mapping process acts as a systematic transformation. It is not a creative or arbitrary step, but rather a set of mathematical and logical rules that ensure no information or semantic meaning is lost during the translation (Elmasri & Navathe, 2016). For example, a relationship in an ER diagram must be converted into a specific mechanism in the relational model, such as a foreign key, to ensure the connection between data points remains intact. Similarly, a specialization hierarchy from an EER diagram must be converted using one of several mapping strategies, each of which preserves the inheritance and constraint structure of the original model. This ensures the connection between data points remains intact. This process is the standard industry practice worldwide. Whether a developer is working at a small startup or a global tech giant, the methodology for converting an ER or EER model into a relational schema remains consistent. This standardization allows different teams to collaborate effectively. It ensures that databases built by one engineer can be maintained or scaled by another using universal principles of relational algebra and design logic.

7.5.3 What Happens During Mapping?

During the mapping process, every element of the ER/EER diagram is translated into its relational equivalent. The most fundamental transformation is converting entities into relations. Each distinct entity type (e.g., STUDENT or COURSE) becomes a relation in the database. Consequently, the attributes associated with those entities are mapped directly to columns within those relations. For instance, the Name and Birth_date attributes of the STUDENT entity become columns in the STUDENT relation, each with a specific data type.

Handling Relationships is more complex and depends on the association's cardinality. In simple cases, a relationship is represented by a foreign key, where the primary key of one relation is placed into another to create a link. However, in many-to-many (M:N) scenarios, a simple link is not enough; designers must create junction relations (also known as associative entities) that hold the primary keys of both participating entities as a composite primary key.

EER constructs introduce additional mapping considerations. Specialization hierarchies, in which a superclass is specialized into one or more subclasses, must be converted using one of several mapping strategies. The choice of strategy affects how inheritance, constraints, and subclass-specific attributes are represented in the relational schema, as discussed in detail in Section 7.5.6.

Finally, Constraints are translated into integrity rules. This includes defining primary keys to ensure entity integrity and foreign keys to enforce referential integrity. These rules ensure that the database remains consistent—for example, preventing a record from being deleted if another relation is still referencing it. This phase essentially encodes the business rules

captured in the ER/EER diagram into database integrity rules, ensuring the data remains valid and reliable throughout its lifecycle.

To help visualize the transition from a conceptual model to a logical database structure, Table 7.11 summarizes how each ER/EER component is transformed during the mapping process.

Table 7.11: ER/EER to Relational Mapping Reference Table

ER Element	Relational Element	Description
Strong Entity	**Relations**	Each strong entity (e.g., COURSE) becomes a relation with its own primary key
Weak Entity	**Relation with composite key**	A weak entity becomes a relation whose primary key combines its partial key with the primary key of the owner entity
Simple Attributes	**Column**	Descriptive properties (e.g., Name, Location) become columns in the relation
Composite Attribute	**Multiple columns**	Each component of a composite attribute (e.g., First_name, Last_name) becomes a separate column
Derived Attribute	**Typically omitted**	Derived attributes (e.g., Age) are usually not stored; they are calculated from stored attributes when needed
Multivalued Attribute	**New relation**	Attributes with multiple values (e.g., Email, Phone) are moved to a separate relation to avoid redundancy
Key Attribute	**Primary Key**	The unique identifier (e.g., Student_ID) used to uniquely identify each tuple in the relation
Relationships (1:1 or 1:N)	**Foreign Key**	A link is created by placing the primary key of one relation into another as a foreign key
Relationships (M:N)	**Junction Relation**	A new relation is created to link two relations that participate in a many-to-many relationship, holding the primary keys of both as a composite key
EER Specialization Hierarchy	**One of several mapping strategies**	Superclass/subclass structures are mapped using strategies such as single-table, multiple-table, or joined-table approaches, depending on constraints and query requirements

7.5.4 Benefits of Systematic Mapping

The primary benefit of a formal mapping process is the preservation of design integrity. By following a set of rigorous rules, designers can be confident that the final database implementation faithfully represents the original business requirements. The mapping process eliminates the guesswork that often leads to errors in relation structure or relationship definitions. Furthermore, the mapping process provides clear documentation. A well-defined mapping document explains exactly how each relation and attribute relates

back to the original ER/EER conceptual model, which is invaluable for future audits or system overviews.

Practically, this process enables database implementation by providing a direct blueprint for SQL DDL (Data Definition Language) statements (covered in Chapter 9). Once the mapping is complete, generating the SQL DDL statements to create the database is straightforward. Lastly, this structured approach supports maintenance and evolution. As business needs change, having a clear map between the conceptual and logical levels allows developers to identify exactly which relations or keys need modification without breaking the rest of the system. It ensures that the database is not designed only for current requirements but is robust enough to scale and be updated as the organization grows.

7.5.5 ER-to-Relational Mapping Algorithm Overview

Database relations underpin relational DBMS, providing structured links between data entities via rules rooted in set theory and predicate logic. Essentially, a **relation** is a table with rows (tuples) and columns (attributes), ensuring *data consistency*, eliminating *ambiguity*, and supporting *complex queries* for reliable operations. Key properties uphold integrity: unique identifiers (keys) distinguish tuples, while entity integrity mandates non-null, unique primary keys to preserve record identity. *Referential integrity* maintains logical connections between related relations, preventing orphaned records and ensuring data dependencies remain valid. *Foreign keys* establish inter-relation relationships, creating a cohesive network in which data remains consistent and meaningful across the entire database. The following seven-step algorithm provides a systematic process for converting ER diagrams into relational schemas (Elmasri & Navathe, 2016), as shown in Figure 7.2.

Figure 7.2: Mapping Algorithm Overview

Step 1: Mapping of Regular Strong Entities into Relations

For each **regular (strong) entity type** in the ER schema, create a separate relation, including all its simple attributes.

When the key of an entity is composite, select the set of simple attributes that comprise it as the primary key for the relation. The *relations* created by mapping entity types are known as **entity relations**, since each tuple represents an entity instance. When multiple keys are identified for an entity during conceptual design, one is selected as the primary key, while the remaining candidate keys are enforced as unique constraints.

Step 2: Mapping of Weak Entities into Relations

Weak entities depend on other entities for their existence and identification. Create a relation for the weak entity that includes its own attributes, along with the primary key of its owner entity as a foreign key. The primary key of this new relation combines the owner's key with the weak entity's partial key.

Step 3: Mapping Binary 1:1 Relationships to Foreign Keys

For each binary one-to-one (1:1) relationship type in the ER model, first identify both relations that correspond to the participating entity types. When two entities have a one-to-one relationship, you have three options: (1) the foreign key approach, (2) the merged relation approach, and (3) the cross-reference (relationship) relation approach.

Foreign Key Approach (Preferred): Add the primary key from one relation as a foreign key in the other relation. Choose the relation representing the entity with total participation (where every instance must participate in the relationship). For example, if every PROFESSOR heads exactly one FACULTY, add Heads_Faculty_ID as a foreign key in the PROFESSOR relation.

Merged Relation Approach: Combine the two entities into a single relation when both sides participate fully. This works when the relationship is mandatory for both entities.

Separate Relationship Relation: Create a third relation containing the primary keys of both entities. This approach requires extra joins and is generally avoided for a 1:1 relationship unless there is a specific reason.

Step 4: Mapping Binary 1:N Relationships to Foreign Keys

First, determine the entity type on the “Many” (N) side of the relationship, and store the primary key of the “One” (1) side as a foreign key in that relation. Two main strategies can be used: (1) the foreign key approach and (2) the cross-reference (relationship) relation approach.

Foreign Key Approach (Preferred): If one FACULTY has many **students**, add **Faculty_ID** as a foreign key in the STUDENT relation. This way, each **student's** record points to their faculty, while multiple students can reference a faculty.

Separate Relationship Relation: Create a standalone relation containing keys from both sides. Use this approach only when a few instances on the "many" side actually participate in the relationship, avoiding numerous NULL values.

Step 5: Mapping Binary M:N Relationships to Junction Relations

Many-to-many relationships require a junction relation. Create a new relation whose primary key combines the primary keys from both participating entities.

For example, if a student can enroll in many courses and each course has many students, create an **ENROLLS_IN** relation with Student_ID and Course_ID as a composite primary key. Add any relationship attributes (such as Enrollment_date or Grade) as additional columns in this junction relation.

Step 6: Mapping Multi-valued Attributes to Separate Relations

Attributes that can have multiple values must be stored in a separate relation to maintain First Normal Form (covered in Chapter 10). Create a new relation containing the multi-valued attribute and the primary key from the parent entity. For instance, if a STUDENT can have multiple emails, create an **EMAIL** relation with Student_ID (foreign key) and Email. The primary key of this relation combines both attributes, allowing multiple emails per student while preventing duplicate entries.

Step 7: Mapping N-ary Relationships to New Relations

When a relationship involves three or more entities, create a separate relationship table that includes the primary keys from all participating entities as foreign keys. The primary key of this new table typically combines all these foreign keys.

For example, a PROFESSOR assigns a TEXTBOOK to a COURSE**,** which would require an ASSIGNMENT relation with the fields Professor_ID, Course_ID, and ISBN, forming a composite primary key.

EER-specific constructs, such as specialization hierarchies, require additional mapping steps, as covered in Section 7.5.6.

7.5.6 EER-to-Relational Mapping Algorithm Overview

Having established the seven-step ER mapping algorithm in Section 7.5.5, we now extend our approach to handle EER constructs introduced in Chapter 5. The primary EER construct requiring additional mapping consideration is the specialization hierarchy, in which a superclass is specialized into one or more subclasses, each inheriting the superclass attributes and relationships while potentially adding its own. The mapping strategy chosen for a specialization hierarchy determines how inheritance, constraints, and subclass-specific attributes are represented in the relational schema. Three main strategies are available, each with different trade-offs (Elmasri & Navathe, 2016). The complete step-by-step application of these strategies to both case studies is demonstrated in Section 7.7.

Strategy A: Single-Relation (Flat) Approach

Create one relation for the entire hierarchy — superclass and all subclasses — including all attributes from both the superclass and every subclass. Add a discriminator attribute (also called a **type attribute**) to indicate which subclass each tuple belongs to. Subclass-specific attributes that do not apply to a given tuple will contain NULL values.

***University System Example*:**

The STUDENT specialization from Chapter 5, where STUDENT is specialized into {BACHELOR_STUDENT, MASTERS_STUDENT, DOCTORAL_STUDENT} with a (d, p) constraint, would produce a single relation:

STUDENT (Student_ID, SSN, First_name, Last_name, Birth_date, Street, City, Zip_code, Country, Email, Enrollment_date, ECTS_completed, Student_type, Major_field, Study_year, Defense_committee, Master_thesis_topic, Total_ECTS_required, Dissertation_topic, Expected_defense_date)

Where Student_type is the discriminator attribute with values such as 'Bachelor', 'Masters', or 'Doctoral'. A tuple representing a BACHELOR_STUDENT would have NULL values for Master_thesis_topic, Dissertation_topic, and Expected_defense_date. A tuple representing a DOCTORAL_STUDENT would have NULL values for Major_field, Study_year, and Master_thesis_topic.

🏢 ***Company System Example*:**

The EMPLOYEE specialization, where EMPLOYEE is specialized into {FULL_TIME_EMPLOYEE, PART_TIME_EMPLOYEE, CONTRACT_EMPLOYEE} with a (d, p) constraint, defined by the discriminator 'Employment_type', would produce:

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Phone, Street, City, Zip_code, Country, Employment_type, Annual_salary_amount, Benefits_package_level, Vacation_days_per_year, Retirement_contribution_percentage, Hourly_rate, Maximum_hours_per_week, Schedule_flexibility, Payment_rate, Contract_start_date, Contract_end_date, Contracting_agency_or_independent)

Where Employment_type is the discriminator attribute with values 'Full-time', 'Part-time', or 'Contract'.

When to use: Best suited for disjoint, total specializations (d, t) where subclasses have few specific attributes. Simple to implement and query, but produces many NULL values for subclass-specific attributes that do not apply to all tuples.

Strategy B: Multiple Relations with Superclass

Create one relation for the superclass containing its own attributes, and one separate relation for each subclass containing the subclass-specific attributes plus the superclass primary key as a foreign key.

🎓 ***University System Example*:**

STUDENT (Student_ID, SSN, First_name, Last_name, Birth_date, Street, City, Zip_code, Country, Email, Enrollment_date, ECTS_completed)

BACHELOR_STUDENT (Student_ID*, Major_field, Study_year, Defense_committee, Total_ECTS_required)

MASTERS_STUDENT (Student_ID*, Master_thesis_topic, Total_ECTS_required, Defense_committee)

DOCTORAL_STUDENT (Student_ID*, Dissertation_topic, Defense_committee, Expected_defense_date)

***Company System Example*:**

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Phone, Street, City, Zip_code, Country)

FULL_TIME_EMPLOYEE (Employee_ID*, Annual_salary_amount, Benefits_package_level, Vacation_days_per_year, Retirement_contribution_percentage)

PART_TIME_EMPLOYEE (Employee_ID*, Hourly_rate, Maximum_hours_per_week, Schedule_flexibility)

CONTRACT_EMPLOYEE (Employee_ID*, Payment_rate, Contract_start_date, Contract_end_date, Contracting_agency_or_independent)

When to use: Best suited for disjoint or overlapping specializations where subclasses have many specific attributes. Preserves superclass data independently and avoids NULL values, but requires joins to retrieve complete entity information.

Strategy C: Multiple Relations without Superclass

Create one relation for each subclass, where each subclass relation includes both the superclass attributes and its own specific attributes. No separate relation is created for the superclass.

***University System Example*:**

BACHELOR_STUDENT (Student_ID, SSN, First_name, Last_name, Birth_date, Street, City, Zip_code, Country, Email, Enrollment_date, ECTS_completed, Major_field, Study_year, Defense_committee, Total_ECTS_required)

MASTERS_STUDENT (Student_ID, SSN, First_name, Last_name, Birth_date, Street, City, Zip_code, Country, Email, Enrollment_date, ECTS_completed, Master_thesis_topic, Total_ECTS_required, Defense_committee)

DOCTORAL_STUDENT (Student_ID, SSN, First_name, Last_name, Birth_date, Street, City, Zip_code, Country, Email, Enrollment_date, ECTS_completed, Dissertation_topic, Defense_committee, Expected_defense_date)

***Company System Example*:**

FULL_TIME_EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Phone, Street, City, Zip_code, Country, Annual_salary_amount, Benefits_package_level, Vacation_days_per_year, Retirement_contribution_percentage)

PART_TIME_EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Phone, Street, City, Zip_code, Country, Hourly_rate, Maximum_hours_per_week, Schedule_flexibility)

CONTRACT_EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Phone, Street, City, Zip_code, Country, Payment_rate, Contract_start_date, Contract_end_date, Contracting_agency_or_independent)

When to use: Only applicable for disjoint, total specializations (d, t) where every superclass entity must belong to exactly one subclass. Eliminates the need for joins but

makes it difficult to query all entities across subclasses simultaneously, since no single superclass relation exists.

Choosing a Strategy

The choice of mapping strategy is guided by the specialization constraints established in the EER diagram, as summarized in Table 7.12 (Elmasri & Navathe, 2016; Hristoski & Dimovski, 2022):

Table 7.12: Choosing a Strategy

Constraint	Recommended Strategy
Disjoint, Total (d, t)	*Strategy A or C — every entity belongs to exactly one subclass*
Disjoint, Partial (d, p)	*Strategy A or B — some entities may not belong to any subclass*
Overlapping, Total (o, t)	*Strategy B — entities may belong to multiple subclasses*
Overlapping, Partial (o, p)	*Strategy B — most flexible; handles all overlap and partial membership cases*

In practice, Strategy B is the most widely used because it cleanly separates superclass and subclass data, supports all constraint types, and scales well as subclasses are added or modified. Strategy A is preferred when subclasses have very few specific attributes and query simplicity is a priority. Strategy C is rarely used in practice due to the difficulty of querying across all entity instances.

Table 7.12 should be read as a guide to the minimum viable strategies for each constraint combination, not as an absolute rule. In particular, Strategy B is also a sound choice for disjoint, total (d, t) specializations when the superclass entity is heavily queried independently of its subclasses — for example, when the system frequently needs all students or all employees regardless of their subclass membership. In such cases, the superclass relation that Strategy B preserves offers a significant practical advantage over Strategy A and Strategy C. The worked examples in Section 7.7 apply this reasoning consistently.

7.6 Creating the Relational Model Schema

In this section, we will demonstrate the process of creating a **relational model schema** using real-world examples following the *Logical Design Process*, described in Section 6.4. This practical approach consolidates all the concepts explained in Section 7.5, providing a systematic methodology for relational database schema design. We will use two comprehensive case studies to illustrate this process for the University Management System and the Company/Business Database.

The process of creating a relational model schema entails translating real-world data into a structured format using relations, following the seven-step ER-to-Relational mapping algorithm described in Section 7.5.5. The process begins with the identification of distinct entities, which are subsequently represented as relations in the schema. Each relation

represents a specific category of data. Within each relation, the columns, called attributes, represent the various characteristics and properties of the corresponding data. The rows, or tuples, represent individual records, with each row corresponding to a unique instance of the entity. A key step in this modeling process involves defining keys to maintain data integrity and ensure a coherent relational structure. Primary keys uniquely identify each tuple in a relation, preventing duplicate records. In contrast, foreign keys are instrumental for linking relations, thereby establishing relationships that are essential to maintaining referential integrity across the database.

University Management System Example

Creating a relational model schema begins with defining relations for each entity and its attributes, establishing primary keys to uniquely identify tuples, and using foreign keys to represent relationships between entities. Based on the complete University Database ER diagram, which integrates all components identified during requirements analysis, shown in Figure 7.3, the University System maps to the following **relational model** schema.

7.6.1 University System Entities

We begin by applying Steps 1 and 2 of the mapping algorithm to the University Database ER Diagram, shown in Figure 7.3. These steps identify the entities and convert them into relations. Strong entities are mapped directly into relations, with each simple attribute becoming a column and the key attribute becoming the primary key. Weak entities require special handling, incorporating the primary key of their owner entity as part of a composite primary key. The University Database ER Diagram contains five strong entities — STUDENT, COURSE, PROFESSOR, FACULTY, and TEXTBOOK — and one weak entity — CLASS_SESSION, which depends on COURSE.

Example of STEP 1: Mapping Strong Entities

In this step, we demonstrate Step 1 using the STUDENT entity from the University Database ER Diagram, as shown in Figure 7.4. Based on step 1 of the ER-to-Relational mapping algorithm, every regular entity in the ER schema will be a separate relation, including all its simple attributes.

The relational schema for the STUDENT entity in the given University Database ER Diagram includes the relation STUDENT and the attributes Student_ID, First_name, Last_name, Birth_date, Street, City, Country, and Zip_code. The Student_ID is underlined, representing the primary key in the relation.

STUDENT (<u>Student_ID</u>, First_name, Last_name, Birth_date, Street, City, Country, Zip_code)

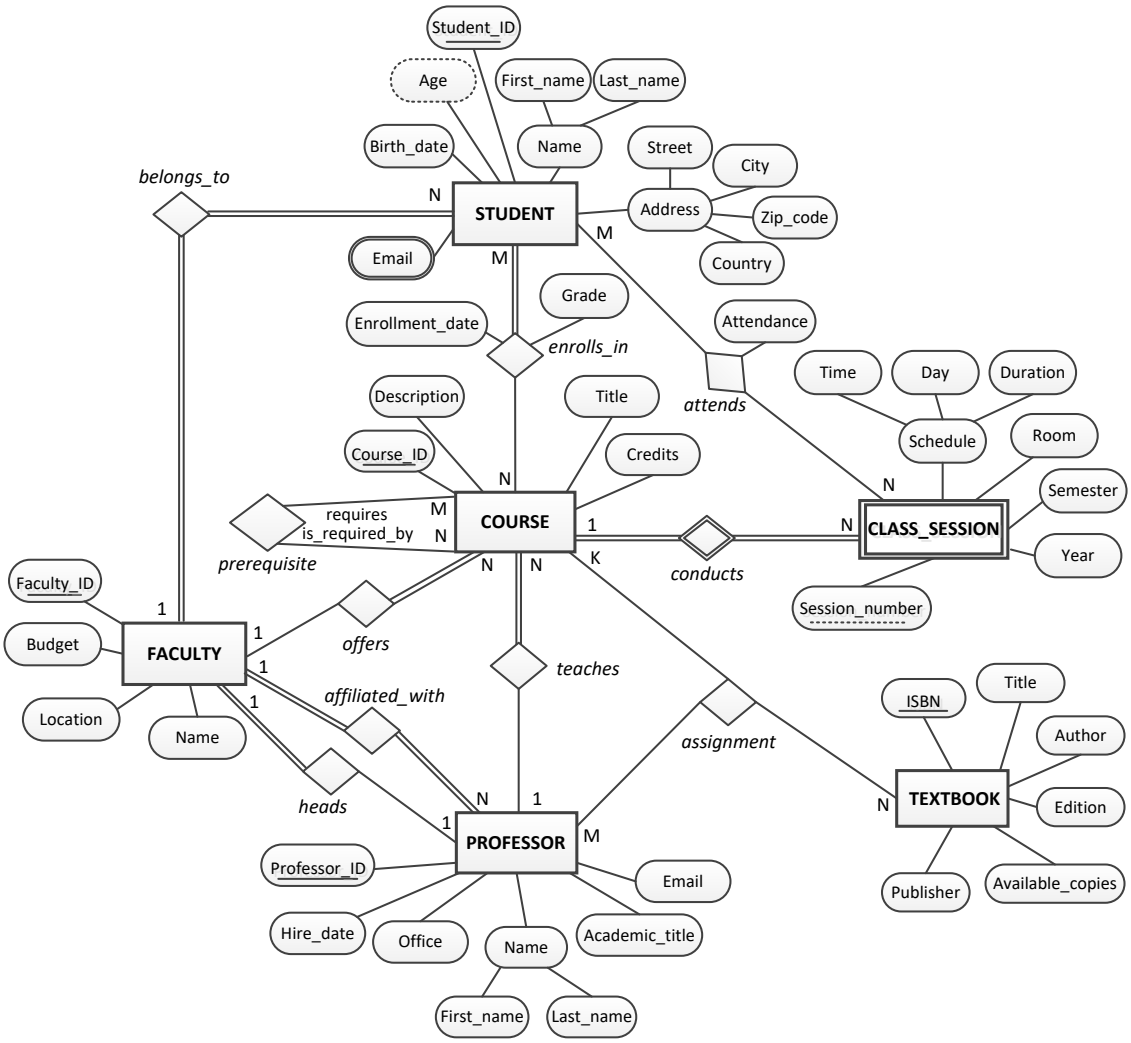


Figure 7.3: Complete University Database ER Diagram

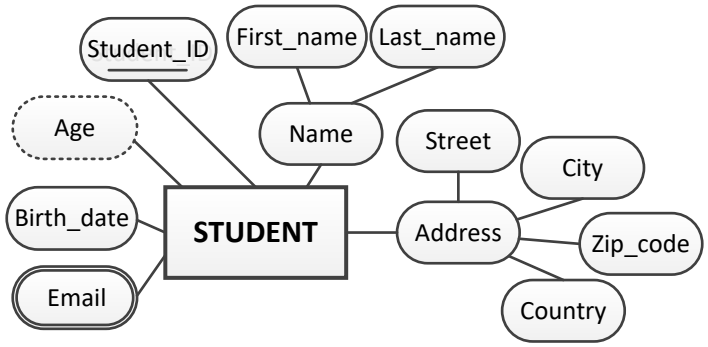


Figure 7.4: STUDENT Entity with Complete Attribute Structure

Key Points:

- Composite attributes flattened: Name → First_name, Last_name
- Derived attributes are not stored in the relation: Age is calculated from Birth_date when needed
- **Note**: The multivalued attribute Email is not included here; it will be handled as a separate relation in Step 6

- **Note**: The foreign key Faculty_ID, derived from the belongs_to relationship between STUDENT and FACULTY, is not visible in Figure 7.4 — it will be added to the STUDENT relation in Step 4 (Example 2)

Example of STEP 2: Mapping Weak Entities

In this step, we demonstrate Step 2 using the weak entity CLASS_SESSION from the given University Database ER Diagram, as shown in Figure 7.5. Based on step 2 of the ER-to-Relational mapping algorithm, a relation is created for each weak entity, including its attributes and the primary key of its identifying owner as a foreign key. In this case, the weak entity CLASS_SESSION depends on the entity COURSE via the foreign key Course_ID. The weak entity CLASS_SESSION has a composite primary key consisting of the foreign key Course_ID, the partial key Session_number, and the attribute Year.

The **relational schema** for the weak entity CLASS_SESSION from the given University Database ER Diagram includes the relation CLASS_SESSION, with foreign key Course_ID, partial key Session_number, and attributes Room, Semester, Day, Time, and Duration.

CLASS_SESSION (Course_ID*, Session_number, Year, Room, Semester, Time, Day, Duration)

Key Points:

- Composite PK: (Course_ID, Session_number, Year) – the combination of Course_ID and Session_number alone is not unique across different years, so Year is added to the primary key
- Course_ID is both part of PK and FK to COURSE
- Total participation enforced through FK constraint

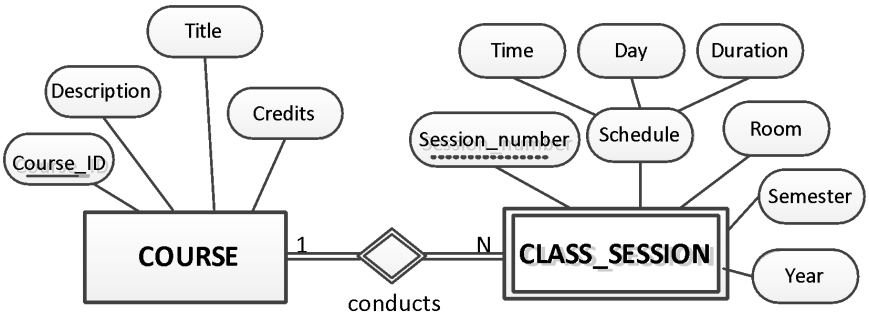


Figure 7.5: ER Diagram of weak entity CLASS_SESSION

7.6.2 University System Relationships and Keys

Relationships between entities are defined using primary keys and foreign keys. A primary key (e.g., Faculty_ID in the FACULTY relation) uniquely identifies each tuple. A foreign key is an attribute in one relation that references the primary key of another, establishing a link.

Example of STEP 3: Mapping 1:1 Relationships

In this step, we demonstrate Step 3 using the 1:1 relationship *heads* between FACULTY and PROFESSOR, from the University Database ER Diagram, as shown in Figure 7.6. A

professor can head at most one faculty, and each faculty has at most one head professor. The PROFESSOR relation has a foreign key Heads_Faculty_ID that references the FACULTY relation, indicating which faculty the professor heads.

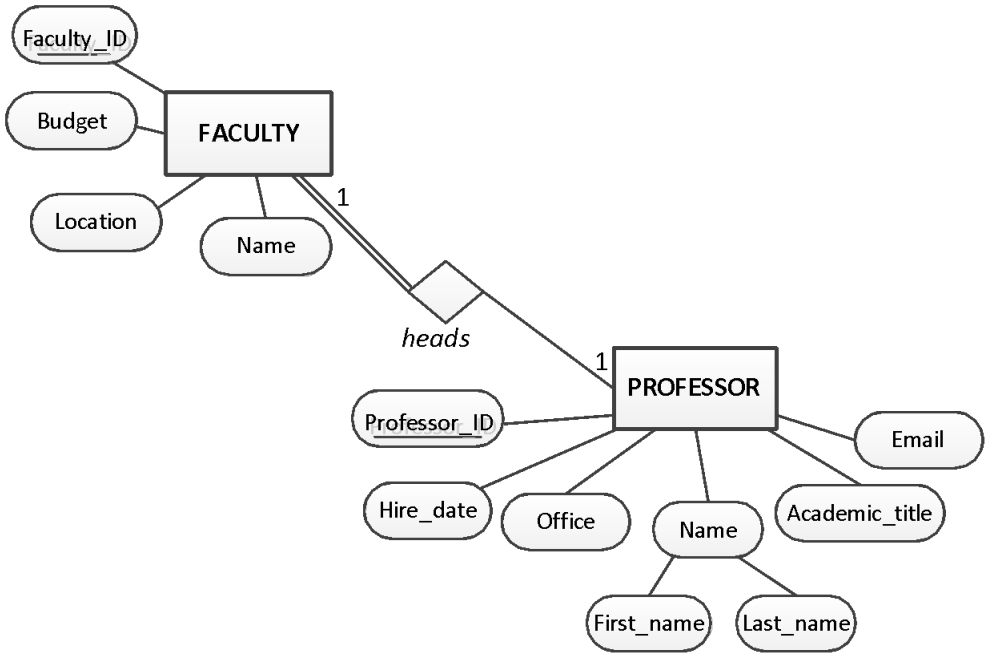


Figure 7.6: ER Diagram of relationship FACULTY heads PROFESSOR

The relational schema for the 1:1 relationship *heads* between FACULTY and PROFESSOR has two options:

Option 1 — Foreign key in the partial participation side: Add Heads_Faculty_ID as a foreign key in the PROFESSOR relation, referencing FACULTY.Faculty_ID. PROFESSOR has partial participation (not all professors head a faculty). In this case, the relational schema (FK in partial side) is:

FACULTY (Faculty_ID, Budget, Location, Name)

PROFESSOR (Professor_ID, First_name, Last_name, Hire_date, Office, Academic_title, Email, Heads_Faculty_ID*)

Option 2 — Foreign key in the total participation side: Add Head_Professor_ID as a foreign key in the FACULTY relation, referencing PROFESSOR.Professor_ID. In this case, the relational schema (FK in total side) is:

FACULTY (Faculty_ID, Budget, Location, Name, Head_Professor_ID*)

PROFESSOR (Professor_ID, First_name, Last_name, Hire_date, Office, Academic_title, Email)

Decision factors:

- Placing the FK on the total participation side → fewer NULL values
- Placing the FK on the partial participation side → more flexible for future changes

Example 1 of STEP 4: Mapping 1:N Relationships

In this step, we demonstrate the mapping of the 1:N relationship *teaches* between PROFESSOR and COURSE, from the University Database ER Diagram, as shown in Figure 7.7. In the University system, each course is taught by exactly one professor. In general, every professor teaches one or more courses, forcing total participation on the COURSE side.

The relational schema for the 1:N relationship *teaches* includes the relation COURSE with a foreign key Professor_ID. Each course is associated with exactly one professor from the PROFESSOR relation.

PROFESSOR (Professor_ID, First_name, Last_name, Hire_date, Office, Academic_title, Email)

COURSE (Course_ID, Title, Description, Credits, Professor_ID*)

Note: Professor_ID in COURSE is NOT NULL because of the total participation!

Note: Faculty_ID* will be added to both PROFESSOR and COURSE when the *affiliated_with* and *offers* relationships are processed. Heads_Faculty_ID* will be added to PROFESSOR from Step 3. The complete PROFESSOR and COURSE schemas are shown in the final relational model in Section 7.6.5.

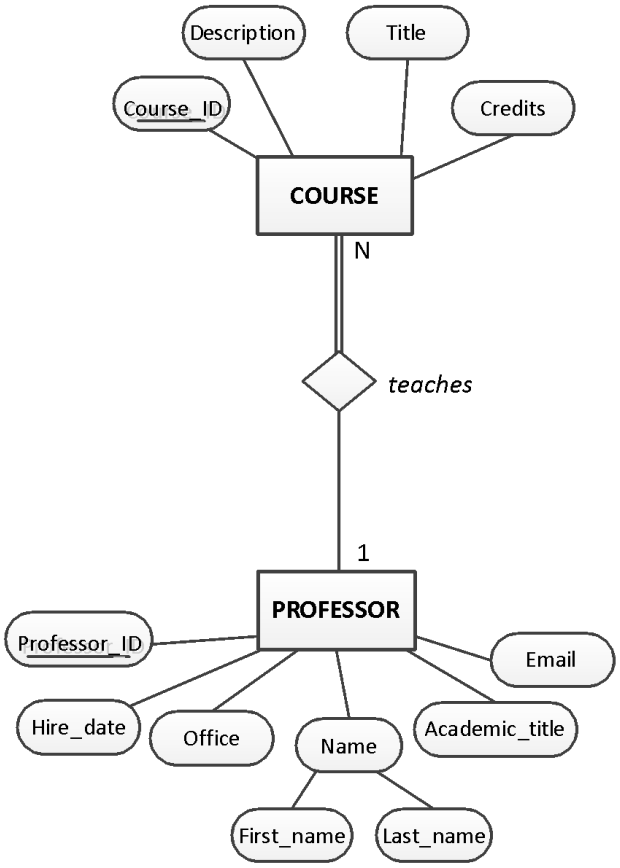


Figure 7.7: University (1:N) relationship PROFESSOR *teaches* COURSE

Example 2 of STEP 4: Mapping 1:N Relationships

In this step, we demonstrate the mapping of the 1:N relationship *belongs_to* between FACULTY and STUDENT, from the University Database ER Diagram, as shown in Figure 7.8. Each STUDENT belongs to exactly one FACULTY, and every FACULTY must have at least one STUDENT, forcing total participation on both sides.

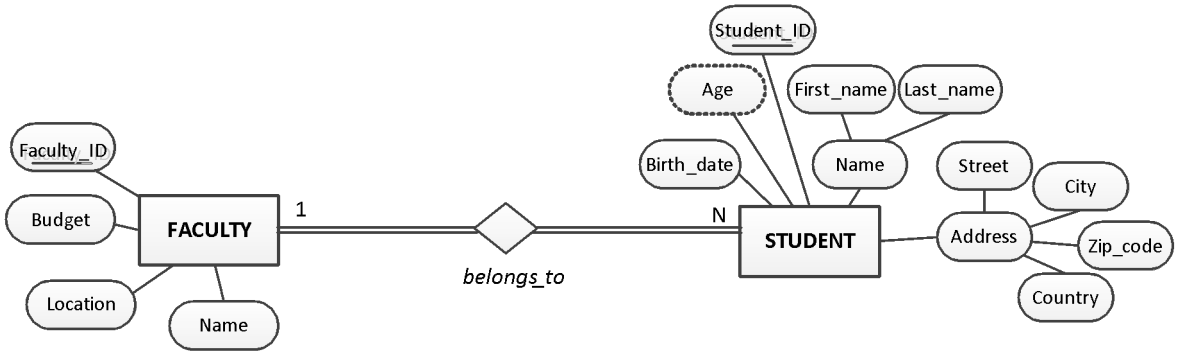


Figure 7.8: University (1:N) relationship STUDENT belongs_to FACULTY

The relational schema for the 1:N relationship *belongs_to* includes the relation FACULTY and the relation STUDENT with a foreign key Faculty_ID, identifying the faculty to which the student belongs.

FACULTY (<u>Faculty_ID</u>, Budget, Location, Name)

STUDENT (<u>Student_ID</u>, First_name, Last_name, Birth_date, Street, City, Zip_code, Country, Faculty_ID*)

Note: Faculty_ID is NOT NULL because of total participation on both sides.

Note: The complete STUDENT schema, including all foreign keys from other relationships, is shown in the final relational model in Section 7.6.5.

The complete list of 1:N (one-to-many) relationships in the University Management System:

- *belongs_to* (STUDENT–FACULTY): Each student belongs to exactly one faculty. Faculty_ID* is a foreign key in STUDENT referencing the primary key Faculty_ID in the FACULTY relation (FACULTY.Faculty_ID).
- *affiliated_with* (PROFESSOR–FACULTY): Each professor is affiliated with exactly one faculty. Faculty_ID* is a foreign key in PROFESSOR referencing FACULTY.Faculty_ID.
- *offers* (FACULTY–COURSE): Each course is offered by exactly one faculty. Faculty_ID* is a foreign key in COURSE referencing FACULTY.Faculty_ID.
- *teaches* (PROFESSOR–COURSE): Each course is taught by exactly one professor. Professor_ID* is a foreign key in COURSE referencing PROFESSOR.Professor_ID.
- *conducts* (COURSE–CLASS_SESSION): Each class session belongs to exactly one course. Course_ID is a foreign key in CLASS_SESSION referencing COURSE.Course_ID. Note: since CLASS_SESSION is a weak entity, this relationship

was already handled in Step 2, where Course_ID was incorporated as part of the composite primary key of CLASS_SESSION.

Example 1 of Step 5: Mapping Binary M:N Relationships to Junction Relations

In this step, **we demonstrate** the mapping of the M:N relationship *enrolls_in* between STUDENT and COURSE, from the University Database ER Diagram, as shown in Figure 7.9. In this case, the many-to-many M:N relationship creates a new relation ENROLLS_IN, whose primary key is formed by combining the primary keys of the two participating entities STUDENT and COURSE.

The relational schema for mapping the M:N relationship includes the new relation ENROLLS_IN, with foreign keys Student_ID* from the relation STUDENT and Course_ID* from the relation COURSE. The relationship attributes are Enrollment_date and Grade.

STUDENT (Student_ID, First_name, Last_name, Birth_date, Street, City, Zip_code, Country)

COURSE (Course_ID, Title, Credits, Description)

ENROLLS_IN (Student_ID*, Course_ID*, Enrollment_date, Grade)

Note: The complete STUDENT and COURSE schemas, including all foreign keys from other relationships, are shown in the final relational model in Section 7.6.5.

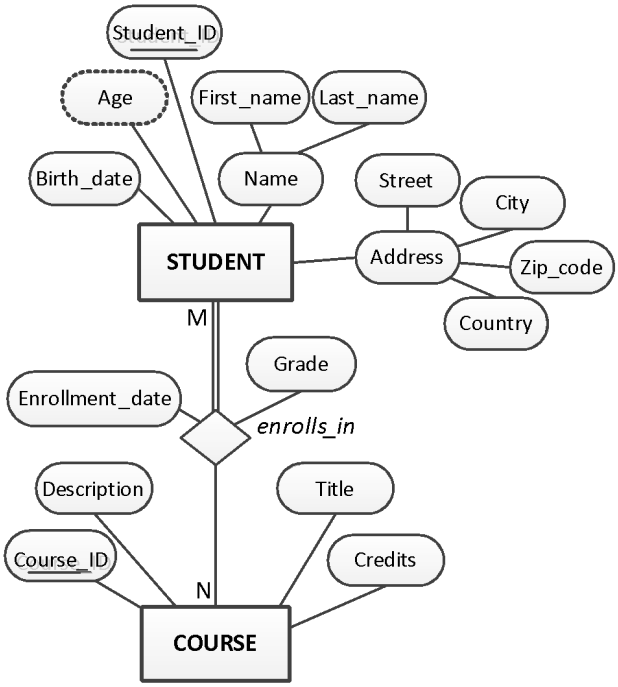


Figure 7.9: University M:N relationship STUDENT *enrolls_in* COURSE

Example 2 of Step 5: Mapping Binary M:N Relationships to Junction Relations

In this step, we demonstrate the mapping of the recursive M:N relationship *prerequisite* on COURSE, from the University Database ER Diagram, as shown in Figure 7.10. In this case, the many-to-many M:N relationship creates a new relation PREREQUISITE.

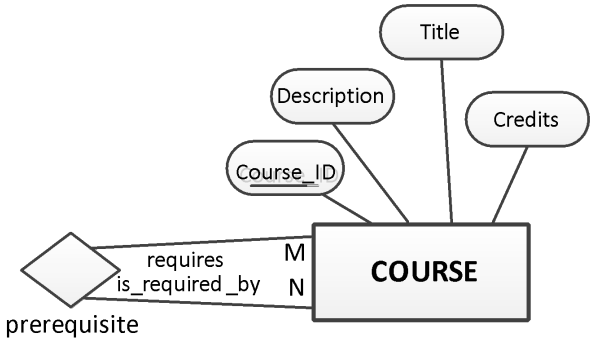


Figure 7.10: University recursive M:N relationship *COURSE prerequisite COURSE*

The relational schema for mapping the M:N relationship includes the new relation PREREQUISITE, identifying the prerequisite courses for each course. The foreign key Course_ID* references the relation COURSE, while Prerequisite_Course_ID* references the same COURSE relation, representing the prerequisite course.

COURSE (Course_ID, Title, Credits, Description)

PREREQUISITE (Course_ID*, Prerequisite_Course_ID*)

Note: The complete COURSE schema, including all foreign keys from other relationships, is shown in the final relational model in Section 7.6.5.

Example 3 of Step 5: Mapping Binary M:N Relationships to Junction Relations

In this step, we demonstrate the mapping of the M:N relationship *attends* between STUDENT and CLASS_SESSION, from the University Database ER Diagram, as shown in Figure 7.11. In this case, the many-to-many M:N relationship creates a new relation ATTENDS, which tracks each student's attendance at a class session.

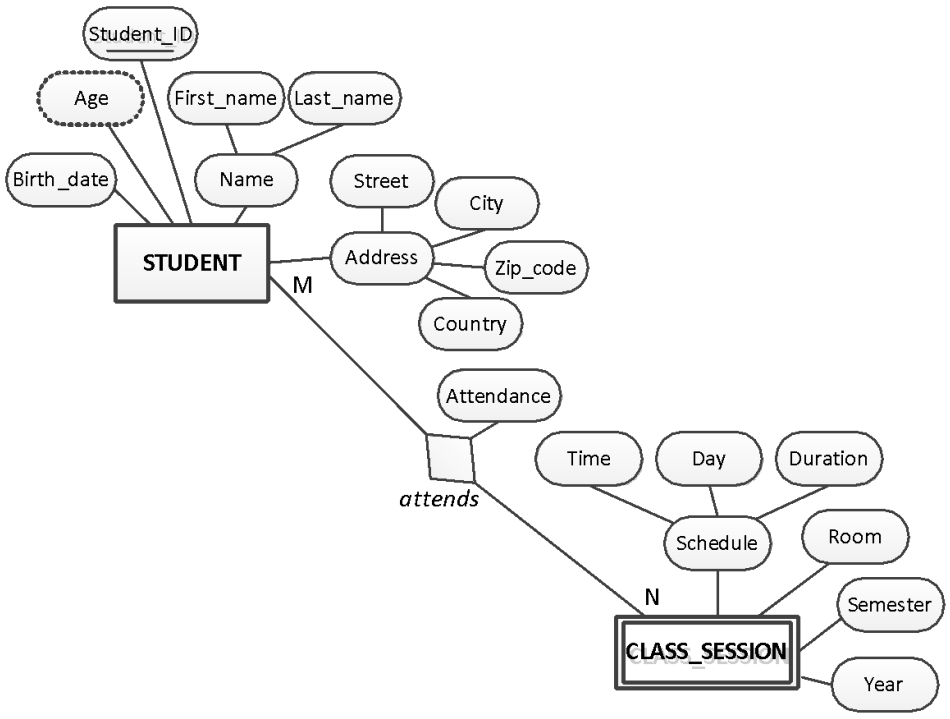


Figure 7.11: University M:N relationship *STUDENT attends CLASS_SESSION*

The relational schema for mapping the M:N relationship includes the new relation ATTENDS. This relation has foreign keys Student_ID* from the relation STUDENT, and *Course_ID, Session_number*, and *Year* from the relation CLASS_SESSION. The relation ATTENDS has the attribute Attendance, which marks the student's attendance at class sessions.

STUDENT (Student_ID, First_name, Last_name, Birth_date, Street, City, Zip_code, Country)

CLASS_SESSION (Course_ID*, Session_number, Year, Room, Semester, Time, Day, Duration)

ATTENDS (Student_ID*, Course_ID*, Session_number*, Year*, Attendance)

Note: The complete STUDENT schema, including all foreign keys from other relationships, is shown in the final relational model in Section 7.6.5.

The complete list of M:N (many-to-many) relationships in the University Management System:

- *enrolls_in* (STUDENT–COURSE): A student can enroll in multiple courses, and a course can have multiple students. This relationship is modeled by the relation ENROLLS_IN, which includes foreign keys from both STUDENT (Student_ID*) and COURSE (Course_ID*), along with relationship attributes such as Enrollment_date and Grade.
- *prerequisite* (COURSE–COURSE): A course can have multiple prerequisites, and a course can be a prerequisite for multiple other courses. The PREREQUISITE relation is modeled with Course_ID* and Prerequisite_Course_ID* as foreign keys, both referencing the COURSE relation.
- *attends* (STUDENT–CLASS_SESSION): A student can attend multiple class sessions, and a class session can have multiple students. This is modeled using the ATTENDS relation, which includes foreign keys for Student_ID* and the composite key CLASS_SESSION (Course_ID*, Session_number*, Year*).

7.6.3 University System Multivalued Attributes and N-ary Relationships

Example of Step 6: Mapping Multivalued Attributes to Separate Relations

In this step, we demonstrate the mapping of the multivalued attribute Email from the STUDENT relation, from the University Database ER Diagram, as shown in Figure 7.12. To satisfy First Normal Form (1NF) (covered in Chapter 10), a multivalued attribute must be placed in a separate relation. In this case, the EMAIL relation is created to store multiple email addresses per student.

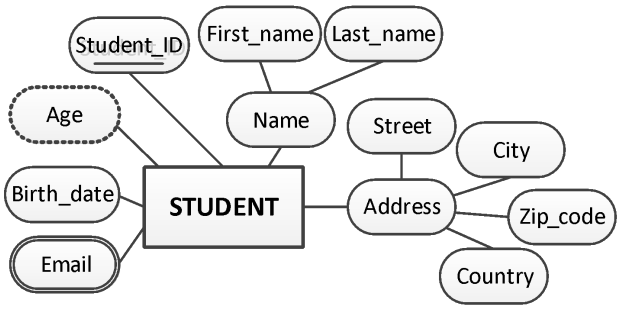


Figure 7.12: STUDENT Entity with Complete Attribute Structure

The relational schema for mapping the multivalued attribute Email includes the new relation EMAIL, with foreign key Student_ID* referencing the STUDENT relation, and the attribute Email. The new relation EMAIL stores separate email addresses (such as personal and university email) for each student, identified by the combination of Student_ID and Email.

STUDENT (Student_ID, First_name, Last_name, Birth_date, Street, City, Zip_code, Country)

EMAIL (Student_ID*, Email)

Note: The complete STUDENT schema, including all foreign keys from other relationships, is shown in the final relational model in Section 7.6.5.

Example of Step 7: Mapping N-ary Relationships

In this step, we demonstrate the mapping of the N-ary relationship assignment between PROFESSOR, COURSE, and TEXTBOOK (M:N:K), from the University Database ER Diagram, as shown in Figure 7.13. This ternary relationship creates a new relation ASSIGNMENT, which tracks which textbook is assigned to which course by which professor.

The relational schema for mapping the N-ary relationship includes the new relation ASSIGNMENT, with the following foreign keys: Course_ID* referencing the relation COURSE, Professor_ID* referencing the relation PROFESSOR, and ISBN* referencing the relation TEXTBOOK.

COURSE (Course_ID, Title, Credits, Description)

TEXTBOOK (ISBN, Title, Author, Edition, Publisher, Available_copies)

PROFESSOR (Professor_ID, First_name, Last_name, Hire_date, Office, Academic_title, Email)

ASSIGNMENT (Professor_ID*, Course_ID*, ISBN*)

Note: The complete COURSE and PROFESSOR schemas, including all foreign keys from other relationships, are shown in the final relational model in Section 7.6.5.

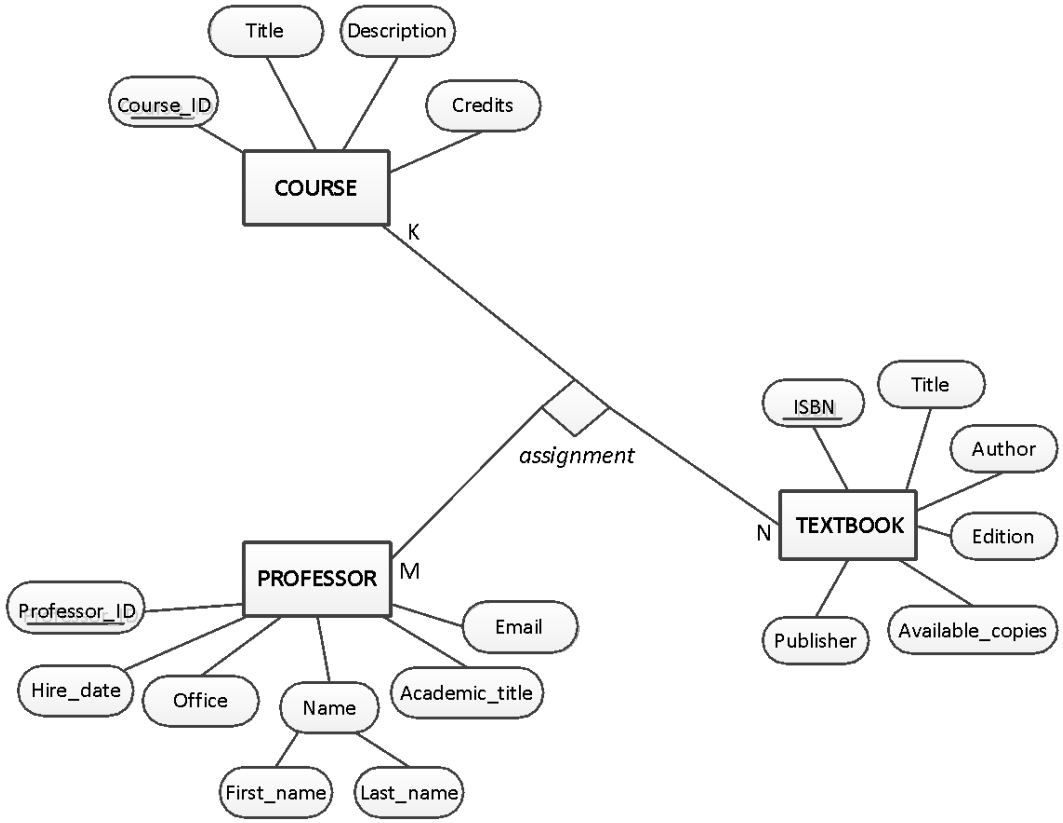


Figure 7.13: ER Diagram of PROFESSOR assigns TEXTBOOK to COURSE (M:N:K)

7.6.4 University System Integrity Constraints

Having completed all seven mapping steps, the resulting relational schema enforces the following essential integrity constraints to ensure data quality and consistency:

- **Entity Integrity:** Underlining the attribute name in a relational schema explicitly indicates that the chosen attribute or attributes serve as the primary key for the relation and cannot have a NULL value, guaranteeing that every tuple is uniquely identifiable.
- **Referential Integrity:** Foreign keys must reference a valid, existing primary key. Foreign keys are marked with an asterisk (*) *to visually distinguish them from other attributes within a relation, ensuring that data in related relations remains consistent and preventing the creation of orphaned records. For example, Student_ID* in the ENROLLS_IN relation must correspond to an actual student in the STUDENT relation.
- **Other Constraints:** The model's structure also enforces specific business rules, such as a course always having an assigned professor (Professor_ID* in COURSE is NOT NULL because of total participation) and a student always belonging to a faculty (Faculty_ID* in STUDENT is NOT NULL because of total participation on both sides).

7.6.5 The Complete Relational Model Schema of the University System

The following complete relational model schema for the University Database System results from applying all seven steps of the ER-to-Relational mapping algorithm to the University Database ER Diagram shown in Figure 7.3. Each relation was derived systematically: entity relations from Steps 1 and 2, foreign keys from Steps 3 and 4, junction relations from Step 5, the EMAIL relation from Step 6, and the ASSIGNMENT relation from Step 7:

FACULTY (Faculty_ID, Name, Budget, Location)

STUDENT (Student_ID, First_name, Last_name, Birth_date, Age, Street, City, Country, Zip_code, Faculty_ID*)

EMAIL (Student_ID*, Email)

PROFESSOR (Professor_ID, First_name, Last_name, Office, Email, Hire_date, Academic_title, Faculty_ID*, Heads_Faculty_ID*)

COURSE (Course_ID, Title, Credits, Description, Faculty_ID*, Professor_ID*)

PREREQUISITE (Course_ID*, Prerequisite_Course_ID*)

ENROLLS_IN (Student_ID*, Course_ID*, Enrollment_date, Grade)

CLASS_SESSION (Course_ID*, Session_number, Year, Room, Semester, Time, Day, Duration)

ATTENDS (Student_ID*, Course_ID*, Session_number*, Year*, Attendance)

TEXTBOOK (ISBN, Title, Author, Edition, Publisher, Available_copies)

ASSIGNMENT (Professor_ID*, Course_ID*, ISBN*)

Note: Age is included as a stored attribute for administrative purposes, though it can also be derived from Birth_date.

Company System Example

To translate the Company system into a relational model schema, we first identify the entities which become relations. Then, we define their attributes, primary keys for unique identification, and foreign keys to establish relationships, thereby ensuring data integrity. Based on the complete Company Database ER diagram, which integrates all components identified during requirements analysis, shown in Figure 7.14, the Company Management System maps to the following relational model schema.

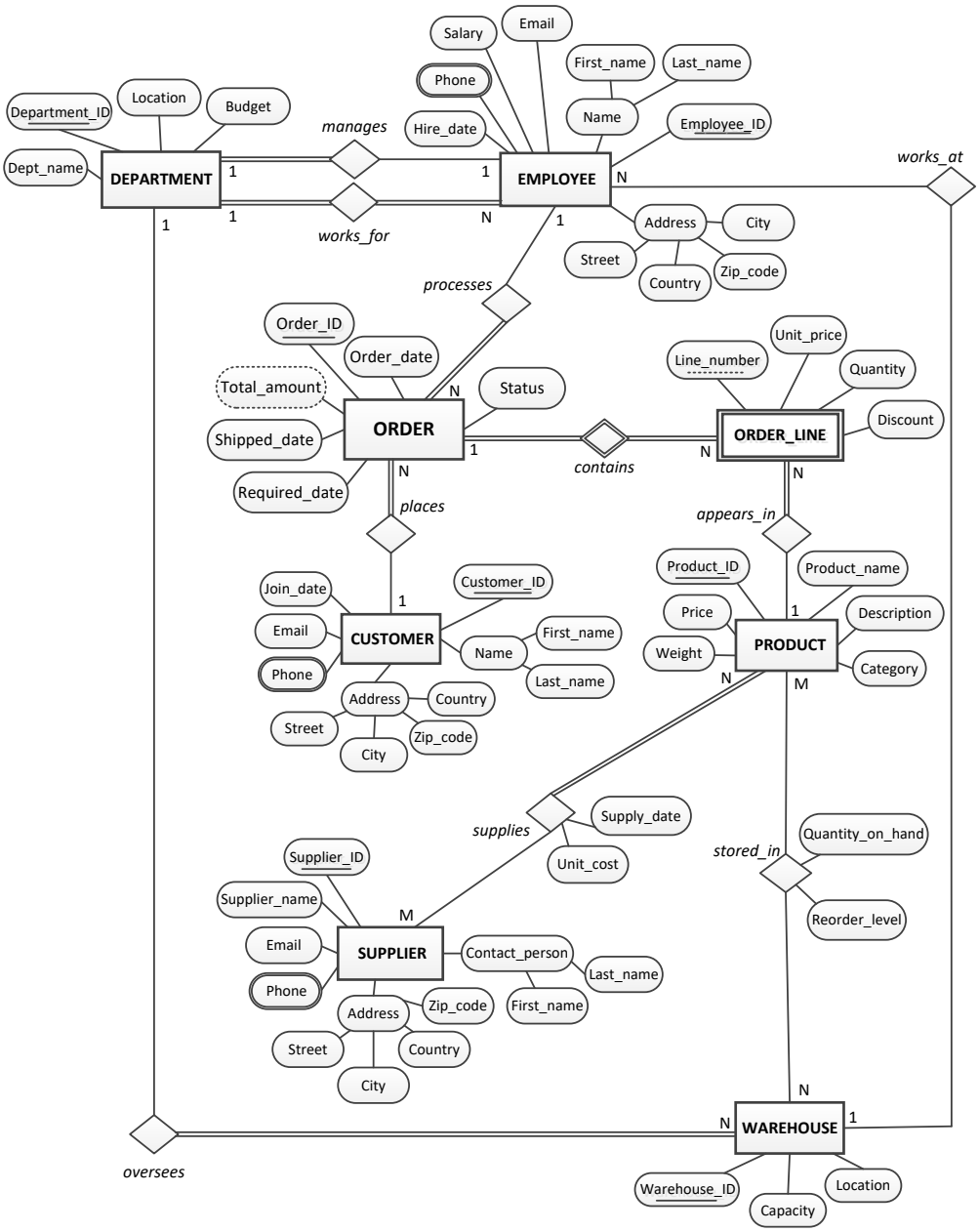


Figure 7.14: Complete Company Database ER Diagram

7.6.6 Company System Entities

We begin by applying Steps 1 and 2 of the mapping algorithm to the Company Database ER Diagram, shown in Figure 7.14. The Company Database ER Diagram contains seven strong entities — DEPARTMENT, WAREHOUSE, EMPLOYEE, ORDER, CUSTOMER, PRODUCT, and SUPPLIER — and one weak entity — ORDER_LINE, which depends on ORDER.

Example of STEP 1: Mapping Strong Entities

In this step, we demonstrate Step 1 using the EMPLOYEE entity from the Company Database ER Diagram, as shown in Figure 7.15.

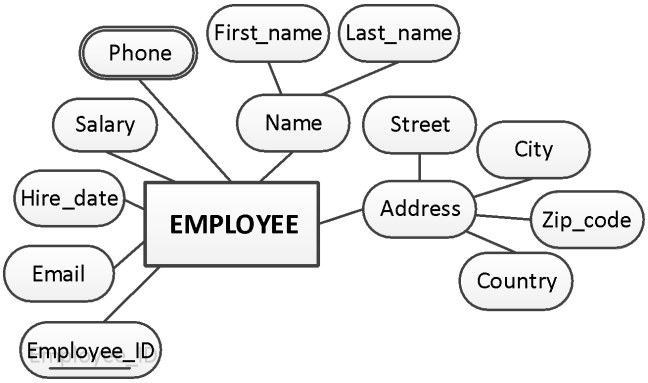


Figure 7.15: EMPLOYEE Entity with Complete Attribute Structure

The relational schema for the EMPLOYEE entity includes the relation EMPLOYEE with attributes Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Country, and Zip_code. Employee_ID is underlined, representing the primary key of the relation.

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Country, Zip_code)

Key Points:

- Composite attributes flattened: Name → First_name, Last_name; Address → Street, City, Country, Zip_code
- **Note:** Phone is a multivalued attribute and will be handled as a separate relation EMPLOYEE_PHONE in Step 6. Similarly, CUSTOMER_PHONE and SUPPLIER_PHONE will be created for CUSTOMER and SUPPLIER, respectively.
- **Note:** Foreign keys Department_ID*, Warehouse_ID*, and Manages_Dept_ID* will be added to the EMPLOYEE relation when the *works_for*, *works_at*, and *manages* relationships are processed in Step 4. The complete EMPLOYEE schema is shown in Section 7.6.9.

Example of STEP 2: Mapping Weak Entities

In this step, we demonstrate Step 2 using the weak entity ORDER_LINE from the Company Database ER Diagram, as shown in Figure 7.16. ORDER_LINE is a weak entity depending on ORDER for its existence. Line_number provides identification within each order, while Unit_price, Quantity, and Discount capture transaction details essential for fulfillment and accounting.

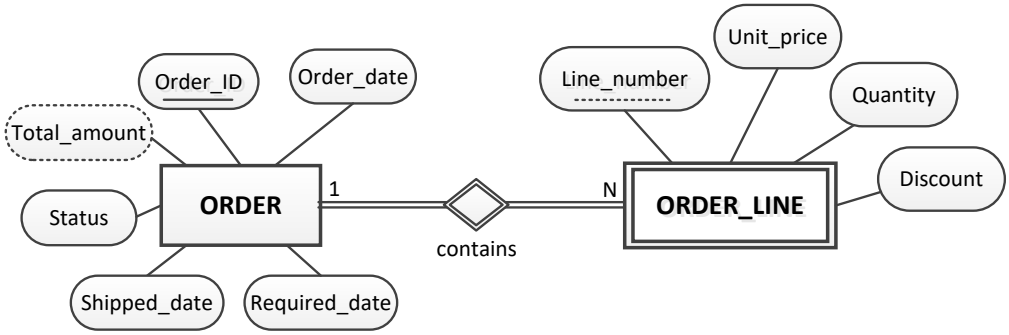


Figure 7.16: Weak entity ORDER_LINE

The relational schema for the weak entity ORDER_LINE includes the relation ORDER_LINE with foreign key Order_ID*, partial key Line_number, and attributes Unit_price, Quantity, and Discount.

ORDER_LINE (Order_ID*, Line_number, Unit_price, Quantity, Discount)

Key Points:

- Composite PK: (Order_ID, Line_number) — Line_number alone is not unique across different orders, so Order_ID is incorporated as part of the composite primary key
- Order_ID* is both part of the PK and an FK referencing ORDER
- Note: Product_ID* will be added to ORDER_LINE when the *appears_in* relationship (ORDER_LINE–PRODUCT) is processed in Step 4. The complete ORDER_LINE schema is shown in Section 7.6.9.

7.6.7 Company System Relationships and Keys

As established in Section 7.6.2, relationships between entities are defined using primary and foreign keys. A primary key (PK), such as Employee_ID, uniquely identifies each tuple in a relation. A foreign key (FK) is an attribute in one relation that references the primary key of another, creating a link between them.

Example of STEP 3: Mapping 1:1 Relationships

In this step, we demonstrate the mapping of the 1:1 relationship *manages* between EMPLOYEE and DEPARTMENT, from the Company Database ER Diagram, as shown in Figure 7.17. The EMPLOYEE relation has a foreign key Manages_Dept_ID that references the DEPARTMENT relation, indicating which department the employee manages.

The relational schema for the 1:1 relationship *manages* between EMPLOYEE and DEPARTMENT has two options:

Option 1 — Foreign key in the partial participation side: EMPLOYEE has partial participation (not all employees manage a department), so Manages_Dept_ID* is placed in the EMPLOYEE relation. In this case, the relational schema is:

DEPARTMENT (Department_ID, Dept_name, Location, Budget)

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Country, Zip_code, Manages_Dept_ID*)

Option 2 — Foreign key in the total participation side: Alternatively, a Manager_Employee_ID* foreign key can be placed in the DEPARTMENT relation, referencing EMPLOYEE.Employee_ID:

DEPARTMENT (Department_ID, Dept_name, Location, Budget, Manager_Employee_ID*)

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Country, Zip_code)

Option 1 is preferred since EMPLOYEE has partial participation — placing the FK on the partial side avoids NULL values in DEPARTMENT for departments that do not yet have a manager assigned.

Note: Department_ID* and Warehouse_ID* will be added to EMPLOYEE when the *works_for* and *works_at* relationships are processed below. The complete EMPLOYEE schema is shown in Section 7.6.9.

Example 1 of Step 4: Mapping 1:N Relationships

Figure 7.17 also shows the 1:N relationship *works_for* between DEPARTMENT and EMPLOYEE. Each employee works for exactly one department, and each department has many employees, forcing total participation on both sides.

Department_ID* is placed in the EMPLOYEE relation as a foreign key referencing DEPARTMENT.Department_ID.

The relational schemas resulting from both relationships shown in Figure 7.17 are:

DEPARTMENT (Department_ID, Dept_name, Location, Budget)

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Country, Zip_code, Department_ID*, Manages_Dept_ID*)

Note: Manages_Dept_ID* allows NULL since not all employees manage a department (partial participation). Department_ID* is NOT NULL because of total participation on both sides.

Note: Warehouse_ID* will be added to EMPLOYEE when the *works_at* relationship is processed. The complete EMPLOYEE schema is shown in Section 7.6.9.

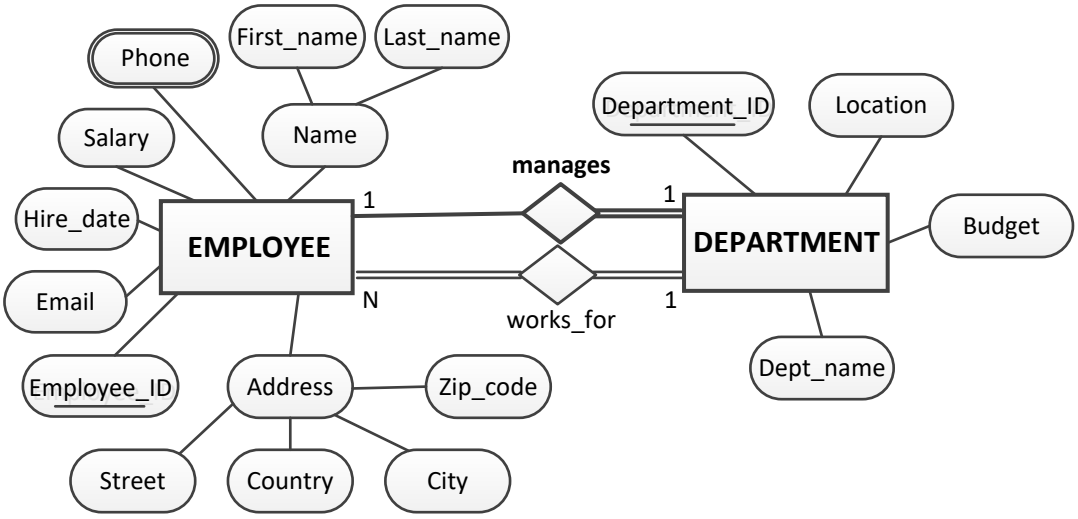


Figure 7.17: EMPLOYEE *manages* DEPARTMENT (1:1) and EMPLOYEE *works_for* DEPARTMENT (1:N)

Example 2 of STEP 4: Mapping 1:N Relationships

In this step, we demonstrate the mapping of the 1:N relationship *oversees* between DEPARTMENT and WAREHOUSE, from the Company Database ER Diagram, as shown in

Figure 7.18. A company department can oversee multiple warehouses, and each warehouse is overseen by exactly one department.

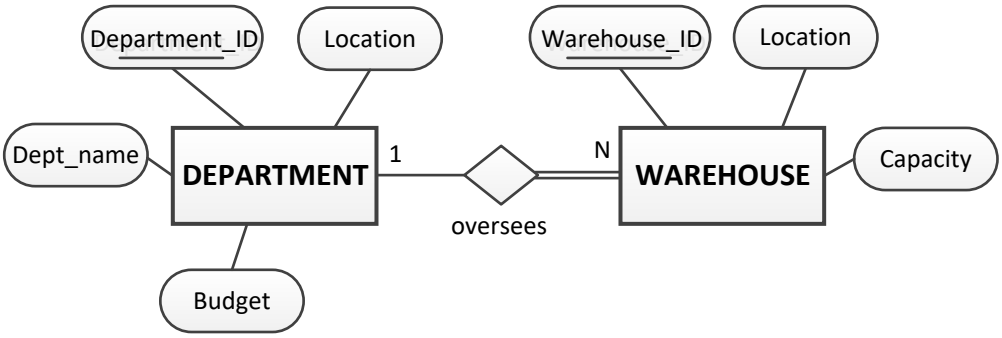


Figure 7.18: One-to-Many Department-Warehouse Relationship

The relational schema for the 1:N relationship *oversees* includes the relation WAREHOUSE with a foreign key Department_ID*. Each warehouse is associated with exactly one department from the relation DEPARTMENT.

DEPARTMENT (Department_ID, Dept_name, Location, Budget)

WAREHOUSE (Warehouse_ID, Location, Capacity, Department_ID*)

Note: The complete DEPARTMENT and WAREHOUSE schemas, including all foreign keys from other relationships, are shown in Section 7.6.9.

The complete list of 1:N (one-to-many) relationships in the Company System:

- *works_for* (EMPLOYEE–DEPARTMENT): An employee works for one department, but a department has many employees. Department_ID* is a foreign key in EMPLOYEE referencing DEPARTMENT.Department_ID.
- *oversees* (DEPARTMENT–WAREHOUSE): A department oversees many warehouses, but each warehouse is overseen by exactly one department. Department_ID* is a foreign key in WAREHOUSE referencing DEPARTMENT.Department_ID.
- *works_at* (EMPLOYEE–WAREHOUSE): An employee works at exactly one warehouse, but a warehouse has many employees. Warehouse_ID* is a foreign key in EMPLOYEE referencing WAREHOUSE.Warehouse_ID.
- *processes* (EMPLOYEE–ORDER): An employee can process many orders, but each order is processed by one employee. Employee_ID* is a foreign key in ORDER referencing EMPLOYEE.Employee_ID.
- *places* (CUSTOMER–ORDER): A customer can place many orders, but each order is placed by one customer. Customer_ID* is a foreign key in ORDER referencing CUSTOMER.Customer_ID.
- *contains* (ORDER–ORDER_LINE): An order has many order lines, but each order line belongs to exactly one order. Order_ID* is part of the composite primary key of ORDER_LINE and a foreign key referencing ORDER.Order_ID.
- *appears_in* (ORDER_LINE–PRODUCT): All order lines must reference a product. Product_ID* is a foreign key in ORDER_LINE referencing PRODUCT.Product_ID.

Example of Step 5: Mapping Binary M:N Relationships to Junction Relations

In this step, we demonstrate the mapping of the M:N relationship stored_in between PRODUCT and WAREHOUSE, from the Company Database ER Diagram, as shown in Figure 7.19. In this case, the many-to-many M:N relationship creates a new relation STORED_IN, whose primary key is formed by combining the primary keys of the two participating entities PRODUCT and WAREHOUSE.

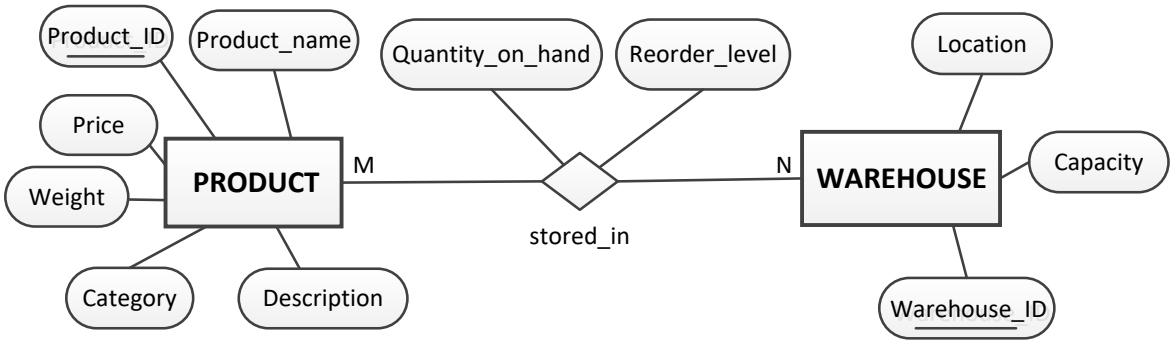


Figure 7.19: Many-to-Many Product-Warehouse Storage Relationship

The relational schema for mapping the M:N relationship includes the new relation STORED_IN, with foreign keys Product_ID* from the PRODUCT relation and Warehouse_ID* from the WAREHOUSE relation. The relationship attributes are Quantity_on_hand and Reorder_level.

STORED_IN (Product_ID*, Warehouse_ID*, Quantity_on_hand, Reorder_level)

Note: The complete PRODUCT and WAREHOUSE schemas are shown in Section 7.6.9.

The complete list of M:N (many-to-many) relationships in the Company System:

- *supplies* (SUPPLIER–PRODUCT): All products must have at least one supplier, and a supplier can supply many products. This is resolved by a new relation SUPPLIES, which includes Product_ID* and Supplier_ID* as foreign keys, along with relationship attributes such as Supply_date and Unit_cost.
- *stored_in* (PRODUCT–WAREHOUSE): A product can be stored in multiple warehouses, and a warehouse can store multiple products. The relation STORED_IN manages this with Product_ID* and Warehouse_ID* as foreign keys, plus attributes Quantity_on_hand and Reorder_level.

Example of Step 6: Mapping Multivalued Attributes to Separate Relations

In this step, **we demonstrate** the mapping of the multivalued attribute Phone from the EMPLOYEE **relation**, from the Company Database ER Diagram, as shown in Figure 7.20. To satisfy First Normal Form (1NF) (covered in Chapter 10), a multivalued attribute must be placed in a separate relation. In this case, the EMPLOYEE_PHONE relation is created to store multiple phone numbers per employee.

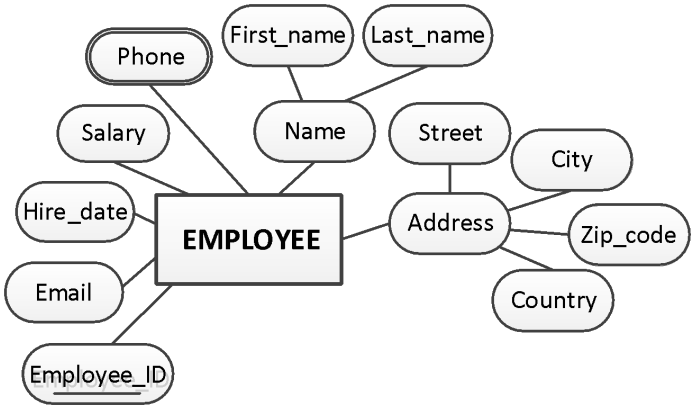


Figure 7.20: EMPLOYEE Entity with Complete Attribute Structure

The relational schema for mapping the multivalued Phone attribute includes the new relation EMPLOYEE_PHONE, with a foreign key Employee_ID* referencing the EMPLOYEE relation, and the Phone_number attribute. This new relation stores all phone numbers for each employee, identified by the combination of Employee_ID and Phone_number.

EMPLOYEE_PHONE (Employee_ID*, Phone_number)

The same approach applies to the multivalued Phone attribute in the CUSTOMER and SUPPLIER relations, resulting in two additional relations:

CUSTOMER_PHONE (Customer_ID*, Phone_number)

SUPPLIER_PHONE (Supplier_ID*, Phone_number)

7.6.8 Company System Integrity Constraints

Having completed all mapping steps, the resulting relational schema enforces the following integrity constraints to maintain data quality and consistency:

- **Entity Integrity:** Underlining the attribute name in a relational schema explicitly indicates that the chosen attribute or attributes serve as the primary key of the relation and cannot have a NULL value, guaranteeing that every tuple is uniquely identifiable. For example, Employee_ID uniquely identifies each tuple in the EMPLOYEE relation, and the composite key (Order_ID, Line_number) uniquely identifies each tuple in the ORDER_LINE relation.
- **Referential Integrity:** Foreign keys must reference a valid, existing primary key, and are marked with an asterisk (*) *to visually distinguish them from other attributes within a relation.* For example*, Department_ID* in the EMPLOYEE relation must correspond to a valid Department_ID in the DEPARTMENT relation. This prevents orphaned records and ensures all relationships remain valid.
- **Domain Constraints:** The data type for each **attribute** is specified (e.g., date, integer, **character string (varchar)**) to ensure data is stored in the correct format.
- **Business Rules:** The schema design enforces specific business rules derived from the requirements, such as every employee belonging to exactly one department (Department_ID* in EMPLOYEE is NOT NULL) and every warehouse being overseen by exactly one department (Department_ID* in WAREHOUSE is NOT NULL).

7.6.9 The Complete Relational Model Schema of the Company System

The following complete relational model schema for the Company Database System results from applying all seven steps of the ER-to-Relational mapping algorithm to the Company Database ER Diagram shown in Figure 7.14. Each relation was derived systematically: entity relations from Steps 1 and 2, foreign keys from Steps 3 and 4, junction relations from Step 5, and phone number relations from Step 6.

DEPARTMENT (Department_ID, Dept_name, Location, Budget)

WAREHOUSE (Warehouse_ID, Location, Capacity, Department_ID*)

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Country, Zip_code, Department_ID*, Warehouse_ID*, Manages_Dept_ID*)

CUSTOMER (Customer_ID, First_name, Last_name, Email, Join_date, Street, City, Country, Zip_code)

PRODUCT (Product_ID, Product_name, Description, Category, Price, Weight)

SUPPLIER (Supplier_ID, Supplier_name, First_name, Last_name, Email, Street, City, Country, Zip_code)

ORDER (Order_ID, Status, Order_date, Required_date, Shipped_date, Employee_ID*, Customer_ID*)

ORDER_LINE (Order_ID*, Line_number, Unit_price, Quantity, Discount, Product_ID*)

EMPLOYEE_PHONE (Employee_ID*, Phone_number)

CUSTOMER_PHONE (Customer_ID*, Phone_number)

SUPPLIER_PHONE (Supplier_ID*, Phone_number)

SUPPLIES (Product_ID*, Supplier_ID*, Supply_date, Unit_cost)

STORED_IN (Product_ID*, Warehouse_ID*, Quantity_on_hand, Reorder_level)

7.7 Mapping EER Constructs to Relations

7.7.1 Introduction to EER Mapping Challenges

Having established the three mapping strategies for EER specialization hierarchies in Section 7.5.6: Strategy A (Single-Relation (Flat) Approach), Strategy B (multiple relations with superclass), and Strategy C (multiple relations without superclass), we now apply these strategies to the actual EER diagrams developed in Chapter 5. Following the same step-by-step approach used in Section 7.6 for basic ER mapping, this section demonstrates how the University system's student and professor specialization hierarchies and the Company system's two independent employee specialization dimensions are each transformed into relational schemas. For each hierarchy, we identify the applicable strategy based on the specialization constraints established in Chapter 5, show the resulting relational schemas with their primary and foreign keys, and conclude with the complete integrated EER relational model for each case study.

7.7.2 University System EER Mapping

The University system contains two specialization hierarchies from Chapter 5 that require EER mapping: the student academic level hierarchy (Figure 7.21) and the professor academic rank hierarchy (Figure 7.22). Each is mapped using Step 8 of the EER-to-Relational mapping algorithm introduced in Section 7.5.6. For the complete integrated University EER diagram showing both hierarchies together with all relationships, refer to Figure 5.19.

Hierarchy 1: STUDENT → {BACHELOR_STUDENT, MASTERS_STUDENT, DOCTORAL_STUDENT}

In this step, we demonstrate the mapping of the student academic level specialization hierarchy from the University EER Diagram, as shown in Figure 7.21. As established in Chapter 5 (Section 5.7.1), STUDENT is specialized into three subclasses based on the Bologna Process academic structure, with a **(d, t)** — disjoint, total — constraint. This means every student belongs to exactly one academic level, and no student exists without an academic level classification.

The three mapping strategies introduced in Section 7.5.6 each produce a different schema structure for this hierarchy. The following overview shows all three options before justifying the chosen approach.

Strategy A: Single-Relation (Flat) Approach

A single relation is created for the entire hierarchy, including all superclass and subclass attributes, with a type discriminator Student_type indicating each student's academic level:

STUDENT (Student_ID, SSN, First_name, Last_name, Birth_date, Street, City, Zip_code, Country, Enrollment_date, ECTS_completed, Student_type, Major_field, Study_year, Total_ECTS_required, Defense_committee, Master_thesis_topic, Dissertation_topic, Expected_defense_date)

Where Student_type is the discriminator attribute with values {'Bachelor', 'Masters', 'Doctoral'}. A BACHELOR_STUDENT tuple would have NULL values for Master_thesis_topic, Dissertation_topic, and Expected_defense_date. A DOCTORAL_STUDENT tuple would have NULL values for Major_field, Study_year, and Master_thesis_topic.

Strategy B: Multiple Relations with Superclass

Separate relations are created for the superclass and each subclass. Each subclass relation contains only its specialized attributes, with the superclass primary key serving as both primary key and foreign key:

STUDENT (Student_ID, SSN, First_name, Last_name, Birth_date, Street, City, Zip_code, Country, Enrollment_date, ECTS_completed)

BACHELOR_STUDENT (Student_ID*, Major_field, Study_year, Defense_committee, Total_ECTS_required)

MASTERS_STUDENT (Student_ID*, Master_thesis_topic, Defense_committee, Total_ECTS_required)

DOCTORAL_STUDENT (Student_ID*, Dissertation_topic, Defense_committee, Expected_defense_date)

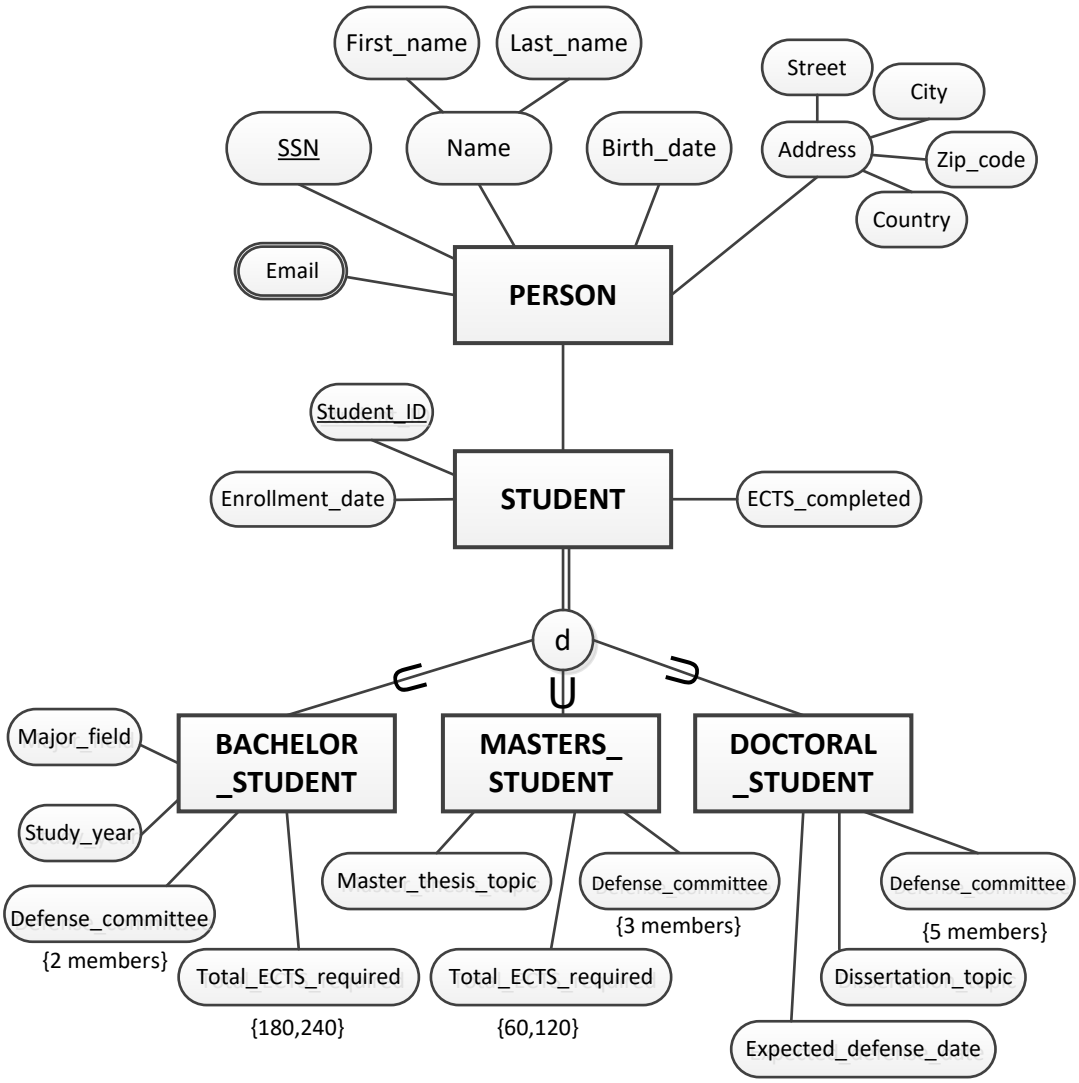


Figure 7.21: Student Academic Level Specialization Hierarchy (from Chapter 5, Figure 5.24)

Strategy C: Multiple Relations without Superclass

No superclass relation is created. Each subclass relation contains both the inherited superclass attributes and its own specialized attributes:

BACHELOR_STUDENT (Student_ID, SSN, First_name, Last_name, Birth_date, Street, City, Zip_code, Country, Enrollment_date, ECTS_completed, Major_field, Study_year, Defense_committee, Total_ECTS_required)

MASTERS_STUDENT (Student_ID, SSN, First_name, Last_name, Birth_date, Street, City, Zip_code, Country, Enrollment_date, ECTS_completed, Master_thesis_topic, Defense_committee, Total_ECTS_required)

DOCTORAL_STUDENT (Student_ID, SSN, First_name, Last_name, Birth_date, Street, City, Zip_code, Country, Enrollment_date, ECTS_completed, Dissertation_topic, Defense_committee, Expected_defense_date)

Chosen Approach: Strategy B

Strategy B is the most appropriate choice for the University student hierarchy for the following reasons:

- The University system has many queries that access general student information — enrollment lists, faculty assignments, attendance records — without needing academic-level-specific details. The STUDENT superclass relation supports these queries efficiently without joins to subclass relations.
- The three subclasses have meaningfully different specialized attributes (Major_field for bachelor's students, Master_thesis_topic for master's students, Dissertation_topic for doctoral students), making Strategy A's NULL-heavy single relation unwieldy and semantically confusing.
- Strategy C eliminates the superclass relation entirely, making it impossible to query all students regardless of academic level without a UNION across three subclass relations — impractical given how frequently the University system needs to treat all students uniformly.
- The foreign key constraint in Strategy B (Student_ID* in each subclass referencing STUDENT) naturally enforces that every bachelor's, master's, and doctoral student must first exist as a STUDENT, reinforcing the IS-A relationship from the EER model.

The relational schema applying Strategy B to the student specialization hierarchy is therefore:

STUDENT (Student_ID, SSN, First_name, Last_name, Birth_date, Street, City, Zip_code, Country, Enrollment_date, ECTS_completed)

BACHELOR_STUDENT (Student_ID*, Major_field, Study_year, Defense_committee, Total_ECTS_required)

MASTERS_STUDENT (Student_ID*, Master_thesis_topic, Defense_committee, Total_ECTS_required)

DOCTORAL_STUDENT (Student_ID*, Dissertation_topic, Defense_committee, Expected_defense_date)

Key Points:

- Student_ID* in each subclass relation serves as both the primary key and a foreign key referencing STUDENT.Student_ID; the dual-role key that enforces the IS-A relationship
- Total_ECTS_required appears in both BACHELOR_STUDENT {180, 240} and MASTERS_STUDENT {60, 120} with different domain constraints, as established in Chapter 5
- Defense_committee appears in all three subclasses with different cardinality constraints: 2 members for bachelor's, 3 for master's, 5 for doctoral

- The (d, t) constraint requires application-level enforcement: every STUDENT tuple must have exactly one corresponding tuple in either BACHELOR_STUDENT, MASTERS_STUDENT, or DOCTORAL_STUDENT; never more than one, never none
- Note: Email is a multivalued attribute visible in Figure 7.21 as a double oval on PERSON. It will be handled as a separate relation in Step 6 and is therefore not included in the STUDENT relation here
- Note: Faculty_ID* will be added to the STUDENT relation from the *belongs_to* relationship established in Section 7.6. The complete STUDENT schema, including all foreign keys, is shown in Section 7.7.3

Hierarchy 2: PROFESSOR → {ASSISTANT_PROFESSOR, ASSOCIATE_PROFESSOR, FULL_PROFESSOR}

In this step, we demonstrate the mapping of the professor academic rank specialization hierarchy from the University EER Diagram, as shown in Figure 7.22. As established in Chapter 5 (Section 5.7.1), PROFESSOR is specialized into three subclasses based on academic rank, with a (**d, t**) — disjoint, total — constraint. This means every professor belongs to exactly one academic rank subclass, and no professor exists without a rank classification.

Since this hierarchy shares the same relational structure and constraint type as Hierarchy 1, we apply Strategy B directly without repeating the full three-strategy comparison.

Chosen Approach: Strategy B

Strategy B is appropriate here for the same reasons as the student hierarchy: many university queries need general professor information — teaching assignments, faculty affiliations, office details — independently of rank-specific attributes. The PROFESSOR superclass relation efficiently supports these queries. The three subclasses have meaningfully different specialized attributes (Employment_type and Years_in_current_rank for assistant professors, Research_publications for associate professors, Years_as_full_professor and H_index for full professors), making Strategy A's NULL-heavy single relation unwieldy. Strategy C eliminates the superclass relation entirely, making it impossible to query all professors regardless of rank without a UNION across three subclass relations.

The relational schema applying Strategy B to the professor specialization hierarchy, based on what is visible in Figure 7.22, is:

PROFESSOR (<u>Professor_ID</u>, SSN, First_name, Last_name, Birth_date, Street, City, Zip_code, Country, Academic_title, Hire_date, Office)

ASSISTANT_PROFESSOR (<u>Professor_ID</u>*, Employment_type, Years_in_current_rank)

ASSOCIATE_PROFESSOR (<u>Professor_ID</u>*, Research_publications, Years_in_current_rank)

FULL_PROFESSOR (<u>Professor_ID</u>*, Years_as_full_professor, H_index)

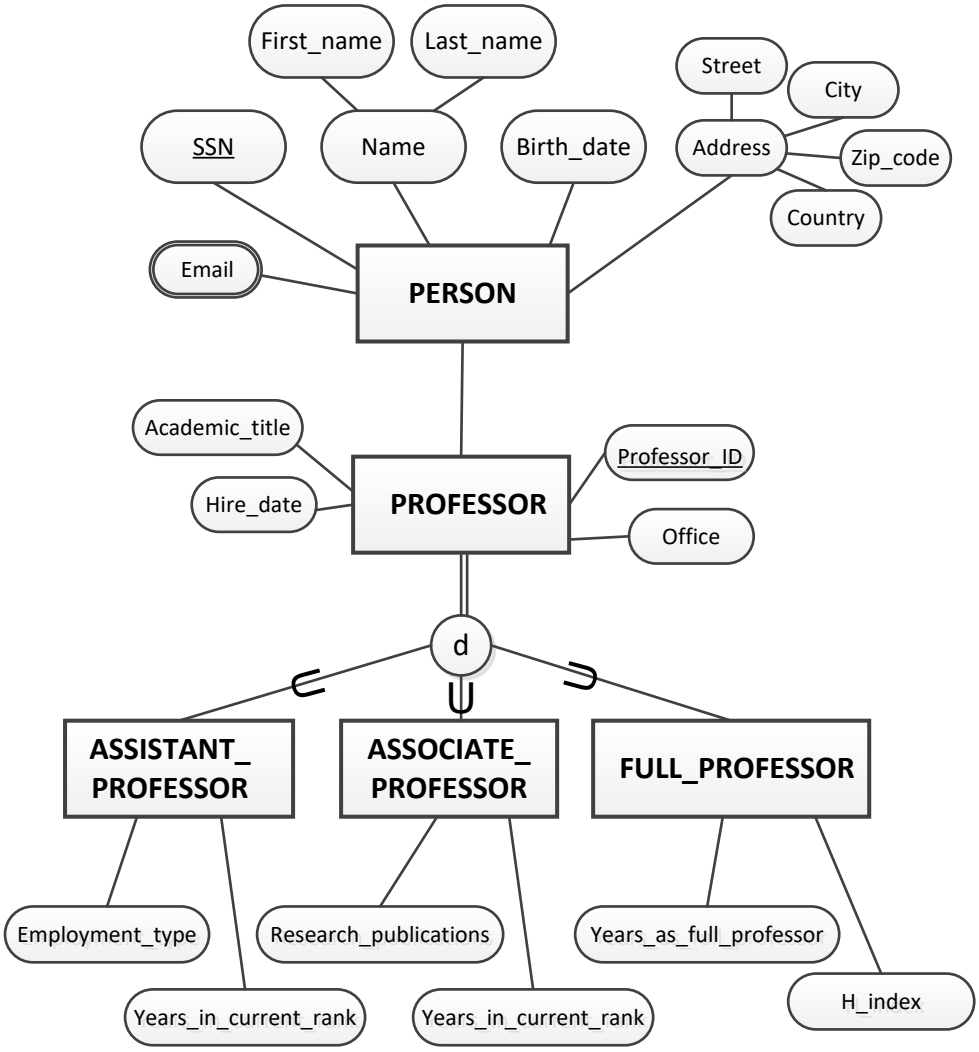


Figure 7.22: Professor Academic Rank Specialization Hierarchy (from Chapter 5, Figure 5.25)

Key Points:

- Professor_ID* in each subclass relation serves as both the primary key and a foreign key referencing PROFESSOR.Professor_ID; the dual-role key that enforces the IS-A relationship
- SSN is inherited from PERSON and included in the PROFESSOR superclass relation
- The (d, t) constraint requires application-level enforcement: every PROFESSOR tuple must have exactly one corresponding tuple in either ASSISTANT_PROFESSOR, ASSOCIATE_PROFESSOR, or FULL_PROFESSOR — never more than one, never none
- Academic_title remains in the PROFESSOR superclass relation as a simple attribute capturing the general title; the subclass relations capture rank-specific quantitative and performance attributes
- Note: Email is a multivalued attribute visible in Figure 7.22 as a double oval on PERSON. It will be handled as a separate relation and is therefore not included in the PROFESSOR relation here

- Note: Faculty_ID* and Heads_Faculty_ID* are not visible in Figure 7.22; they will be added to the PROFESSOR relation from the *affiliated_with* and *heads* relationships established in Section 7.6. The complete PROFESSOR schema is shown in Section 7.7.3

7.7.3 Complete University EER Relational Schema

The following complete relational schema for the University system results from integrating the EER mapping from Section 7.7.2 with the base ER schema from Section 7.6.5, corresponding to the Complete Integrated University EER Model shown in Figure 7.23 (from Chapter 5, Figure 5.28). For additional detail on the specialization hierarchy structure alone, refer to Figure 5.27. Each relation was derived systematically: the base entity and relationship relations from Sections 7.6.1–7.6.5, the student academic level hierarchy from Hierarchy 1 (Section 7.7.2), the professor academic rank hierarchy from Hierarchy 2 (Section 7.7.2), and the remaining PERSON subclasses visible in Figure 5.28. Strategy B is applied consistently throughout all levels of the hierarchy — PERSON as the top-level superclass relation, each role-based entity (STUDENT, PROFESSOR, INSTRUCTOR, STAFF, ALUMNUS) as an intermediate subclass relation, and each academic-level or rank-based entity as a second-level subclass relation.

Top-Level Superclass Relation:

PERSON (SSN, First_name, Last_name, Birth_date, Street, City, Zip_code, Country)

Multivalued Attribute Relation (inherited from PERSON):

EMAIL (SSN*, Email)

First-Level Subclass Relations (PERSON subclasses):

STUDENT (SSN*, Student_ID, Enrollment_date, ECTS_completed, Faculty_ID*)

PROFESSOR (SSN*, Professor_ID, Academic_title, Hire_date, Office, Faculty_ID*, Heads_Faculty_ID*)

INSTRUCTOR (SSN*, Instructor_ID, Salary, Hire_date, Department_ID, Teaching_schedule, Course_taught, Student_evaluation_average)

STAFF (SSN*, Staff_ID, Hire_date, Department_assigned, Job_title, Office_location, Employment_status)

ALUMNUS (SSN*, Alumni_ID, Graduation_date, Degree_earned, Current_occupation)

Second-Level Subclass Relations — Student Academic Level Hierarchy (Strategy B — from Section 7.7.2, Hierarchy 1):

BACHELOR_STUDENT (SSN*, Major_field, Study_year, Defense_committee, Total_ECTS_required)

MASTERS_STUDENT (SSN*, Master_thesis_topic, Defense_committee, Total_ECTS_required)

DOCTORAL_STUDENT (SSN*, Dissertation_topic, Defense_committee, Expected_defense_date)

Second-Level Subclass Relations — Professor Academic Rank Hierarchy (Strategy B — from Section 7.7.2, Hierarchy 2):

ASSISTANT_PROFESSOR (SSN*, Employment_type, Years_in_current_rank)

ASSOCIATE_PROFESSOR (SSN*, Research_publications, Years_in_current_rank)

FULL_PROFESSOR (SSN*, Years_as_full_professor, H_index)

Shared Subclass — Lattice Structure:

TEACHING_ASSISTANT (SSN*, Hours_per_week, Semester_appointment, TA_training_completed, Supervisor_ID*)

Core Entity Relations (from Section 7.6.5):

FACULTY (Faculty_ID, Name, Budget, Location)

COURSE (Course_ID, Title, Credits, Description, Faculty_ID*, Professor_ID*)

TEXTBOOK (ISBN, Title, Author, Edition, Publisher, Available_copies)

Weak Entity Relation:

CLASS_SESSION (Course_ID*, Session_number, Year, Room, Semester, Time, Day, Duration)

Relationship Relations:

PREREQUISITE (Course_ID*, Prerequisite_Course_ID*)

ENROLLS_IN (SSN*, Course_ID*, Enrollment_date, Grade)

ATTENDS (SSN*, Course_ID*, Session_number*, Year*, Attendance)

ASSIGNMENT (SSN*, Course_ID*, ISBN*)

Notes on the Complete Schema:

- PERSON is the top-level superclass. SSN* in all first-level subclass relations (STUDENT, PROFESSOR, INSTRUCTOR, STAFF, ALUMNUS) serves as both the primary key and a foreign key referencing PERSON.SSN, applying Strategy B at the first level of the hierarchy.
- SSN* in all second-level subclass relations (BACHELOR_STUDENT, MASTERS_STUDENT, DOCTORAL_STUDENT, ASSISTANT_PROFESSOR, ASSOCIATE_PROFESSOR, FULL_PROFESSOR) serves as both the primary key and a foreign key referencing the respective first-level relation — STUDENT.SSN for student subclasses and PROFESSOR.SSN for professor subclasses — applying Strategy B at the second level.
- EMAIL stores multiple email addresses per person, identified by the combination of SSN* and Email, since Email is a multivalued attribute on PERSON inherited by all subclasses.
- TEACHING_ASSISTANT.SSN* serves as both the primary key and a dual foreign key referencing both DOCTORAL_STUDENT.SSN (through the student inheritance path) and INSTRUCTOR.SSN (through the instructor inheritance path), reflecting the lat-

tice structure established in Chapter 5. Supervisor_ID* references PROFESSOR.SSN through the *supervised_by* relationship.

- STUDENT.Faculty_ID* references FACULTY.Faculty_ID, derived from the *belongs_to* relationship.
- PROFESSOR.Faculty_ID* references FACULTY.Faculty_ID, derived from the *affiliated_with* relationship.
- PROFESSOR.Heads_Faculty_ID* references FACULTY.Faculty_ID, derived from the *heads* relationship.
- COURSE.Professor_ID* references PROFESSOR.Professor_ID, derived from the *teaches* relationship. Although SSN is the primary key of the PROFESSOR relation in the integrated EER schema, Professor_ID is retained as an alternate key with a UNIQUE constraint, making it a valid foreign key target and preserving semantic clarity in relations that specifically link to a professor role.
- COURSE.Faculty_ID* references FACULTY.Faculty_ID, derived from the *offers* relationship.
- INSTRUCTOR contains Department_ID as a regular attribute (not a foreign key) since no DEPARTMENT relation exists in the current University relational schema. In a future extension of the University system, a DEPARTMENT relation could be introduced, at which point Department_ID would become a foreign key referencing DEPARTMENT.Department_ID.
- STAFF contains Department_assigned as a regular attribute (not a foreign key) for the same reason — it identifies the department to which a staff member is assigned, but no DEPARTMENT relation currently exists in the University schema to reference. As with INSTRUCTOR, the introduction of a DEPARTMENT relation in a future extension would allow Department_assigned to be formalized as a foreign key.
- STAFF.Employment_status has a domain constraint {Full-time, Part-time, Contract} as visible in Figure 7.23.
- ENROLLS_IN and ATTENDS use SSN* rather than a separate Student_ID* since SSN is the primary key carried through the entire inheritance chain from PERSON down to STUDENT and its subclasses.
- ASSIGNMENT.SSN* references PROFESSOR.SSN, derived from the *assignment* ternary relationship between PROFESSOR, COURSE, and TEXTBOOK.
- Professor_ID, Student_ID, Instructor_ID, Staff_ID, and Alumni_ID are retained as regular (non-key) attributes in their respective first-level subclass relations to support application-level identification and display purposes, but SSN is the relational primary key at all levels of the hierarchy.

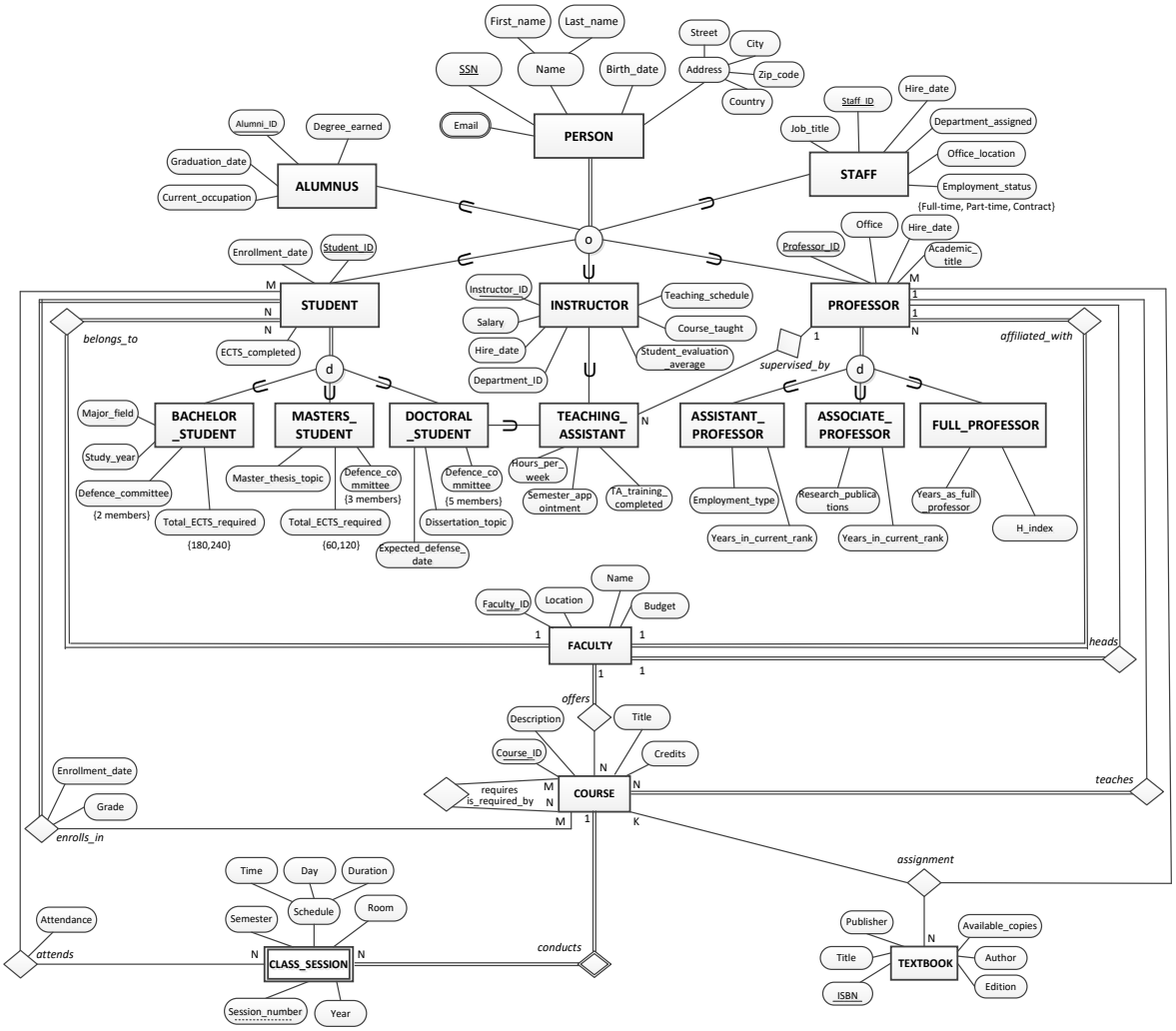


Figure 7.23: Complete Integrated University EER Model (from Chapter 5, Figure 5.28)

7.7.4 Company System EER Mapping

The Company system contains two independent specializations of the EMPLOYEE entity from Chapter 5 (Section 5.7.2): the Employment Type specialization and the Job Function specialization. Unlike the University system's hierarchical lattice, these two specializations are orthogonal — they operate independently, and an employee can simultaneously belong to one subclass from each dimension. Each specialization is mapped separately using Step 8 of the EER-to-Relational mapping algorithm. The Employment Type specialization is shown in Figure 7.24 (from Chapter 5, Figure 5.30) and the Job Function specialization is shown in Figure 7.25 (from Chapter 5, Figure 5.29). For the complete Company EER diagram showing both specializations together, refer to Figure 5.32.

Specialization 1: EMPLOYEE → {FULL_TIME_EMPLOYEE, PART_TIME_EMPLOYEE, CONTRACT_EMPLOYEE}

In this step, we demonstrate the mapping of the Employment Type specialization from the Company EER Diagram, as shown in Figure 7.24. As established in Chapter 5 (Section 5.7.2), EMPLOYEE is specialized into three employment type subclasses with a **(d, t)** —

disjoint, total — constraint. This means every employee belongs to exactly one employment type, and no employee exists without an employment type classification.

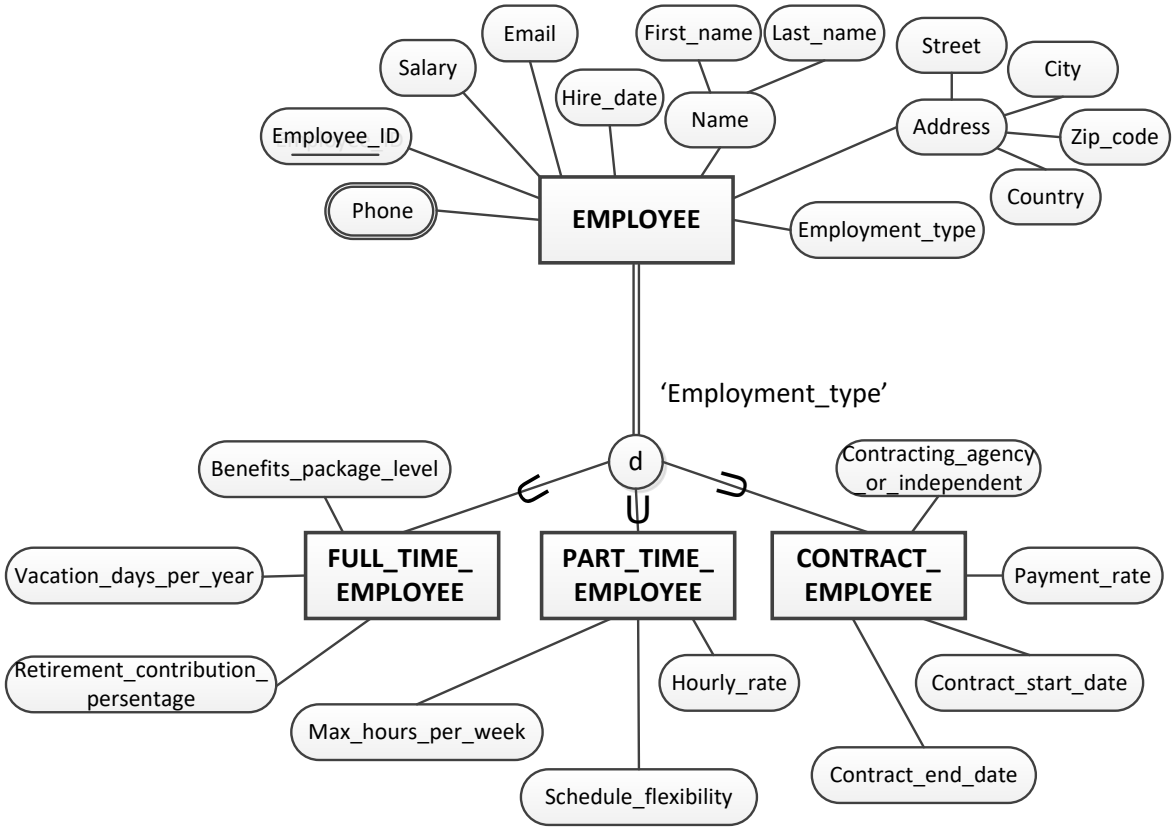


Figure 7.24: Employment Type Specialization Hierarchy (from Chapter 5, Figure 5.30)

Since this hierarchy uses the same constraint type (d, t) as the University student hierarchy in Section 7.7.2, we apply Strategy B directly without repeating the full three-strategy comparison.

Chosen Approach: Strategy B

Strategy B is appropriate because the EMPLOYEE superclass relation must exist to store the base employee attributes used across the entire Company system. Additionally, the three subclasses have clearly distinct specialized attributes, making Strategy A's NULL-heavy single relation unnecessarily complex. Strategy C is not suitable because it would eliminate the EMPLOYEE superclass relation, which is central to the entire Company relational schema.

The relational schema applying Strategy B to the Employment Type specialization, based on what is visible in Figure 7.24, is:

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Zip_code, Country, Employment_type)

FULL_TIME_EMPLOYEE (Employee_ID*, Benefits_package_level, Vacation_days_per_year, Retirement_contribution_percentage)

PART_TIME_EMPLOYEE (Employee_ID*, Hourly_rate, Max_hours_per_week, Schedule_flexibility)

CONTRACT_EMPLOYEE (Employee_ID*, Payment_rate, Contract_start_date, Contract_end_date, Contracting_agency_or_independent)

Key Points:

- Employee_ID* in each subclass relation serves as both the primary key and a foreign key referencing EMPLOYEE.Employee_ID. It is the dual-role key that enforces the IS-A relationship
- Employment_type is the discriminator attribute visible on EMPLOYEE in Figure 7.24, indicating which employment type subclass each employee belongs to: {'Full-time', 'Part-time', 'Contract'}
- The (d, t) constraint requires application-level enforcement: every EMPLOYEE tuple must have exactly one corresponding tuple in either FULL_TIME_EMPLOYEE, PART_TIME_EMPLOYEE, or CONTRACT_EMPLOYEE; never more than one, never none
- *Note*: Phone is a multivalued attribute visible in Figure 7.24 as a double oval on EMPLOYEE. It is handled as a separate relation EMPLOYEE_PHONE in the base ER mapping (Section 7.6.8) and is therefore not included in the EMPLOYEE relation here

Specialization 2: EMPLOYEE → {MANAGER_EMPLOYEE, SALES_EMPLOYEE, WAREHOUSE_EMPLOYEE, ADMIN_EMPLOYEE}

In this step, we demonstrate the mapping of the Job Function specialization from the Company EER Diagram, as shown in Figure 7.25. As established in Chapter 5 (Section 5.7.2), EMPLOYEE is specialized into four job function subclasses with a **(d, p)** — disjoint, partial — constraint. This means an employee belongs to at most one job function subclass, and some employees, general employees who perform standard functions, do not belong to any subclass.

The Job Function specialization has a **(d, p)** constraint, and is the only partial specialization in our case studies. Strategy B is the only strategy that naturally accommodates partial participation. Strategy A would produce NULL values for all job-function-specific attributes for general employees with no specialized function, which is semantically misleading. Strategy C cannot support partial specialization at all since it eliminates the superclass relation, leaving general employees with nowhere to be stored. Strategy B handles this naturally — general employees such as E4 (Anita Toshevska) simply have a tuple in EMPLOYEE with no corresponding job function subclass tuple.

The relational schema applying Strategy B to the Job Function specialization, based on what is visible in Figure 7.25, is:

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Zip_code, Country, Job_function)

MANAGER_EMPLOYEE (Employee_ID*, Management_level, Budget_authority, Strategic_planning_responsibility, Leadership_training_complete)

SALES_EMPLOYEE (Employee_ID*, Sales_territory, Commission_rate, Monthly_sales_quota, Customer_portfolio_size)

WAREHOUSE_EMPLOYEE (Employee_ID*, Shift_schedule, Physical_location, Inventory_management_system_trained)

ADMIN_EMPLOYEE (Employee_ID*, Office_location, Administrative_level, Document_management_responsibility)

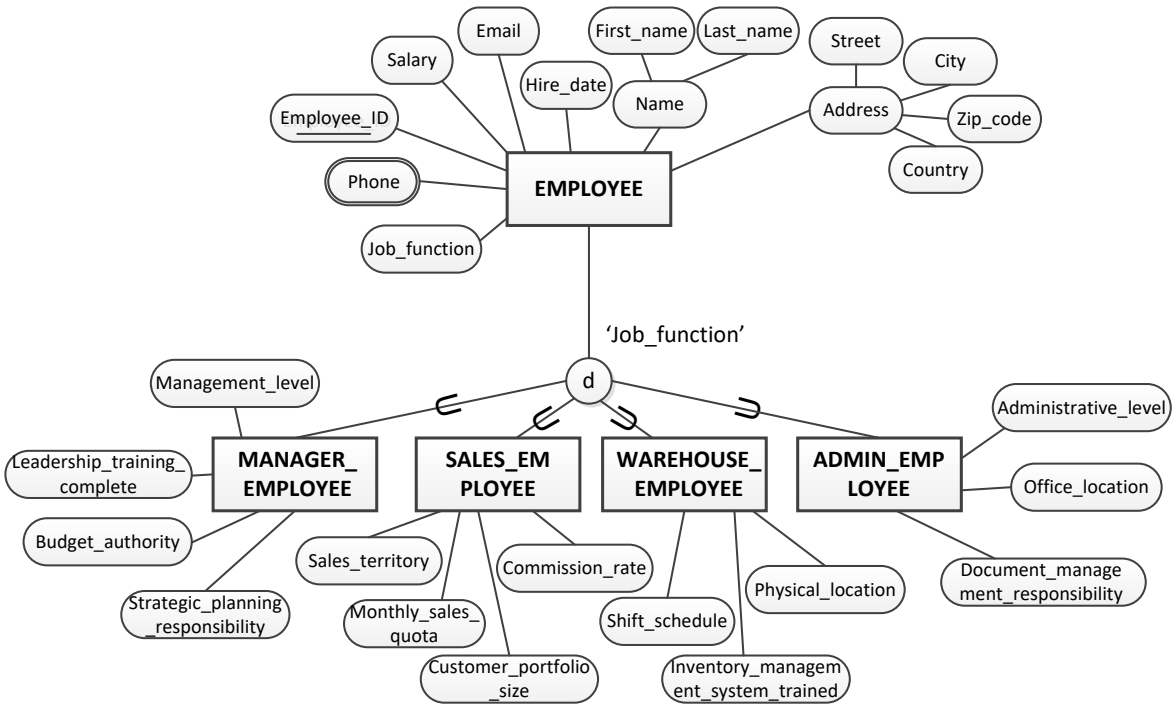


Figure 7.25: Job Function Specialization Hierarchy (from Chapter 5, Figure 5.29)

Key Points:

- Employee_ID* in each subclass relation serves as both the primary key and a foreign key referencing EMPLOYEE.Employee_ID
- Job_function is the discriminator attribute visible on EMPLOYEE in Figure 7.25, indicating which job function subclass each employee belongs to — {NULL, 'Manager', 'Sales', 'Warehouse', 'Admin'}, where NULL represents general employees with no specialized function
- The (d, p) constraint means some EMPLOYEE tuples have no corresponding job function subclass tuple. These represent general employees performing standard duties without specialized classification. From our data, E4 (Anita Toshevska) is an example of a general employee with no job function subclass entry
- *Note*: Phone is a multivalued attribute visible in Figure 7.25 as a double oval on EMPLOYEE. It is handled as a separate relation EMPLOYEE_PHONE in the base ER mapping (Section 7.6.8) and is therefore not included in the EMPLOYEE relation here

The two specializations are independent: an employee can simultaneously belong to one employment type subclass AND one job function subclass, or to an employment type subclass only. For example, E1 (Zoran Pejovski) belongs to both

MANAGER_EMPLOYEE and FULL_TIME_EMPLOYEE; E4 (Anita Toshevska) belongs to FULL_TIME_EMPLOYEE only, with no job function subclass entry.

7.7.5 Complete Company EER Relational Schema

The following complete relational schema for the Company system results from integrating both independent EER specializations from Section 7.7.4 with the base ER schema from Section 7.6.9, corresponding to the Complete Integrated Company EER Model shown in Figure 7.26 (from Chapter 5, Figure 5.33). Each relation was derived systematically: the base entity and relationship relations from Sections 7.6.6–7.6.9, the Employment Type specialization from Specialization 1 (Section 7.7.4), and the Job Function specialization from Specialization 2 (Section 7.7.4). Strategy B is applied to both specializations, with the EMPLOYEE superclass relation serving as the central entity connecting all base ER relationships and both specialization dimensions.

Note: Comparing this complete EER schema with the base ER schema from Section 7.6.9, two important refinements emerge from the EER mapping:

- Manages_Dept_ID* moves from EMPLOYEE into MANAGER_EMPLOYEE, where it semantically belongs; only managers manage a department
- Warehouse_ID* moves from EMPLOYEE into WAREHOUSE_EMPLOYEE, where it semantically belongs; only warehouse employees are assigned to specific warehouses through the *works_at* relationship

In the base ER schema, both foreign keys were placed in EMPLOYEE as nullable attributes. The EER mapping produces a more precise and semantically accurate schema by placing each foreign key in its specific subclass relation.

Superclass Relation:

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Zip_code, Country, Job_function, Employment_type, Department_ID*)

Multivalued Attribute Relation:

EMPLOYEE_PHONE (Employee_ID*, Phone_number)

Employment Type Subclass Relations (Strategy B — from Section 7.7.4, Specialization 1):

FULL_TIME_EMPLOYEE (Employee_ID*, Benefits_package_level, Vacation_days_per_year, Retirement_contribution_percentage)

PART_TIME_EMPLOYEE (Employee_ID*, Hourly_rate, Max_hours_per_week, Schedule_flexibility)

CONTRACT_EMPLOYEE (Employee_ID*, Payment_rate, Contract_start_date, Contract_end_date, Contracting_agency_or_independent)

Job Function Subclass Relations (Strategy B — from Section 7.7.4, Specialization 2):

MANAGER_EMPLOYEE (Employee_ID*, Management_level, Budget_authority, Strategic_planning_responsibility, Leadership_training_complete, Manages_Dept_ID*)

SALES_EMPLOYEE (Employee_ID*, Sales_territory, Commission_rate, Monthly_sales_quota, Customer_portfolio_size)

WAREHOUSE_EMPLOYEE (Employee_ID*, Shift_schedule, Physical_location, Inventory_management_system_trained, Warehouse_ID*)

ADMIN_EMPLOYEE (Employee_ID*, Office_location, Administrative_level, Document_management_responsibility)

Core Entity Relations (from Section 7.6.9):

DEPARTMENT (Department_ID, Dept_name, Location, Budget)

WAREHOUSE (Warehouse_ID, Location, Capacity, Department_ID*)

CUSTOMER (Customer_ID, First_name, Last_name, Email, Join_date, Street, City, Country, Zip_code)

PRODUCT (Product_ID, Product_name, Description, Category, Price, Weight)

SUPPLIER (Supplier_ID, Supplier_name, Contact_first_name, Contact_last_name, Email, Street, City, Country, Zip_code)

ORDER (Order_ID, Status, Order_date, Required_date, Shipped_date, Employee_ID*, Customer_ID*)

Weak Entity Relation:

ORDER_LINE (Order_ID*, Line_number, Unit_price, Quantity, Discount, Product_ID*)

Multivalued Attribute Relations (from Section 7.6.9):

CUSTOMER_PHONE (Customer_ID*, Phone_number)

SUPPLIER_PHONE (Supplier_ID*, Phone_number)

Relationship Relations (from Section 7.6.9):

SUPPLIES (Product_ID*, Supplier_ID*, Supply_date, Unit_cost)

STORED_IN (Product_ID*, Warehouse_ID*, Quantity_on_hand, Reorder_level)

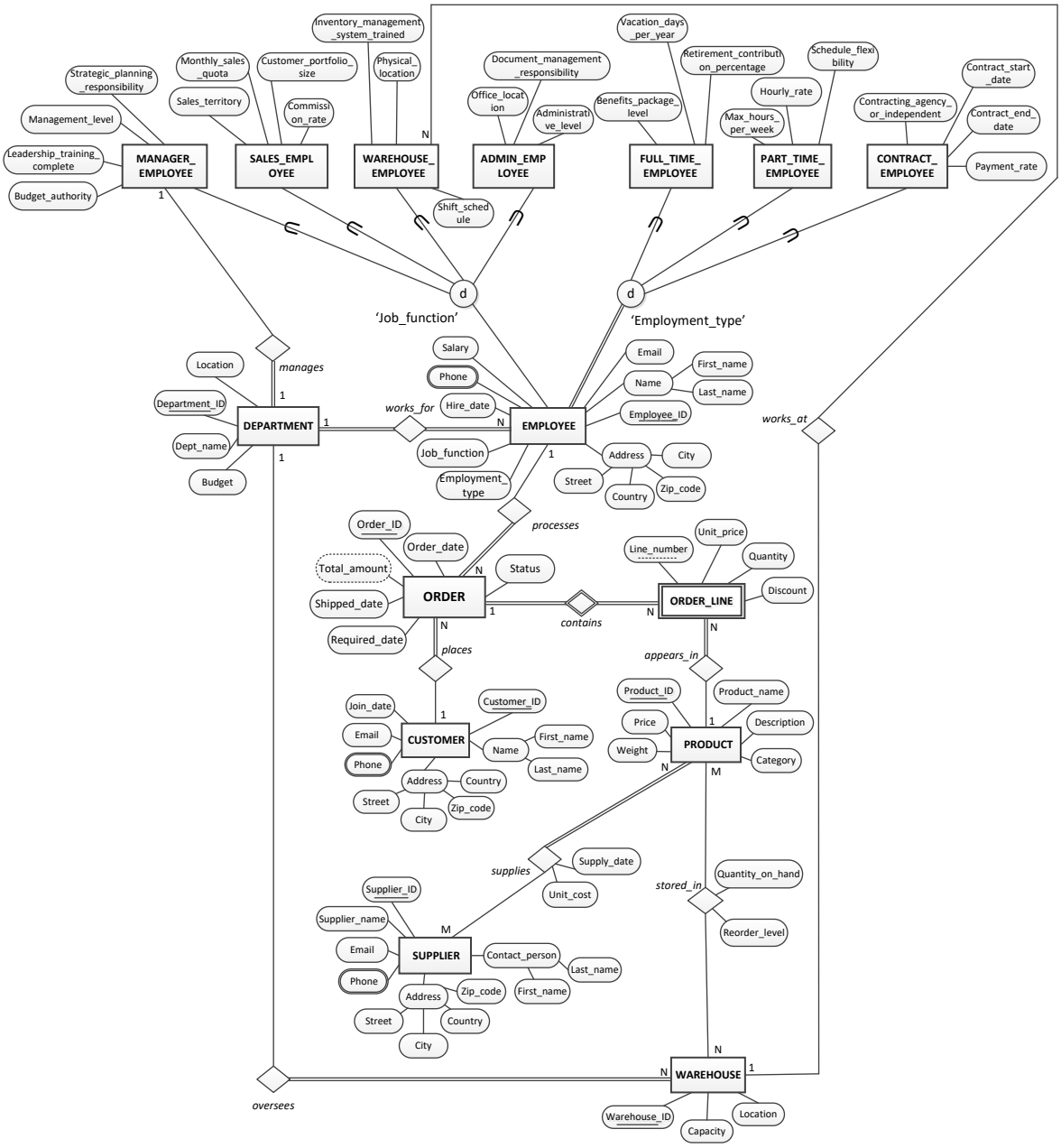


Figure 7.26: Complete Integrated Company EER Model (from Chapter 5, Figure 5.33)

Notes on the Complete Schema:

- EMPLOYEE is the central superclass relation. Job_function and Employment_type are discriminator attributes retained in EMPLOYEE. Job_function may be NULL for general employees (partial specialization, d, p); Employment_type must always have a value (total specialization, d, t)
- Employee_ID* in all subclass relations serves as both the primary key and a foreign key referencing EMPLOYEE.Employee_ID
- The two specializations are independent: every EMPLOYEE tuple must have exactly one employment type subclass entry, while job function subclass entries are optional — general employees have none

- EMPLOYEE.Department_ID* references DEPARTMENT.Department_ID, derived from the *works_for* relationship (N:1) — every employee works for exactly one department
- MANAGER_EMPLOYEE.Manages_Dept_ID* references DEPARTMENT.Department_ID, derived from the *manages* relationship (1:1) — only managers manage a department, so this foreign key belongs in MANAGER_EMPLOYEE
- WAREHOUSE_EMPLOYEE.Warehouse_ID* references WAREHOUSE.Warehouse_ID, derived from the *works_at* relationship (N:1) — only warehouse employees are assigned to warehouses, so this foreign key belongs in WAREHOUSE_EMPLOYEE
- WAREHOUSE.Department_ID* references DEPARTMENT.Department_ID, derived from the *oversees* relationship (1:N) — each warehouse is overseen by exactly one department
- ORDER.Employee_ID* references EMPLOYEE.Employee_ID, derived from the *processes* relationship
- ORDER.Customer_ID* references CUSTOMER.Customer_ID, derived from the *places* relationship
- ORDER_LINE.Product_ID* references PRODUCT.Product_ID, derived from the *appears_in* relationship
- Total_amount in ORDER is a derived attribute calculated from the sum of all ORDER_LINE amounts. It is not stored in the relation and is calculated when needed
- SUPPLIER contact person attributes are stored as Contact_first_name and Contact_last_name, representing the components of the composite attribute Contact_person from the SUPPLIER entity

7.8 Summary

This chapter provides a comprehensive overview of the **Relational Model,** the foundational framework for modern database systems. It begins by explaining how this model is essential for a University Management System, transforming conceptual Entity-Relationship (ER) diagrams into structured, logical relations (tables).

The chapter defines the relational model as a mathematically sound approach that organizes data into two-dimensional tables. Each table consists of rows, called tuples, and columns, called **attributes**. The model's core characteristics include its logical structure, which is defined by a **schema**, and its enforcement of data integrity through **keys** and **constraints**. This simplicity and flexibility have made it the backbone of modern relational database systems, which are experiencing significant growth, driven by cloud-based solutions and digital transformation initiatives.

A key focus of the chapter is the role of **relational keys** in maintaining data integrity. It defines and differentiates between several types of keys:

Superkey: An attribute or set of attributes that uniquely identifies a tuple.

Candidate Key: A minimal superkey that is both unique and irreducible.

Primary Key: The chosen candidate key that uniquely identifies each tuple and cannot be null.

Foreign Key: An attribute that links a tuple to a tuple in another relation, enforcing referential integrity.

The chapter also elaborates on essential integrity constraints, including entity integrity (which prevents primary keys from being null) and referential integrity (which ensures foreign keys reference existing primary keys).

The chapter presents a systematic seven-step algorithm for transforming Entity-Relationship (ER) diagrams into relational schemas. Step 1 maps regular entity types to relations with their simple attributes, establishing primary keys from the ER model's key attributes. Step 2 addresses weak entity types by creating relations that combine the partial key with the owner entity's primary key to form composite primary keys. Step 3 handles binary 1:1 relationships by adding foreign keys to either participating relation, while Step 4 manages binary 1:N relationships by placing foreign keys on the N-side to avoid NULL values. Step 5 transforms binary M:N relationships into separate relations with composite primary keys comprising foreign keys from both participating entities. Step 6 addresses multi-valued attributes by creating separate relations with composite keys, and Step 7 maps n-ary relationships (degree ≥ 3) to relations with primary keys combining all participating entity keys. Throughout the mapping process, designers must consider attribute placement, NULL value avoidance, and referential integrity enforcement. The algorithm is demonstrated comprehensively through both the University database (mapping students, courses, enrollments, and weak entity CLASS_SESSION) and the Company database (mapping employees, orders, products, and the ternary SUPPLIES relationship), establishing complete relational schemas that preserve all semantic information from the conceptual ER models.

The chapter extends the basic ER-to-Relational mapping algorithm with specialized strategies for Enhanced Entity-Relationship (EER) constructs, introduced in Section 7.5.6 and demonstrated through complete case study applications in Section 7.7. Three mapping strategies for specialization hierarchies are presented: Strategy A (single relation with a type discriminator), Strategy B (multiple relations with a superclass), and Strategy C (multiple relations without a superclass). The choice between strategies is guided by the specialization constraints — disjoint or overlapping, total or partial — established in the EER diagram. Strategy B was consistently applied across both case studies as the most widely applicable approach.

Review Questions

Conceptual Understanding

7.1. What is the relational model? Explain how it differs from conceptual models such as ER and EER diagrams, and why it is necessary as an intermediate step between conceptual design and physical database implementation.

7.2. Define the following relational model terms and explain the role of each in organizing and maintaining data: (a) relation, (b) tuple, (c) attribute, (d) domain, (e) schema.

7.3. Why are tuples in a relation not ordered? Why are duplicate tuples not allowed? How do these properties contribute to the mathematical foundations of the relational model?

7.4. Explain the difference between a superkey, a candidate key, a primary key, and an alternate key. Why does the relational model require us to distinguish between these key types rather than simply designating any unique attribute as the primary key?

7.5. Define foreign key and explain how it implements relationships between relations in the relational model. Using examples from the University system, describe at least two foreign key relationships and identify which entity is on the one side and which is on the many side.

7.6. Explain the difference between entity integrity and referential integrity. Why is each constraint considered essential to the correctness of a relational database? Provide one example of a violation of each constraint using the Company system.

7.7. What is the difference between a simple attribute, a composite attribute, a multivalued attribute, and a derived attribute? Explain how each type is handled differently during the ER-to-Relational mapping process, providing examples from either case study.

7.8. How does the mapping of a weak entity differ from the mapping of a strong entity? Using CLASS_SESSION from the University system and ORDER_LINE from the Company system, explain why each is a weak entity and how their composite primary keys are formed.

ER-to-Relational Mapping

7.9. Describe the seven-step ER-to-Relational mapping algorithm. For each step, identify which ER construct it addresses and give one example from either the University or Company case study showing the result of applying that step.

7.10. Explain how binary 1:1 relationships are mapped to relational schemas. What are the two main options available to the designer, and what factors determine which option is preferred? Illustrate your answer using the *heads* relationship between PROFESSOR and FACULTY from the University system.

7.11. Explain how binary 1:N relationships are mapped to relational schemas. Using the *belongs_to* relationship between STUDENT and FACULTY as an example, identify which relation receives the foreign key, and explain what participation constraints determine whether the foreign key allows NULL values.

7.12. Explain how M:N relationships are mapped to relational schemas. Why do M:N relationships require a junction relation while 1:N relationships do not? Using the *enrolls_in* relationship between STUDENT and COURSE, identify the primary key of the junction relation and any relationship attributes it contains.

7.13. The University system contains a recursive M:N relationship (*prerequisite*) on the COURSE entity. Explain what a recursive relationship is and how it is mapped to a relational schema. How does the PREREQUISITE relation reference COURSE twice, and what names are given to the two foreign keys?

7.14. Explain how N-ary relationships are mapped to relational schemas. Using the ternary *assignment* relationship between PROFESSOR, COURSE, and TEXTBOOK, describe the resulting ASSIGNMENT relation, identify its composite primary key, and explain why each component is both a primary key attribute and a foreign key.

7.15. The Company system includes a *manages* (1:1) relationship and a *works_for* (1:N) relationship, both between EMPLOYEE and DEPARTMENT, as shown in Figure 7.17. Explain how both relationships are handled in the relational schema, what foreign keys are produced, and how participation constraints affect whether each foreign key allows NULL values.

EER Mapping

7.16. Describe the three EER mapping strategies (Strategy A, Strategy B, and Strategy C) for specialization hierarchies. For each strategy, explain how relations are structured, what happens to superclass and subclass attributes, and when the strategy is most appropriate.

7.17. Table 7.12 in this chapter provides guidance for choosing a mapping strategy based on specialization constraints. Explain why Strategy B is the only appropriate choice for overlapping specializations (o, t and o, p), while Strategy C is only appropriate for disjoint, total (d, t) specializations.

7.18. The University system applies Strategy B to the student academic level specialization (STUDENT → {BACHELOR_STUDENT, MASTERS_STUDENT, DOCTORAL_STUDENT}, d, t). Explain what the (d, t) constraint means, how Strategy B represents this constraint in the relational schema, and what application-level enforcement is required that cannot be automatically handled by foreign key constraints alone.

7.19. The Company system contains two independent specializations of EMPLOYEE: the Employment Type specialization (d, t) and the Job Function specialization (d, p). Explain the difference between these two constraint combinations, how each affects the relational schema produced by Strategy B, and why an employee can simultaneously belong to one subclass from each dimension.

7.20. Comparing the base ER relational schema for the Company system (Section 7.6.9) with the complete EER relational schema (Section 7.7.5), two important refinements emerge. Identify these two refinements, explain why each foreign key moved from the EMPLOYEE superclass relation to a specific subclass relation, and discuss what this reveals about the advantage of EER modeling over basic ER modeling for capturing precise semantic relationships.

Exercises

7.1. Relational Model Schema Construction — Library Management System

A city library has the following requirements: "The library manages books, members, and loans. Each book has an ISBN, title, publication year, and number of available copies. Books can have multiple authors. Members have a membership ID, first name, last name, date of birth, email address, and one or more phone numbers. Members borrow books from the

library. Each loan records the loan date, due date, and return date. A loan is processed by exactly one librarian. Librarians have an employee ID, name, and hire date."

a) Identify all entities and classify each as strong or weak. Justify your classification.

b) For each entity, list all attributes and identify their type (simple, composite, multivalued, or derived).

c) Identify all relationships, their cardinality ratios, and participation constraints.

d) Apply the seven-step ER-to-Relational mapping algorithm step by step, showing the result of each step.

e) Write the complete relational model schema using the notation established in this chapter, with primary keys underlined and foreign keys marked with an asterisk.

f) Identify all foreign keys in your schema and state which integrity constraint (entity or referential) each one enforces.

7.2. Weak Entity Identification and Mapping — Hotel Management System

Consider the following Hotel Management System requirements: "A hotel has rooms and guests. Each room has a room number, type (single, double, suite), floor, and daily rate. Guests have a guest ID, name, passport number, email, and phone number. Guests make reservations. Each reservation belongs to exactly one guest and covers one or more rooms. A reservation includes a check-in date, a check-out date, and a total price. Each reservation can include multiple invoice lines, each recording one room, the nightly rate charged, and the number of nights. Invoice lines cannot exist without a reservation."

a) Identify all weak entities in this scenario and explain why each qualifies as a weak entity.

b) For each weak entity, identify its partial key and its identifying owner entity.

c) Apply Step 2 of the mapping algorithm to produce relational schemas for all weak entities. Show the composite primary key formation clearly.

d) Write the complete relational model schema for the entire system.

e) Explain what would happen to the invoice lines if a reservation record were deleted, and describe how referential integrity enforces this dependency.

7.3. Relationship Mapping Analysis — Sports League System

A sports league database has the following requirements: "The league organizes teams, players, and matches. Each team has a team ID, name, city, and founding year. Players have a player ID, first name, last name, date of birth, and position. A player belongs to exactly one team at a time, but a team can have many players. Each team must have at least one player. One player from each team serves as the team captain — not every player is a captain. Matches are played between exactly two teams. Each match has a match ID, date, venue, and final score. Players participate in matches, and for each participation, the system records the player's minutes played, goals scored, and yellow cards received. Some players may not participate in any match, but every match must have participating players."

a) Identify all relationships and classify each as 1:1, 1:N, or M:N.

b) For each relationship, determine the participation constraints (total or partial) for each participating entity and justify your answer.

c) Apply Steps 3, 4, and 5 of the mapping algorithm to each relationship. Show which foreign keys are produced and where they are placed.

d) For the 1:1 captain relationship, show both mapping options and explain which is preferred based on the participation constraints.

e) Write the complete relational model schema and identify any foreign keys that allow NULL values, explaining why.

7.4. Multivalued Attributes and N-ary Relationships — Publishing System

A publishing house manages the following: "The publisher produces books and magazines. Authors write publications. Each author has an author ID, first name, last name, nationality, and one or more email addresses. Each book has an ISBN, title, price, and publication date. A book can be written by multiple authors, and an author can write multiple books. For each book-author combination, the system records the author's contribution role (lead author, co-author, editor). Books are assigned to editors for review. Each assignment records which editor reviewed which book in which month and year, along with the editor's recommendation. Editors are employees with an employee ID, name, and hire date."

a) Identify all multivalued attributes and explain how each is handled during mapping using Step 6.

b) Identify the M:N relationship and the N-ary relationship in this scenario.

c) Apply Step 5 to the M:N relationship, identifying the junction relation and all its attributes, including relationship attributes.

d) Apply Step 7 to the N-ary relationship, forming the composite primary key and identifying all foreign keys.

e) Write the complete relational model schema for the entire system.

7.5. Complete ER-to-Relational Mapping — Medical Clinic System

A medical clinic has the following requirements: "The clinic employs doctors and nurses. Doctors have a doctor ID, first name, last name, specialization, email, and one or more phone numbers. Nurses have a nurse ID, first name, last name, and shift schedule. The clinic is organized into departments such as Cardiology and Pediatrics. Each department has a department ID, name, location, and budget. Each doctor works for exactly one department, and each department has many doctors. Exactly one doctor heads each department, and not all doctors are department heads. Patients visit the clinic. Each patient has a patient ID, first name, last name, date of birth, address (street, city, zip code, country), and one or more phone numbers. Patients schedule appointments with doctors. Each appointment has an appointment number, date, time, and diagnosis notes. An appointment belongs to exactly one patient and is conducted by exactly one doctor. An appointment cannot exist without a patient. During appointments, doctors may prescribe medications. Each prescription records the appointment, medication name, dosage, and duration."

a) Identify all entities and classify each as strong or weak.

b) Identify all composite and multivalued attributes and explain how each is handled during mapping.

c) Apply all seven steps of the mapping algorithm systematically, clearly showing the result of each step.

d) Write the complete relational model schema with all primary keys underlined and all foreign keys marked with an asterisk.

e) Identify which foreign keys allow NULL values and explain why based on participation constraints.

f) Verify your schema by checking that all original entities, relationships, and attributes from the requirements are represented.

7.6. EER Mapping Strategy Selection — Employee Specialization

An organization has the following employee classification requirements: "The organization employs staff who are classified into three categories: Technical Staff (programming languages known, years of technical experience, certification level), Administrative Staff (office location, administrative level, document management responsibility), and Management Staff (budget authority, number of direct reports, strategic planning responsibility). Every employee must belong to exactly one category (disjoint, total specialization). All employees share common attributes: employee ID, first name, last name, email, salary, hire date, and address."

a) Apply all three mapping strategies (Strategy A, Strategy B, and Strategy C) to this specialization hierarchy. For each strategy, write the complete relational schema with all attributes, primary keys, and foreign keys.

b) For Strategy A, identify which attributes will contain NULL values for each employee category and explain why.

c) For Strategy B, explain the dual role of Employee_ID* in each subclass relation.

d) For Strategy C, explain why this strategy is only applicable for disjoint, total (d, t) specializations. Would it be applicable if some employees did not belong to any category?

e) Compare the three strategies using the following criteria: number of relations produced, presence of NULL values, need for joins to retrieve complete employee information, and ease of enforcing the disjoint and total constraints.

f) Which strategy would you recommend for this scenario and why? Justify your answer by referencing Table 7.12 from this chapter.

7.7. EER Mapping with Partial Specialization — Vehicle Fleet System

A vehicle fleet management company has the following requirements: "The company owns vehicles. Each vehicle has a vehicle ID, make, model, year, and license plate. Vehicles are classified by function into specialized categories: Delivery Vehicles (maximum load capacity, cargo dimensions), Passenger Vehicles (seating capacity, accessibility features), and Service Vehicles (equipment type, service area). However, some vehicles in the fleet are general purpose and do not belong to any specialized category. All specialized vehicles are disjoint — no vehicle can belong to more than one category."

a) Identify the constraint combination (d/o, t/p) for this specialization and explain your reasoning.

b) Using Table 7.12 from this chapter, determine which mapping strategies are appropriate for this constraint combination.

c) Explain why Strategy C cannot be used for this specialization. What problem would arise if you tried to apply it?

d) Apply Strategy B to this specialization hierarchy. Write the complete relational schema showing all relations, primary keys, and foreign keys.

e) Show how a general purpose vehicle (one with no specialized function) is represented in Strategy B's schema. How does this differ from how a delivery vehicle is represented?

f) Write two sample tuples — one for a general purpose vehicle and one for a delivery vehicle — showing how Strategy B stores each in the schema.

7.8. Multiple Independent Specializations — Insurance Company

An insurance company has the following requirements: "The company employs agents who are classified independently along two dimensions. By contract type, agents are either Permanent Employees (pension contribution percentage, annual leave days, health insurance plan) or Contract Agents (contract start date, contract end date, agency name). Every agent must have exactly one contract type. By specialization area, agents can be Life Insurance Specialists (life product certifications, minimum coverage knowledge), Property Insurance Specialists (property assessment certification, regional territory), or General Agents (not specialized). Not all agents have a specialization area. Both classifications are independent — a permanent employee can be a life insurance specialist, a property specialist, or a general agent, and the same applies to contract agents."

a) Identify the two independent specialization dimensions, their subclasses, and the constraint combination (d/o, t/p) for each dimension.

b) Explain why these two specializations must be mapped separately rather than as a single hierarchy.

c) For the Contract Type specialization (d, t), apply Strategy B and write the resulting relational schemas.

d) For the Specialization Area specialization (d, p), apply Strategy B and write the resulting relational schemas. Show how a general agent (with no specialization area) is represented.

e) Write the complete integrated relational schema combining both specializations with the base AGENT relation.

f) Using your schema, show how the following two agents would be stored: (i) a permanent employee who is a life insurance specialist, (ii) a contract agent who is a general agent with no specialization area.

7.9. EER Mapping Validation and Error Correction

A student has produced the following relational schema for a School Management System with a PERSON superclass:

PERSON (Person_ID, First_name, Last_name, Birth_date, Email)

STUDENT (Student_ID, Person_ID*, Grade_level, GPA, Enrollment_date)

TEACHER (Teacher_ID, Person_ID*, Subject, Hire_date, Salary)

STUDENT_TEACHER (Student_ID*, Teacher_ID*, Role, Start_date)

The original EER diagram specifies: PERSON is specialized into STUDENT and TEACHER with (d, p) constraints. Some persons are neither students nor teachers. A shared subclass STUDENT_TEACHER exists for persons who are both students and teachers simultaneously (for example, graduate students who also teach).

a) Identify all errors in the schema above. For each error, explain what is wrong and what the correct approach should be.

b) In the subclass relations STUDENT and TEACHER, what should serve as the primary key — Student_ID, Teacher_ID, or Person_ID*? Justify your answer based on the Strategy B conventions established in this chapter.

c) The STUDENT_TEACHER shared subclass should reflect the lattice structure from the EER diagram. Rewrite the STUDENT_TEACHER relation correctly, showing how it links to both parent relations using the appropriate primary and foreign keys.

d) Produce the corrected complete relational schema for this system, applying Strategy B consistently throughout.

e) The (d, p) constraint means some PERSON tuples have no corresponding subclass entry. Show how a sample person who is neither a student nor a teacher would be represented in your corrected schema.

7.10. Comprehensive EER Mapping — Healthcare Specialization System

A healthcare organization has the following EER requirements: "The organization employs medical professionals. All medical professionals share common attributes: employee ID, first name, last name, email, salary, hire date, and address. Medical professionals are classified by role into three categories: Doctors (medical license number, specialization area, years of experience), Nurses (nursing certification level, ward assignment, shift schedule), and Technicians (technical certification, laboratory area). Every medical professional must belong to exactly one role category (disjoint, total). Independently, all medical professionals are classified by employment type: Full-time (annual salary amount, pension contribution, annual leave days) or Part-time (hourly rate, maximum hours per week, flexible schedule indicator). Every medical professional must have exactly one employment type (disjoint, total). These two classifications are independent. Additionally, some senior nurses who have obtained advanced medical degrees can also function as doctors. These Nurse Practitioners inherit from both NURSE and DOCTOR and have additional attributes: prescribing authority (boolean) and supervising doctor ID."

a) Identify all specialization hierarchies, their constraint combinations, and whether they are independent or hierarchical.

b) For each specialization hierarchy, select the most appropriate mapping strategy from Table 7.12 and justify your choice.

c) Apply Strategy B to the Role specialization (Doctor, Nurse, Technician) and write the resulting relational schemas.

d) Apply Strategy B to the Employment Type specialization (Full-time, Part-time) and write the resulting relational schemas.

e) Write the NURSE_PRACTITIONER shared subclass relation, identifying its primary key and all foreign keys referencing the parent relations. Explain the dual role of the primary key in this lattice structure.

f) Write the complete integrated relational schema combining all specializations, including the base relation, both sets of subclass relations, and the shared subclass.

g) Identify any refinements that the EER mapping introduces compared to a basic ER schema of the same system — specifically, which foreign keys should move from the superclass relation into specific subclass relations based on the EER model.

References and Recommended Readings

Primary References

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM, 13*(6), 377–387.

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of Database Systems* (7th ed.). Pearson.

Hristoski, I., & Dimovski, T. (2022). *Dizajn na relaciski bazi na podatoci, Tom 1* [Design of Relational Databases, Volume 1]. University "St. Kliment Ohridski", Bitola. (In Macedonian)

Teorey, T. J., Lightstone, S. S., Nadeau, T., & Jagadish, H. V. (2011). *Database Modeling and Design: Logical Design* (5th ed.). Morgan Kaufmann.

Additional References

Connolly, T. M., & Begg, C. E. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson.

Coronel, C., & Morris, S. (2017). *Database Systems: Design, Implementation, & Management* (12th ed.). Cengage Learning.

Date, C. J. (2003). *An Introduction to Database Systems* (8th ed.). Addison-Wesley.

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database System Concepts* (7th ed.). McGraw-Hill.

Historical and Theoretical Foundations

Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J. N., Griffiths, P. P., King, W. F., III, Lorie, R. A., McJones, P. R., Mehl, J. W., Putzolu, G. R., Traiger, I. L., Wade, B. W., & Watson, V. (1976). System R: Relational approach to database management. *ACM Transactions on Database Systems, 1*(2), 97–137. https://doi.org/10.1145/320455.320457

Codd, E. F. (1990). *The Relational Model for Database Management: Version 2.* Addison-Wesley.

Stonebraker, M., Held, G., Wong, E., & Kreps, P. (1976). The design and implementation of INGRES. *ACM Transactions on Database Systems, 1*(3), 189–222. https://doi.org/10.1145/320473.320476

Teorey, T. J., Yang, D., & Fry, J. P. (1986). A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys, 18*(2), 197–222.

Todd, S. J. P. (1976). The Peterlee relational test vehicle — A system overview. *IBM Systems Journal, 15*(4), 285–308. https://doi.org/10.1147/sj.154.0285

Practical Design Methodologies

Batini, C., Ceri, S., & Navathe, S. B. (1992). *Conceptual Database Design: An Entity-Relationship Approach.* Benjamin/Cummings.

Markowitz, V. M., & Makowsky, J. A. (1990). Identifying extended entity-relationship object structures in relational schemas. *IEEE Transactions on Software Engineering, 16*(8), 777–790.

Simsion, G. C., & Witt, G. C. (2005). *Data Modeling Essentials* (3rd ed.). Morgan Kaufmann.

Contemporary Perspectives

Ambler, S. W., & Holub, A. (2006). *The Object Primer: Agile Model-Driven Development with UML 2.0* (3rd ed.). Cambridge University Press.

DAMA International. (2017). *DAMA-DMBOK: Data Management Body of Knowledge* (2nd ed.). Technics Publications.

Date, C. J. (2019). *Database Design and Relational Theory: Normal Forms and All That Jazz* (2nd ed.). O'Reilly Media.

Chapter 8

8. Relational Algebra

8.1 Introduction to Relational Algebra

Imagine an academic advisor receiving a request to identify the names of all students enrolled in the course "Database Systems". This journey involves merging insights from three relations. First is the COURSE relation to identify the name of the required course. Then the advisor weaves together their academic paths, checking the ENROLLS_IN relation. Finally, the STUDENT entity is used to retrieve the names of the students, showcasing the academic paths of each individual. This represents a common yet powerful academic query: identifying students who are participating in a specific database course.

The data model must not only define the structure and constraints of a database but also provide a set of **operations** for *manipulating the data*. We covered the structures and constraints of the formal relational model in a previous chapter. In this chapter, we focus on the basic set of operations that form the relational algebra (Codd, 1970; Elmasri & Navathe, 2016). These operations are essential tools that enable users to specify retrieval requests and construct complex queries, helping students see their practical importance. This chapter also provides an overview of relational algebra, offering deeper theoretical insights into data manipulation.

The result of any retrieval query in **relational algebra** is a new relation. This enables the algebra operations to produce new relations, which can then be further manipulated using other operations within the same algebra. A sequence of these operations forms a relational algebra expression whose final result is a relation representing the outcome of a database query (Date, 2003; Elmasri & Navathe, 2016).

Relational algebra is crucial because it provides a formal and theoretical foundation for **relational model operations** and, perhaps more importantly, serves as a basis for *implementing and optimizing queries* within the internal modules of relational database systems. Although most commercial systems do not expose relational algebra directly to end users, their query processing and optimization mechanisms are fundamentally based on relational algebraic principles (Codd, 1990; Silberschatz et al., 2019).

8.2 Relational Algebra

Relational algebra is the theoretical foundation of relational databases and query processing, providing a formal mathematical framework for data manipulation. It defines operations such as selection, projection, union, and join that transform relations into new ones. These operations are fundamental to how a DBMS processes queries internally, making them essential for understanding query execution. Learning relational algebra helps in understanding query optimization and connects theoretical concepts with practical implementation, making it vital to modern database systems.

Relational algebra is a *procedural query language* that defines a set of operations—such as selection, projection, union, difference, and join—to systematically process and transform relations. It enables users to specify a sequence of operations to obtain desired results, making it essential for understanding how queries work internally in systems such as DBMSs (Elmasri & Navathe, 2016; Silberschatz et al., 2019). **Relational algebra** is a fundamental component of the relational data model, consisting of operations that are based on *set theory* and *first-order predicate logic* (Codd, 1970; Date, 2003).

Relational algebra operations can be categorized into two main groups. The **first** group includes standard set operations from mathematical *set theory*, such as union, intersection, set difference, and Cartesian product. These operations are applicable because each relation in the model is essentially a set of tuples (Elmasri & Navathe, 2016; Codd, 1990). The **second** group consists of operations specifically designed for the relational model, including selection, projection, join, rename, and division. These operations use predicates and conditions derived from first-order logic to filter, combine, and reshape relational data. The predicate operations represent the complex relationships when working with relational data, including selection, projection, and join. This two-group classification reflects the theoretical origins of relational algebra. A more detailed operational categorization — organized by how each operation functions and the number of relations it requires — is presented in Section 8.3.

Selection and *projection* are **unary** operations that operate on a single relation, serving as essential tools for filtering and shaping data. In contrast, more *complex binary* operations, such as join, allow for the combination of related tuples from two relations based on specific conditions, which is critical for querying normalized relational schemas (Garcia-Molina et al., 2008; Silberschatz et al., 2019). Furthermore, advanced operations, such as *outer joins*, outer *unions*, and aggregate functions (e.g., COUNT, SUM, AVG), have been introduced to accommodate more sophisticated querying requirements (Date, 2019). These enhancements to the original relational algebra support a wide range of real-world database applications and are vital for expressing queries that go beyond basic relational operations.

While relational algebra defines operations for the relational model, relational calculus offers a more abstract, declarative way to specify queries. These are two distinct formal query languages for the relational model: **relational algebra** is procedural, specifying how to retrieve data through a sequence of operations, while **relational calculus** is declarative, specifying what data to retrieve without defining the retrieval steps. This chapter focuses exclusively on relational algebra, exploring its operations, notation, and practical applications in query processing and optimization. Understanding relational algebra is

essential for query optimization, database design, and reasoning about data. Learning relational algebra helps you write complex and efficient queries and provides the foundation for understanding how database systems internally process *SQL queries*. Mastery of relational algebra is fundamental for becoming skilled in database theory and practice. Key learning outcomes include understanding relational algebra operations, mastering query construction using algebraic expressions, and recognizing how these operations form the basis for SQL query processing (Elmasri & Navathe, 2016; Silberschatz et al., 2019). This knowledge can inspire confidence in your ability to design and work with effective database systems.

While SQL syntax is based on tuple relational calculus, database systems internally process SQL queries using relational algebra (Garcia-Molina et al., 2008). The significance of relational algebra goes beyond its mathematical elegance to its real-world impact on modern database systems. It provides the conceptual and operational foundation for how database management systems process queries, giving designers and developers a clear vocabulary for data manipulation. Its procedural nature makes it especially valuable for query optimization. SQL queries are converted into relational algebra expressions, and transformation rules are applied to generate efficient execution plans. This formalism enables systematic reasoning about query equivalence, performance, and optimization strategies. Relational algebra operations can be combined to create complex queries and enhance optimal system performance. Understanding this foundation can boost your confidence in working with database systems and their optimization processes.

The Relational Algebra Foundational Characteristics

Formal Foundation: Relational algebra provides a mathematical and theoretical basis for all operations on a relational database. It ensures that queries are well-defined and predictable.

Procedural Language: It is a procedural language, meaning it specifies the step-by-step process for retrieving data. You explicitly tell the system how to get the result.

Query Optimization: Although users don't typically write queries in relational algebra, it is the language that database systems use internally to optimize query performance. The database engine translates a high-level query (like SQL) into a sequence of relational algebra operations and then optimizes that sequence for efficiency.

Set-Based Operations: Relational algebra operations are fundamentally based on mathematical set theory. This means the results of any operation are always a new set of tuples, and properties like commutativity and associativity apply to many operations.

Relational algebra serves as the procedural language that outlines how a database system handles a query. When an academic advisor wishes to retrieve a list of all students enrolled in the "Database Systems" course, they phrase their request using a high-level language like SQL. The database's internal query optimizer, in turn, converts this request into a detailed, sequential set of relational algebra operations. The optimizer might start the task by filtering the COURSE table to identify the particular database course. Next, it would

employ a series of join operations to connect the ID of that course to the ENROLLS_IN table, and subsequently to the STUDENT table. Finally, a projection operation would select only the names of the students for the final result. This approach ensures that the complex task of merging data from multiple tables is executed efficiently and logically, highlighting the importance of relational algebra in optimizing modern database queries.

Learning Approach: Our examples throughout this chapter utilize comprehensive **University Management System** and **Company/Business Database** models that demonstrate the **concepts of relational algebra.** This hands-on approach will bridge the gap between abstract theory and practical application in query processing and optimization mechanisms.

8.3 Relational Algebra Operations

Relational algebra is a procedural query language that acts as the foundational language for the relational model, providing a formal framework for manipulating and querying data stored in relational databases. Each operation in relational algebra takes one or more relations as input and produces a new relation as output, allowing operations to be composed into complex query expressions.

Relational algebra **operations** are broadly divided into *three* distinct *categories*.

The **basic** operations form the core foundation with essential operators like selection (σ), projection (π), and set operations, providing the minimum complete set sufficient to express any relational query.

Derived operations build upon these fundamentals to offer more convenient and intuitive ways to express common database operations like joins and intersections.

Extended operations expand the algebra's practical utility by adding capabilities like aggregation, grouping, and sorting that are essential for real-world database applications.

This organizational structure reflects both the mathematical elegance of the relational algebra and the practical needs of database query processing, where basic operations provide theoretical completeness while derived and extended operations enhance expressiveness and usability.

These operations can be understood from two complementary perspectives. The **theoretical perspective** classifies operations as basic (minimal complete set), derived (built from basic operations for convenience), and extended (practical additions). The **operational perspective**, which we adopt for organizing this chapter, categorizes operations by how they function: unary operations on single relations, set operations on compatible pairs, binary operations joining relations, and aggregate operations for summarization. Table 8.1 illustrates how these two perspectives align, showing that both frameworks describe the same operations from different analytical angles.

Table 8.1: Mapping Theoretical and Operational Perspectives

Theoretical Category	***Operational Categories***	***Operations***
Basic Operations	Unary Operations	σ (Selection), π (Projection), ρ (Rename)

Theoretical Category	*Operational Categories*	*Operations*
Derived Operations	Set Operations	∪ (Union), − (Difference), × (Cartesian Product)
	Binary Operations	⋈ (Join), ÷ (Division)
	Set Operations	∩ (Intersection)
Extended Operations	Aggregate Operations	γ (Grouping), φ (Aggregates), τ (Sorting)
	Unary Operations	δ (Distinct)

The following sections organize operations from an operational perspective, as it provides a clearer framework for understanding how each operation transforms relations and how they can be combined to answer complex queries.

8.3.1 Categories of Operations

Relational algebra operations are systematically organized into **four primary categories** based on their functionality and the number of relations they operate upon.

Unary operations act on a single relation to produce a new relation. These fundamental operations include selection (σ) for filtering tuples based on conditions, projection (π) for selecting specific attributes, rename (ρ) for changing relation or attribute names, and distinct (δ) for eliminating duplicate tuples. Unary operations provide the essential tools for filtering and reshaping individual relations.

Set operations are derived from mathematical set theory and operate on two union-compatible relations—relations that have the same number of attributes with compatible data types. These operations include union (∪) for combining tuples from two relations, intersection (∩) for finding common tuples, and set difference (−) for identifying tuples present in one relation but not another. The Cartesian product (×), while also a set operation, does not require union compatibility and produces all possible combinations of tuples from two relations.

Binary operations combine related data from two relations based on specific conditions. The join operation (⋈) and its variants form the basis of multi-table queries, enabling the grouping of tuples from different relations based on common attributes or specified conditions. The division operation (÷) addresses specialized queries that require finding tuples in one relation associated with all tuples in another relation.

Aggregate and extended operations enhance the algebra's practical capabilities beyond its theoretical foundations. Grouping operations (γ) partition tuples based on shared attribute values, while aggregate functions (φ) such as COUNT, SUM, AVG, MIN, and MAX summarize data within groups or across entire relations. Sorting operations (τ) order results based on specified attributes and criteria. These extensions are vital for expressing complex analytical queries and generating summary information from large datasets.

This categorization provides a logical framework for understanding how relational algebra operations progress from simple single-relation manipulations to sophisticated multi-relation queries with computational results.

Table 8.2 provides a comprehensive reference for all relational algebra operations covered in this chapter. Each operation will be explained in detail in subsequent sections.

Table 8.2: Summary of the Notation for Relational Algebra

Symbol	Operator Name	Function
σ (sigma)	Selection	*Selects tuples (rows) from a relation that satisfy a given condition.*
π (pi)	Projection	*Projects specific attributes (columns) from a relation.*
∪	Union	*Combines tuples from two relations, eliminating duplicates.*
∩	Intersection	*Returns tuples that are present in both of two relations.*
-	Set Difference	*Returns tuples that are in one relation but not in another.*
×	Cartesian Product	*Creates a new relation by combining all possible combinations of tuples from two relations.*
⋈	Natural Join	*Combines rows from two tables based on columns with the same name and data type, and eliminates duplicate columns.*
÷	Division	*Finds tuples in one relation that are associated with all tuples in another relation.*
ρ (rho)	Rename	*Renames a relation or its attributes to make queries easier to write.*
δ (delta)	Distinct operation	*Distinct operation eliminates duplicate tuples from a relation, ensuring that the result contains only unique rows.*
γ (gamma)	Grouping operation	*Grouping operations enable the partitioning of tuples into groups based on shared attribute values.*
φ (phi)	Aggregate functions	*Applies aggregate functions to a relation, potentially with grouping.*
τ (tau)	Sorting operation	*Sorting operations for ordering the output data based on specified attributes and criteria.*
ω (omega)	NULL value	*Represents values of attributes that may be unknown or may not apply to a tuple.*

8.3.2 Operation Notation

In relational algebra, the conditional notation for filtering and shaping data, derived from propositional logic with quantifiers and predicates, is written as a subscript (lower) following established mathematical conventions where operators use subscripts for parameters or conditions, similar to summation notation. This positioning creates a clear visual hierarchy that distinguishes the main operator (σ or π) from its modifying condition, while separating parameters from operands—the subscript contains the Boolean expression that controls the operation, while parentheses contain the relation being operated upon. This notation

prevents ambiguity by clearly distinguishing which part is the condition versus the relation, and maintains compositional clarity in complex nested expressions.

Relational algebra notation is systematically organized into distinct groups based on functionality and theoretical importance.

Unary Operations (on a single relation)

Unary Relational Operations work on a single relation to produce a new one. The selection (σ) operation filters tuples (rows) based on a condition, while the projection (π) operation selects a subset of attributes (columns). The rename (ρ) operation gives a new name to a relation or its attributes.

Selection (σ): Filters a relation horizontally, selecting a subset of tuples that satisfy a given condition.

- **Notation**: $\boldsymbol{\sigma_{condition}(R)}$

Projection (π): Filters a relation vertically, selecting a subset of attributes (columns) from a relation.

- **Notation**: $\boldsymbol{\pi_{attributes}(R)}$

Rename (ρ): Gives a new name to a relation or its attributes.

- **Notation**: $\boldsymbol{\rho_{S(attributes)}(R)}$

Distinct (**δ**): Eliminates duplicate tuples from a relation.

- **Notation**: **δ(R)**

Set Operations (on two union-compatible relations)

Relational algebra operations from set theory are based on mathematical set theory, treating relations as sets of tuples (Codd, 1970; Elmasri & Navathe, 2016; Date, 2003). This includes the union (∪), intersection (∩), and difference/minus (-) operations, which combine or compare two union-compatible relations, and the Cartesian product (×), which pairs every tuple from one relation with every tuple from another.

Union (∪): Combines all unique tuples from two relations.

- **Notation**: **R∪S**

Intersection (∩): Returns tuples that exist in both relations.

- **Notation**: **R∩S**

Difference/Minus (-): Returns tuples that are in the first relation but not in the second.

- **Notation**: **R-S**

Cartesian Product (×): Combines every tuple of the first relation with every tuple of the second. The result is a new relation with all attributes from both relations.

- **Notation**: **R×S**

Binary Operations (on two relations)

Binary Relational Operations join two relations. These are the join and division operations, which are essential for combining related data from different tables.

Join (⋈): Combines tuples from two relations based on a common join condition. There are various types of joins, but the most common is Equijoin, which uses an equality condition.

- **Notation**: $R \bowtie_{join_condition} S$

Division (÷): Used to find all tuples in one relation that are related to all tuples in another.

- **Notation**: $R \div S$

Aggregate Operations

Aggregate Operations apply aggregate functions to a relation, potentially with grouping. These operations are used to summarize data and include functions like COUNT, SUM, AVG, MIN, and MAX. They are powerful for generating summary information from large datasets, such as the total number of students or the average salary of employees.

Grouping operation (γ): Grouping operations enable the partitioning of tuples into groups based on shared attribute values, typically to support aggregate computations across these groups.

- **Notation**: $\gamma_{grouping_attributes}(R)$

Aggregate functions (φ): Apply aggregate functions to a relation, potentially with grouping. Typical aggregate functions include COUNT, SUM, AVG, MIN, and MAX.

- **Notation**: $\varphi_{aggregate_function_list}(R)$

Sorting operation (τ): Sorting operations allow for ordering the output data based on specified attributes and criteria. Sorting arranges tuples in a relation according to one or more attributes in ascending or descending order.

- **Notation**: $\tau_{sorting_attributes}(R)$

Logical Operators Within Selection Conditions

Logical operators in relational algebra provide the fundamental building blocks for constructing complex selection conditions and for filtering data within database queries (Elmasri & Navathe, 2016; Silberschatz et al., 2019). These operators are not standalone relational algebra operations; rather, they are Boolean combinators used within selection predicates. In relational algebra, logical operators AND, OR, and NOT are intrinsically connected to the selection operation through their foundation in propositional logic and first-order predicate logic. Selection conditions are expressed as logical formulas where these operators combine predicates to specify tuple selection criteria. The AND operator conjoins multiple predicates, OR provides alternatives, and NOT negates conditions. This logical foundation allows students to feel confident in constructing complex query expressions that precisely define result sets through Boolean combinations of predicates, enabling sophisticated data retrieval specifications.

Logical AND operator (∧): Combines two or more conditions and evaluates to true only when all constituent conditions are simultaneously satisfied, enabling precise filtering criteria such as selecting employees whose salary exceeds a specified threshold and whose department matches a given value (Date, 2003).

- **Notation**: ∧

Logical OR operator (∨): Represents inclusive disjunction and evaluates to true when at least one of the specified conditions holds, supporting queries such as retrieving products whose category matches a given value or whose price is below a defined amount (Elmasri & Navathe, 2016).

- **Notation**: ∨

Logical NOT operator (~): Negates a condition, returning tuples that do not satisfy the specified predicate, such as selecting customers whose city does not equal a particular location (Codd, 1970; Garcia-Molina et al., 2008).

- **Notation**: ~

These logical operators enable the construction of sophisticated selection conditions that form the foundation for complex data retrieval in relational algebra.

8.4 Unary Relational Operations

Unary relational operations are a fundamental concept in relational algebra, the theoretical foundation of relational database systems (Elmasri & Navathe, 2016). These operations are called "unary" because they operate on a single relation (table) at a time to produce a new relation. The main unary operations covered in this chapter are selection, projection, rename, and distinct. The selection operation acts as a horizontal filter, choosing a subset of tuples (rows) from a relation based on a specific condition. The projection operation acts as a vertical filter, selecting a subset of attributes (columns) from a relation, discarding the rest. This allows us to focus on a specific, relevant slice of the data. These operations are essential for querying and manipulating data effectively.

8.4.1 The Selection Operation

The **selection** operation, represented by the symbol **σ**, is a fundamental *unary* relational operation used to choose a subset of tuples from a relation that satisfies a specific selection condition. We can consider the **selection** operation to be a filter that keeps only those tuples (rows) that satisfy a qualifying condition. The operation can also be visualized as a horizontal partition of the relation into two sets of tuples—those tuples, interpreted as values that satisfy the predicate or condition and are **selected**, and those tuples that do not satisfy the predicate or condition and are *filtered out.* The result of a selection operation is a new relation that is a subset of the original relation. The schema of the resulting relation is identical to the schema of the original relation. The selection condition can be a simple comparison (e.g., Budget > 1000000) or a more complex logical expression involving multiple attributes and operators. For example, if we need to identify students older than 21 that are from Ohrid, the selection would be on two conditions (Age > 21) and (City = 'Ohrid').

This operation is the first step in many data retrieval queries, allowing us to narrow down our focus to the specific data we need to analyze.

> **The SELECTION Operation**
>
> $\sigma_{predicate}(R)$ **-** The **Selection** operation works on a relation R and defines a relation that contains only the tuples of R that satisfy specific condition (*predicate*).

Throughout this chapter, we present numbered relational algebra requirements (**req8.1**, **req8.2**, etc.) that demonstrate practical applications of each operation. Each query requirement is followed by its relational algebra expression and results.

University System Example

req8.1: *List all faculties with a budget greater than 1,000,000.*

Relational algebra expression using the selection operation:

$$\sigma_{Budget>1000000}(FACULTY)$$

Explanation of each component in the selection operation:

- **σ (sigma)**: This symbol represents the selection operation, which is used to filter rows from a relation.
- **Budget>1000000**: This is the selection condition (or predicate). It specifies the criteria for a row to be included in the result. In this case, it's looking for rows where the value in the Budget attribute is greater than 1,000,000.
- **FACULTY**: This is the relation (or table) on which the operation is performed.

In simple terms, the expression reads as: *select all tuples from the FACULTY relation where the Budget is greater than 1,000,000*. The result is a new relation containing only those specific rows that satisfy the condition, as shown in Table 8.3.

Table 8.3: Results of the selection operation for the FACULTY relation data

Faculty_ID	*Name*	*Budget*	*Location*
F001	Computer Science	1500000	Main Campus
F002	Electrical Engineering	1200000	East Wing
F004	Bioengineering	1100000	North Building

req8.2: *List all students from Ohrid who are older than 21.*

Relational algebra expression using the selection operation:

$$\sigma_{Age>21 \wedge City = 'Ohrid'}(STUDENT)$$

This uses the selection operation (σ) with a compound Boolean condition that combines two criteria using the logical AND operator (∧). The condition "Age > 21 ∧ City = 'Ohrid'" ensures that only students who satisfy both requirements are returned, as they must be older than 21 AND their city must be Ohrid. All attributes of the matching student records are included in the result, as shown in Table 8.4.

Table 8.4: Results of the selection operation on the STUDENT relation

Student_ID	*First_name*	*Last_name*	*Birth_date*	*Age*	*Street*	*City*	*Country*	*Zip_code*	*Faculty_ID*
S001	Sarah	Smith	25.08.1995	30	Jane Sandanski 32	Ohrid	Macedonia	6000	F001
S005	Elena	Mitan	01.07.2003	22	Turisticka 22	Ohrid	Macedonia	6000	F005

Company System Example

req8.3: *List all departments with a budget greater than 300,000.*

Relational algebra expression using the selection operation:

$$\sigma_{\text{Budget>300000}}(\textbf{DEPARTMENT})$$

Explanation of each component in the selection operation:

- **σ (sigma)**: This symbol represents the selection operation.
- **Budget>300000**: This is the selection condition (or predicate). It specifies that only tuples (rows) where the 'Budget' attribute has a value greater than 300,000 should be included in the result.
- **DEPARTMENT**: This is the relation (or table) on which the selection operation is performed.

This expression will return all rows from the DEPARTMENT table where the Budget is greater than 300,000, as shown in Table 8.5.

Table 8.5: Results of the selection operation on the DEPARTMENT relation

Department_ID	*Dept_name*	*Location*	*Budget*
D1	Sales	Skopje	500000

req8.4: *List all warehouses with a capacity of less than or equal to 5,000, located in Ohrid.*

Relational algebra expression using the selection operation:

$$\sigma_{\text{Capacity <= 5000 } \wedge \text{ Location = 'Ohrid'}}(\textbf{WAREHOUSE})$$

This uses the selection operation (σ) with a compound Boolean condition combining two criteria using the logical AND operator (∧). The condition "Capacity <= 5000 ∧ Location = 'Ohrid'" ensures that only warehouses satisfying both requirements are returned - they must have a capacity less than or equal to 5,000 AND be located in Ohrid. All attributes of the matching warehouse records are included in the result, as shown in Table 8.6.

Table 8.6: Results of the selection operation on the WAREHOUSE relation

Warehouse_ID	*Location*	*Capacity*	*Department_ID*
W3	*Ohrid*	*5000*	*D3*

8.4.2 The Projection Operation

The **projection** operation, represented by the symbol **π**, is a unary relational operation that selects certain attributes (columns) from a table while discarding others. If we are interested in only a subset of the attributes of a relation, we use the projection operation to create a new relation that contains only those attributes. The result of the projection operation can be visualized as a vertical partition of the original relation into two sets of attributes: one has the needed columns and forms the result of the operation, and the other contains the discarded columns. Unlike the selection operation, which reduces the number of rows, the projection operation reduces the number of columns. The degree of the resulting relation is equal to the number of attributes specified in the projection list, and if the projection on a set of attributes results in duplicate tuples, these duplicates are eliminated, as relations, by definition, cannot contain duplicate tuples. This operation is essential for simplifying the view of data, allowing us to focus on only the most relevant information for a given query.

The Projection Operation

$\pi_{attributes}(R)$ **-** The projection operation works on relation R and defines a relation that contains a vertical subset of R, selecting only the specified attributes.

University System Example

req8.5: *List the professor ID, first name, last name, academic title, hire date, and Faculty ID from the PROFESSOR table.*

Relational algebra expression using the **projection** operation:

$$\pi_{Professor_ID,\ First_name,\ Last_name,\ Academic_title,\ Hire_date,\ Faculty_ID}(PROFESSOR)$$

Explanation of each component in the projection operation:

- **π (pi)**: This symbol represents the projection operation.
- ***Professor_ID, First_name, Last_name, Academic_title, Hire_date, Faculty_ID***: These are the attributes (columns) that will be included in the result. All other attributes from the PROFESSOR relation will be excluded.
- **PROFESSOR**: This is the relation on which the projection operation is performed.

This expression will return a new relation containing only the specified columns from the PROFESSOR table, as shown in Table 8.7.

Table 8.7: Results of the **projection** operation on the PROFESSOR relation

Professor_ID	*First_name*	*Last_name*	*Academic_title*	*Hire_date*	*Faculty_ID*
P001	Kliment	Ohridski	Full Professor	01.08.2007	F001
P002	Naum	Ohridski	Associate Professor	15.01.2004	F002
P003	Kiril	Prosvetitel	Full Professor	25.09.2001	F004
P004	Metodij	Prosvetitel	Full Professor	13.09.2002	F001
P005	Blaze	Koneski	Assistant Professor	19.10.2012	F005

P006	Richard	Johnson	Associate Professor	02.09.2022	F001

Company System Example

req8.6: *List the customers by first name, last name, city and join date.*

Relational algebra expression using the projection operation:

$$\pi_{\text{First_name,Last_name,City,Join_date}}(\textbf{CUSTOMER})$$

Explanation of each component in the projection operation:

- **π (pi)**: This symbol represents the projection operation.
- **First_name, Last_name, City, Join_date**: These are the attributes that will be included in the result. All other attributes will be excluded.
- **CUSTOMER**: This is the relation on which the projection operation is performed.

This expression will return all rows from the CUSTOMER table, but only with the First_name, Last_name, City, and Join_date columns, as shown in Table 8.8.

Table 8.8: Results of the **projection** operation on the CUSTOMER relation

First_name	*Last_name*	*City*	*Join_date*
Ana	Ivanovska	Skopje	10.01.2022
Petar	Petrovski	Bitola	15.03.2022
Marija	Georgievska	Ohrid	20.05.2022
Nikola	Nikolovski	Prilep	25.07.2022
Elena	Trajkovska	Tetovo	30.09.2022
Goran	Stojanovski	Kumanovo	05.01.2023
Ivana	Jovanovska	Veles	12.03.2023

8.4.3 The Rename Operation

The **rename** operation, represented by the symbol ρ, is a unary relational operation that allows changing the name of a relation and optionally renaming its attributes, providing flexibility in query composition and result presentation. This operation is particularly useful for several purposes: combining relations with similar schemas but different names, resolving attribute name conflicts in joins, creating meaningful names for intermediate results in complex queries, and preparing relations for union operations that require compatible attribute names.

> The **rename** operation has several notation forms:
>
> - $\rho_{S}(R)$ - renames relation R to S while keeping all attribute names unchanged,
> - $\rho_{S(\text{attributes})}(R)$ - renames the relation to S and simultaneously renames all attributes to the specified list, and
> - $\rho_{\text{attributes}}(R)$ - keeps the same relation name but renames only the attributes.

University System Example

req8.7: *Create ACADEMIC_PROFESSOR relation with attributes Acc_first_name, Acc_last_name, and Academic_Title from the existing PROFESSOR relation.*

Relational algebra expression using the rename operation:

$$\rho_{ACADEMIC_PROFESSOR(Acc_first_name,\ Acc_last_name,\ Academic_Title)}(\pi_{First_name,\ Last_name,\ Academic_title}(PROFESSOR))$$

Explanation of each component in the rename operation:

- **ρ (rho):** This symbol represents the rename operation.
- **ACADEMIC_PROFESSOR (Acc_first_name, Acc_last_name, Academic_Title):** This creates a new relation named "ACADEMIC_PROFESSOR" and renames its three attributes to Acc_first_name, Acc_last_name, and Academic_Title respectively.
- **$\pi_{First_name,\ Last_name,\ Academic_title}$ (PROFESSOR):** This is the inner projection that first selects only the three relevant attributes from the PROFESSOR relation, discarding the remaining six (Professor_ID, Office, Email, Hire_date, Faculty_ID, Heads_Faculty_ID). The rename is then applied to this projected result.

This expression will return all rows from the PROFESSOR table, now in a new relation named **ACADEMIC_PROFESSOR** with the new column names Acc_first_name, Acc_last_name, and Academic_Title, as shown in Table 8.9.

Table 8.9: Results of the **rename** operation on the PROFESSOR relation

ACADEMIC_PROFESSOR

Acc_first_name	*Acc_last_name*	*Academic_Title*
Kliment	Ohridski	Full Professor
Naum	Ohridski	Associate Professor
Kiril	Prosvetitel	Full Professor
Metodij	Prosvetitel	Full Professor
Blaze	Koneski	Assistant Professor
Richard	Johnson	Associate Professor

Company System Example

req8.8: *Create **SELLING_ITEM** relation with attributes item name, item description, category, price, and weight from the existing PRODUCT relation.*

Relational algebra expression using the rename operation:

$$\rho_{SELLING_ITEM(Item_name,\ Item_description,\ Category,\ Price,\ Weight)}(\pi_{Product_name,\ Description,\ Category,\ Price,\ Weight}(PRODUCT))$$

Explanation of each component in the rename operation:

- **ρ (rho):** This symbol represents the rename operation.

- **SELLING_ITEM (Item_name, Item_description, Category, Price, Weight):** This creates a new relation named "SELLING_ITEM" and renames its five attributes to Item_name, Item_description, Category, Price, and Weight, respectively.
- $\pi_{\text{Product_name, Description, Category, Price, Weight}}$ **(PRODUCT):** This is the inner projection that first selects only the five relevant attributes from the PRODUCT relation, discarding Product_ID. The rename is then applied to this projected result.

This expression will return all rows from the PRODUCT table, now in a new relation named **SELLING_ITEM** with the column names Item_name, Item_description, Category, Price, and Weight, as shown in Table 8.10.

Table 8.10: Results of the **rename** operation on the PRODUCT relation

SELLING_ITEM

Item_name	*Item_description*	*Category*	*Price*	*Weight*
Ajvar	Traditional Macedonian spread	Food	372	0.5
Tavce Gravce	Baked beans dish	Food	434	0.7
Rakija	Fruit brandy	Beverage	682	1
Mastika	Mastic-flavored liqueur	Beverage	930	1
Sirenje	White cheese	Dairy	465	0.4
Kajmak	Creamy dairy product	Dairy	620	0.3
Ohrid Pearl	Jewelry	Accessories	9300	0.1
Macedonian Ruby	Gemstone	Accessories	31000	0.05
Filigree	Traditional craft	Souvenir	4650	0.2
Galichnik Wedding Costume	Traditional clothing	Apparel	74400	2.5
Vranec Wine	Red wine	Beverage	1116	1.5
Selsko Meso	Village-style meat dish	Food	868	0.8

8.4.4 The Distinct Operation

The distinct operation eliminates duplicate tuples from a relation, ensuring that the result contains only unique rows. While not always explicitly denoted with a special symbol in traditional relational algebra notation, it is an implicit part of relational algebra since relations are mathematically defined as sets, which inherently contain no duplicates. The distinct operation is particularly important when dealing with projections that may create duplicate tuples, as they might produce repeated rows when projecting onto a subset of attributes that doesn't uniquely identify each tuple. For example, the *ENROLLS_IN relation* has separate tuples for each course enrollment, so a student who enrolls in multiple courses appears multiple times. When projecting only the Student_ID attribute, *using the distinct operation* would ensure that each *student appears only once in the result*.

The Distinct Operation

***δ*(R)** - The distinct operation ensures that results contain unique rows, even if the relation R contains duplicate tuples.

University System Example

req8.9: *List all unique Student_IDs from the ENROLLS_IN relation.*

To find all unique Student_IDs from the ENROLLS_IN relation, we use the delta (δ) operation. Relational algebra expression using the distinct operation:

$$\delta(\pi_{\text{Student_ID}}(\text{ENROLLS_IN}))$$

This expression first projects the Student_ID column from the ENROLLS_IN relation and then applies the distinct operation to ensure only unique student IDs are in the result.

As shown in Table 8.11, the result shows a single entry for each student who is enrolled in at least one course, effectively providing a unique list of all enrolled students.

Table 8.11: Results of the distinct operation on the ENROLLS_IN relation

Student_ID
S001
S002
S003
S004
S005
S006

Company System Example

req8.10: *Find the unique status values of all customer orders.*

Relational algebra expression using the distinct operation:

$$\delta(\pi_{\text{Status}}(\text{ORDER}))$$

This expression first projects the Status column from the ORDER relation, then applies a DISTINCT operation to return only unique status codes.

As shown in Table 8.12, there is a single entry for each unique status code in the order purchase requests.

Table 8.12: Results of the distinct operation on the ORDER relation

Status
Shipped
Pending
Processing
Canceled

In practical database systems, the distinct operation corresponds directly to SQL's DISTINCT keyword and is essential for queries requiring unique values, statistical analysis, and data deduplication. The distinct operation maintains the relational model's mathematical foundation by ensuring that query results remain proper relations (sets) rather than multi-sets (bags), though many commercial database systems actually work with multi-sets for performance reasons and require explicit distinct specification to eliminate duplicates.

8.5 Set Operations

Set operations in relational algebra are fundamental operations that combine tuples from two or more relations to produce a new relation. These operations are analogous to mathematical set theory. Three of these operations, Union, Intersection, and Difference, require the relations to be **union-compatible**, meaning they must have the same number of attributes and corresponding attributes must have the same data types (Elmasri & Navathe, 2016). The Cartesian Product does not require union compatibility. The four primary set operations are: **Union (∪)**, which combines all unique tuples from two relations; **Intersection (∩)**, which returns tuples common to both relations; **Difference/Minus (−)**, which returns tuples present in the first relation but not the second; and **Cartesian Product (×)**, which pairs every tuple from the first relation with every tuple from the second, creating a relation with attributes from both.

8.5.1 The Union Operation

The **union** operation is a fundamental set operation in relational algebra that combines all tuples from two relations to produce a new relation containing every unique tuple from both. The operation requires the two relations to be **union-compatible** - they must have the same number of attributes with corresponding attributes having compatible domains. This compatibility ensures that combined tuples maintain a consistent structure, preventing data integrity issues. The result is a single relation containing all unique tuples from the two source relations, with duplicates automatically eliminated.

> **The Union Operation**
>
> **R ∪ S** - The union operation of two relations R and S returns all tuples that appear in R, in S, or in both. Duplicate tuples are automatically eliminated.

University System Example

req8.11: *Find all Faculty IDs for faculties that are either defined in the FACULTY relation or have an affiliated professor in the PROFESSOR relation.*

Relational algebra expression using the union operation:

$$\pi_{\text{Faculty_ID}}(\textbf{FACULTY}) \cup \pi_{\text{Faculty_ID}}(\textbf{PROFESSOR})$$

Explanation of each component in the union operation:

- $\pi_{\text{Faculty_ID}}(\textbf{FACULTY})$: This projects the FACULTY relation to get a list of all unique Faculty IDs.

- $\pi_{Faculty_ID}$**(PROFESSOR)**: This projects the PROFESSOR relation to get a list of all unique Faculty IDs that have an affiliated professor.
- **Union (∪)**: This operator combines the two lists, eliminating duplicates.

Since the PROFESSOR relation has a foreign key to the FACULTY relation, all Faculty_IDs in the PROFESSOR relation must also exist in the FACULTY relation. Therefore, in this specific case, the union will return the same result as simply projecting on the FACULTY relation alone, as shown in Table 8.13. This example demonstrates the union operation's ability to combine results, though the referential integrity constraint makes the combination redundant here.

Table 8.13: Results of the union operation on the FACULTY and PROFESSOR relations.

Faculty_ID
F001
F002
F003
F004
F005

req8.12: *List all people (students and professors) with their First_name and Last_name.*

The relational algebra expression must ensure that the relations we are uniting are union-compatible. This means they must have the same number of attributes and corresponding attributes must have compatible data types. Step-by-step approach:

1. **Projection on STUDENT**: Project First_name and Last_name from the STUDENT relation. **StudentsInfo=**$\pi_{First_name,Last_name}$**(STUDENT)**
2. **Projection on PROFESSOR**: Project First_name and Last_name from the PROFESSOR relation. **ProfessorsInfo=**$\pi_{First_name,Last_name}$**(PROFESSOR)**
3. **Union**: Combine StudentsInfo and ProfessorsInfo relations, eliminating duplicates.

StudentsInfo ∪ ProfessorsInfo

Complete relational algebra expression:

$\pi_{First_name,Last_name}$**(STUDENT)** ∪ $\pi_{First_name,Last_name}$**(PROFESSOR)**

The results in Table 8.14 show all unique individuals from both relations.

Table 8.14: Results of the union operation on the STUDENT and PROFESSOR relations

First_name	***Last_name***
Anica	*Stojnasen*
Blaze	*Koneski*
Bojan	*Jovansen*
David	*Boskovski*
Elena	*Mitan*
Kiril	*Prosvetitel*

Kliment	*Ohridski*
Kosta	*Dojcinoski*
Metodij	*Prosvetitel*
Naum	*Ohridski*
Richard	*Johnson*
Sarah	*Smith*

Note that the order shown is for presentation purposes only; relational algebra operations do not guarantee any specific ordering unless explicitly specified using a sorting operation

Company System Example

req8.13: *List all cities where there are employees or customers who live.*

Relational algebra expression using the union operation:

$$\pi_{City}(\text{EMPLOYEE}) \cup \pi_{City}(\text{CUSTOMER})$$

Explanation of each component in the union operation:

- π_{City}**(EMPLOYEE):** This projects only the City attribute from the EMPLOYEE relation, giving us a list of all cities where employees are located.
- π_{City}**(CUSTOMER)**: This projects only the City attribute from the CUSTOMER relation, giving us a list of all cities where customers are located.
- **Union (∪)**: This operator combines the results of the two projections, including all unique cities found in either the employee cities list or the customer cities list. Duplicate cities are automatically removed by the union operation, as shown in Table 8.15.

Table 8.15. Results of the union operation on the EMPLOYEE and CUSTOMER relations.

City
Skopje
Bitola
Ohrid
Prilep
Tetovo
Kumanovo
Veles

8.5.2 The Intersection Operation

The **intersection** operation produces a relation containing all tuples that exist in both relations R and S. This set theory operation requires that R and S be union-compatible, meaning they must have the same number of attributes with corresponding attributes having compatible domains. The resulting relation contains only those tuples that appear identically in both input relations.

The Intersection Operation

R ∩ S - The intersection operation returns all tuples that exist in both R and S. The relations must be union-compatible.

The intersection operation is particularly useful for identifying commonalities between datasets. For example, it can find students enrolled in both Computer Science and Mathematics courses, products available in multiple warehouses, or customers who have made purchases in both online and retail channels.

University System Example

req8.14: *Find the ISBNs of all textbooks that have been assigned to a course*.

The relational algebra expression finds the intersection of all ISBNs in the TEXTBOOK relation and all ISBNs in the ASSIGNMENT relation. The projections π_{ISBN}(TEXTBOOK) and π_{ISBN}(ASSIGNMENT) are union-compatible since both contain only the ISBN attribute:

$$\pi_{ISBN}(\text{TEXTBOOK}) \cap \pi_{ISBN}(\text{ASSIGNMENT})$$

Explanation of each component in the intersection operation:

- **π_{ISBN}(TEXTBOOK):** This projects the TEXTBOOK relation to get a list of every ISBN available in the library.
- **π_{ISBN}(ASSIGNMENT):** This projects the ASSIGNMENT relation to get a list of every ISBN that has been assigned to a course.
- **Intersection (∩):** This operator returns only the ISBNs that exist in both of the projected lists, effectively giving us the list of textbooks that have been assigned to courses.

The results in Table 8.16 list all ISBNs that appear in both the TEXTBOOK and ASSIGNMENT relations, indicating which textbooks have been assigned to courses.

Table 8.16: Results of the intersection operation on the TEXTBOOK and ASSIGNMENT relations.

ISBN
978-0134093413
978-2134093493
978-0834080185
978-0131498642
978-6321286413
978-0471754471
978-0571754171
978-0134080185
978-7134082853

978-0321286433
978-7821286423

Company System Example

req8.15: *List all cities where there are both customers and employees.*

Relational algebra expression using the intersection operation:

π_{City}**(EMPLOYEE)** $\cap$ π_{City}**(CUSTOMER)**

Explanation of each component in the intersection operation:

- π_{City}**(EMPLOYEE)**: This projects only the City attribute from the EMPLOYEE relation.
- π_{City}**(CUSTOMER)**: This projects only the City attribute from the CUSTOMER relation.
- **Intersection (∩):** This operator returns all unique cities that are common to both the set of employee cities and the set of customer cities.

The results in Table 8.17 show cities where both employees and customers are located.

Table 8.17: Results of the intersection operation on the EMPLOYEE and CUSTOMER relations.

City
Skopje
Bitola
Ohrid
Prilep
Tetovo

8.5.3 The Set Difference Operation

The **set difference** operation, also known as **minus (–)**, is a binary operation that takes two relations, R and S, and returns a new relation containing all tuples that are in R but not in S. Like union and intersection, this operation requires the two relations to be **union-compatible**. The operation is not commutative, meaning $R - S$ is not the same as $S - R$. This operation is useful for queries such as finding students who have not enrolled in any courses, products not yet sold, or employees without assigned projects.

> **The Set Difference Operation**
>
> **R − S** - The set difference operation returns all tuples that are in relation R but not in relation S.

University System Example

req8.16: *Find faculties that have no affiliated professors.*

The relational algebra expression uses the FACULTY and PROFESSOR relations:

Step-by-step approach:

1. **Projection on FACULTY**: Project the Faculty_ID attribute from the FACULTY relation
   $F = \pi_{Faculty_ID}(FACULTY)$
2. **Projection on PROFESSOR**: Project the Faculty_ID attribute from the PROFESSOR relation, giving us the Faculty_IDs that have at least one associated professor.
   $P = \pi_{Faculty_ID}(PROFESSOR)$
3. **Set Difference**: Perform the set difference (F − P) to find Faculty_IDs that exist in FACULTY but not in PROFESSOR.
   $R = F - P$

Complete relational algebra expression:

$$\pi_{Faculty_ID}(FACULTY) - \pi_{Faculty_ID}(PROFESSOR)$$

As shown in Table 8.18, F003 (Economy & Tourism) is the only faculty that does not have any professors associated with it in the PROFESSOR relation.

Table 8.18: Results of the University example using ***Set Difference (or Minus)*** operation.

Faculty_ID
F003

Company System Example

req8.17: *List all cities where there are customers but not employees*.

Relational algebra expression using the set difference operation:

Step-by-step approach:

1. **Projection on CUSTOMER**: Project the City attribute from the CUSTOMER relation, giving us all cities where customers are located.
   $C = \pi_{City}(CUSTOMER)$
2. **Projection on EMPLOYEE**: Project the City attribute from the EMPLOYEE relation, giving us all cities where employees are located.
   $E = \pi_{City}(EMPLOYEE)$
3. **Set Difference**: Perform the set difference (C − E) to find cities that have customers but not employees.
   $R = C - E$

Complete relational algebra expression:

$$\pi_{City}(CUSTOMER) - \pi_{City}(EMPLOYEE)$$

As shown in Table 8.19, the cities Kumanovo and Veles have customers but no employees.

Table 8.19: Results of the set difference operation on the CUSTOMER and EMPLOYEE relations.

City
Kumanovo
Veles

8.5.4 The Cartesian Product Operation

The Cartesian product operation combines two relations to produce a new relation consisting of all possible combinations of tuples from the two input relations (Codd, 1970; Date, 2003). If one relation R has n tuples and p attributes, and the other relation S has m tuples and q attributes, the Cartesian product will contain (n × m) tuples with (p + q) attributes. Although powerful, the Cartesian product often produces very large intermediate results and is therefore typically used together with selection operations to restrict the output to meaningful tuple combinations (Elmasri & Navathe, 2016).

> **The Cartesian product Operation**
>
> **R × S** - The Cartesian product operation defines a relation that is the concatenation of every tuple of relation R with every tuple of relation S.

The two relations may have attributes with the same name. In this case, the attribute names are prefixed with the relation name to maintain the uniqueness of attribute names in the resulting relation.

University System Example

req8.18: *Find all of the FACULTY Name and Location with affiliated PROFESSOR first name, last name, and hire date, using the Cartesian product operation.*

Relational algebra expression using the Cartesian product operation on projected relations.

Step-by-step approach:

1. **Projection on FACULTY**: Project only the Name and Location attributes from the FACULTY relation.
   $$\mathbf{F = \pi_{Name,\ Location}(FACULTY)}$$
2. **Projection on PROFESSOR**: Project only the First_name, Last_name, and Hire_date attributes from the PROFESSOR relation.
   $$\mathbf{P = \pi_{First_name,\ Last_name,\ Hire_date}(PROFESSOR)}$$
3. **Cartesian Product**: Perform the Cartesian product between F and P, combining every row from F with every row from P.
   $$\mathbf{R = F \times P}$$

Complete relational algebra expression:

$$\mathbf{(\pi_{Name,\ Location}(FACULTY)) \times (\pi_{First_name,\ Last_name,\ Hire_date}(PROFESSOR))}$$

The Cartesian product combines every tuple from one relation with every tuple from another relation. If relation F has n tuples and relation P has m tuples, their Cartesian product F × P will result in n × m tuples. As shown in Table 8.20, since the FACULTY projection has 5 rows and the PROFESSOR projection has 6 rows, the resulting table contains 5 × 6 = 30

rows. Each row contains a combination of a faculty's name and location with a professor's information.

Important: This Cartesian product creates all possible combinations, including many meaningless pairings of faculties with professors who may not actually be affiliated with those specific faculties. In practice, this operation would be combined with a selection condition (or a JOIN operation, covered in the next section) to filter only the relevant, meaningful combinations.

Table 8.20: Results of the Cartesian product operation on FACULTY and PROFESSOR relations.

Faculty Name	*Location*	*Professor First_name*	*Professor Last_name*	*Hire_date*
Computer Science	Main Campus	Kliment	Ohridski	01.08.2007
Computer Science	Main Campus	Naum	Ohridski	15.01.2004
Computer Science	Main Campus	Kiril	Prosvetitel	25.09.2001
Computer Science	Main Campus	Metodij	Prosvetitel	13.09.2002
Computer Science	Main Campus	Blaze	Koneski	19.10.2012
Computer Science	Main Campus	Richard	Johnson	02.09.2022
Electrical Engineering	East Wing	Kliment	Ohridski	01.08.2007
Electrical Engineering	East Wing	Naum	Ohridski	15.01.2004
Electrical Engineering	East Wing	Kiril	Prosvetitel	25.09.2001
Electrical Engineering	East Wing	Metodij	Prosvetitel	13.09.2002
Electrical Engineering	East Wing	Blaze	Koneski	19.10.2012
Electrical Engineering	East Wing	Richard	Johnson	02.09.2022
Economy & Tourism	West Wing	Kliment	Ohridski	01.08.2007
Economy & Tourism	West Wing	Naum	Ohridski	15.01.2004
Economy & Tourism	West Wing	Kiril	Prosvetitel	25.09.2001
Economy & Tourism	West Wing	Metodij	Prosvetitel	13.09.2002
Economy & Tourism	West Wing	Blaze	Koneski	19.10.2012
Economy & Tourism	West Wing	Richard	Johnson	02.09.2022
Bioengineering	North Building	Kliment	Ohridski	01.08.2007
Bioengineering	North Building	Naum	Ohridski	15.01.2004
Bioengineering	North Building	Kiril	Prosvetitel	25.09.2001
Bioengineering	North Building	Metodij	Prosvetitel	13.09.2002
Bioengineering	North Building	Blaze	Koneski	19.10.2012
Bioengineering	North Building	Richard	Johnson	02.09.2022
Humanities	South Building	Kliment	Ohridski	01.08.2007

Humanities	South Building	Naum	Ohridski	15.01.2004
Humanities	South Building	Kiril	Prosvetitel	25.09.2001
Humanities	South Building	Metodij	Prosvetitel	13.09.2002
Humanities	South Building	Blaze	Koneski	19.10.2012
Humanities	South Building	Richard	Johnson	02.09.2022

Company System Example

req8.19: *List all of the employees' first name, last name, city where they work, department name, and location, using the Cartesian product operation.*

The relational algebra expression for the Cartesian product of the projected EMPLOYEE and DEPARTMENT relations is:

$$(\pi_{\text{First_name, Last_name, City}}(\textbf{EMPLOYEE})) \times (\pi_{\text{Dept_name, Location}}(\textbf{DEPARTMENT}))$$

This operation creates a new relation containing every possible combination of an employee tuple with a department tuple (Table 8.21).

Table 8.21: Results of the Cartesian product operation on the EMPLOYEE and DEPARTMENT relations.

First_name	*Last_name*	*Employee. City*	*Dept_name*	*Department. Location*
Zoran	Pejovski	Skopje	Sales	Skopje
Zoran	Pejovski	Skopje	Marketing	Bitola
Zoran	Pejovski	Skopje	Human Resources	Ohrid
Nikola	Delcevski	Skopje	Sales	Skopje
Nikola	Delcevski	Skopje	Marketing	Bitola
Nikola	Delcevski	Skopje	Human Resources	Ohrid
Marija	Naumoska	Bitola	Sales	Skopje
Marija	Naumoska	Bitola	Marketing	Bitola
Marija	Naumoska	Bitola	Human Resources	Ohrid
Anita	Toshevska	Bitola	Sales	Skopje
Anita	Toshevska	Bitola	Marketing	Bitola
Anita	Toshevska	Bitola	Human Resources	Ohrid
Naum	Stojanoski	Ohrid	Sales	Skopje
Naum	Stojanoski	Ohrid	Marketing	Bitola
Naum	Stojanoski	Ohrid	Human Resources	Ohrid
Hristina	Miloseska	Ohrid	Sales	Skopje
Hristina	Miloseska	Ohrid	Marketing	Bitola
Hristina	Miloseska	Ohrid	Human Resources	Ohrid

Dimitar	Tocko	Prilep	Sales	Skopje
Dimitar	Tocko	Prilep	Marketing	Bitola
Dimitar	Tocko	Prilep	Human Resources	Ohrid
Biljana	Osmani	Tetovo	Sales	Skopje
Biljana	Osmani	Tetovo	Marketing	Bitola
Biljana	Osmani	Tetovo	Human Resources	Ohrid

As shown in Table 8.21, since the EMPLOYEE projection has 8 rows and the DEPARTMENT projection has 3 rows, the resulting table contains 8 × 3 = 24 rows. Each row contains a combination of an employee's information with a department's information, without any regard for whether that employee actually works in that specific department. This creates many meaningless combinations.

8.6 Binary Relational Operations

Binary relational operations in relational algebra combine two relations to produce a new relation, forming the foundation for complex database queries (Codd, 1970; Elmasri & Navathe, 2016). The most prominent and frequently used binary operation is the join (⋈) operation, which combines tuples from two relations based on a common attribute or set of attributes (Silberschatz et al., 2019). The natural join, a specific type of join, merges relations by matching tuples with identical values in their common attributes, thereby eliminating duplicate columns in the resulting relation (Elmasri & Navathe, 2016; Date, 2003). For example, joining a PROFESSOR relation with a FACULTY relation on a faculty identifier attribute produces a new relation that integrates faculty information with corresponding professors' details. The theta join generalizes this concept by allowing arbitrary comparison predicates between attributes of the two relations, using operators such as equality, inequality, greater-than, and less-than, thereby providing greater flexibility in how relations are combined (Garcia-Molina et al., 2008; Silberschatz et al., 2019).

The division (÷) operation, although less frequently encountered, addresses a specific class of queries involving universal quantification. Division takes two relations and produces a relation containing those tuples from the first relation that are related to all tuples in the second relation (Date, 2003; Elmasri & Navathe, 2016). For instance, to identify students enrolled in all required courses, the division operation returns students whose enrollment records include all courses listed in the ENROLLS_IN relation. This operation is especially useful for queries involving conditions such as "all," "every," or "complete set of," even though it can often be expressed through combinations of other relational algebra operations (Codd, 1990). Collectively, these binary operations enable the formulation of complex data-retrieval requirements and serve as the theoretical backbone of SQL query languages and relational database management systems (Silberschatz et al., 2019; Garcia-Molina et al., 2008).

8.6.1 The Natural Join Operation

Joins are compound operators that combine tuples from two or more relations based on related attributes, internally implemented through a sequence of cross product, selection, and sometimes projection operations. The natural join is the most fundamental and commonly used join type, often referred to simply as "join," denoted by the symbol ⋈. A natural join (R ⋈ S) automatically combines relations R and S by matching tuples that have identical values for all attributes with the same name, eliminating duplicate columns in the result.

> **The JOIN Operation**
>
> **R⋈S** The **JOIN** operation produces all combinations of tuples from relations R and S.

A **Natural Join (⋈)** is a compound relational algebra operator that combines a Cartesian product, a selection, and a projection. It works by:

1. It conceptually starts with the **Cartesian product** of the two relations.
2. **Selection -** It then automatically selects rows where the values of common attributes are equal. This implicitly creates the join condition.
3. **Projection** - of all unique attributes from both relations.

Joins are compound operators involving cross product, selection, and (sometimes) projection. The relational algebra operator for natural join can be written as:

$$R \bowtie S \equiv \pi_{attributes}(\sigma_{R.A_1=S.A_1 \wedge R.A_2=S.A_2 \wedge \ldots}(R \times S)),$$

where the cross product creates all possible tuple combinations, selection filters for matching common attributes, and projection removes duplicate columns.

University System Example

req8.20: *List all of the PROFESSOR (Professor_ID, First_name, Last_name, Hire_date and Faculty_ID) using a "natural join" relational algebra with FACULTY (Name, Location, Budget).*

The Relational Algebra expression for the natural join is as follows:

PROFESSOR⋈FACULTY

The result of this operation is a new relation (table) that includes all the columns from both the PROFESSOR and FACULTY tables, but only for the rows where PROFESSOR.Faculty_ID matches FACULTY.Faculty_ID (Table 8.22).

Table 8.22: Results of the ***natural join*** operation for the PROFESSOR and FACULTY relations.

Professor_ID	*First_name*	*Last_name*	*Academic_title*	*Hire_date*	*Faculty_ID*	*Heads_Faculty_ID*	*Name*	*Budget*	*Location*
P001	Kliment	Ohridski	Full Professor	01.08.2007	F001	NULL	Computer Science	1500000	Main Campus

P002	Naum	Ohridski	Associate Professor	15.01.2004	F002	F002	Electrical Engineering	1200000	East Wing
P003	Kiril	Prosvetitel	Full Professor	25.09.2001	F004	NULL	Bioengineering	1100000	North Building
P004	Metodij	Prosvetitel	Full Professor	13.09.2002	F001	F001	Computer Science	1500000	Main Campus
P005	Blaze	Koneski	Assistant Professor	19.10.2012	F005	F005	Humanities	800000	South Building
P006	Richard	Johnson	Associate Professor	02.09.2022	F001	NULL	Computer Science	1500000	Main Campus

Note: The expression PROFESSOR⋈FACULTY returns all attributes from both relations, producing a 12-column result that also includes the Office and Email attributes from the PROFESSOR relation. These two columns are omitted from Table 8.22 for display purposes only; in a full implementation, they would appear between the Last_name and Hire_date columns.

Notice in Table 8.22, each row now contains the complete information for a professor and their corresponding faculty. This is a much more useful result than a Cartesian product, which would have created many meaningless combinations.

🏢 *Company System Example*

req8.21: *List all the employees' first name, last name, city where they work, department name, and location.*

The relational algebra expression for a Natural Join between EMPLOYEE and DEPARTMENT to list the employee First_name, Last_name, City, and their department's Dept_name and Location is:

$$\pi_{\text{First_name, Last_name, City, Dept_name, Location}}(\text{EMPLOYEE} \bowtie \text{DEPARTMENT})$$

The result of this operation is a new relation, shown in Table 8.23, that includes employee First_name, Last_name, City from EMPLOYEE relation, and Dept_name, Location from DEPARTMENT relation, but only for the rows where EMPLOYEE.Department_ID matches DEPARTMENT.Department_ID.

Table 8.23: Results of the ***natural join*** operation for the EMPLOYEE and DEPARTMENT relations.

First_name	*Last_name*	*Employee. City*	*Dept_name*	*Department. Location*
Zoran	Pejovski	Skopje	Sales	Skopje
Nikola	Delcevski	Skopje	Sales	Skopje
Marija	Naumoska	Bitola	Marketing	Bitola
Anita	Toshevska	Bitola	Marketing	Bitola
Naum	Stojanoski	Ohrid	Human Resources	Ohrid

Hristina	Miloseska	Ohrid	Human Resources	Ohrid
Dimitar	Tocko	Prilep	Sales	Skopje
Biljana	Osmani	Tetovo	Marketing	Bitola

8.6.2 The Equijoin Operation

The **equijoin** operation produces all combinations of tuples from relations R and S that satisfy a join condition containing only equality comparisons between attributes. Unlike natural join, which automatically matches on common attribute names, equijoin allows explicit specification of which attributes to compare using equality operators (=). The join condition typically takes the form R.A = S.B, where A and B are attributes from relations R and S, respectively, and multiple equality conditions can be combined using logical AND (∧). Unlike natural join, equijoin preserves all attributes from both relations, including any duplicate columns, making it useful when you need to maintain all original attribute names or when joining on attributes with different names. This operation forms the foundation for most practical database joins and directly corresponds to SQL's JOIN...ON syntax with equality conditions.

> **The Equijoin Operation**
>
> **$R \bowtie_{<join_condition>} S$** The **equijoin** operation produces all combinations of tuples from relations R and S, that satisfy a join condition with only equality comparisons.

University System Example

req8.22: *List the STUDENTS (first name, last name) and the courses (title) they have been enrolled.*

An equijoin is a type of join operation in relational algebra that combines rows from two relations (tables) based on an equality condition between specified attributes. In your provided example, the operation is a chain of equijoins. It's a valid way to connect *multiple* tables to retrieve the desired information.

Relational algebra expression is using the equijoin operation that joins the STUDENT, ENROLLS_IN, and COURSE relations to get all the data in one large relation, as follows:

$$(\mathbf{STUDENT} \bowtie_{\mathbf{Student_ID=ENROLLS_IN.Student_ID}} \mathbf{ENROLLS_IN}) \bowtie_{\mathbf{Course_ID=ENROLLS_IN.Course_ID}} \mathbf{COURSE}$$

Description with explanation of what each part means in the equijoin operation:

- **Equijoin ($\bowtie_{<join_condition>}$): Relation S**, that joins the STUDENT and ENROLLS_IN tables on the common attribute Student_ID.

  $$\mathbf{S = STUDENT} \bowtie_{\mathbf{Student_ID=ENROLLS_IN.Student_ID}} \mathbf{ENROLLS_IN}$$

- **Equijoin ($\bowtie_{<join_condition>}$): Relation C**, that joins the result of the first join, relation S, with the COURSE table on the common attribute Course_ID.

  $$\mathbf{C = S} \bowtie_{\text{Course_ID=ENROLLS_IN.Course_ID}} \text{COURSE}$$

- **Projection (π):** This selects, in relation **R**, from the resulting joined relation, we project the specified attributes: First_name, Last_name (from the STUDENT relation), and Title (from the COURSE relation).

$$R = \pi_{First_name,\ Last_name,\ Title}(C)$$

The complete relational algebra expression is:

$$\pi_{First_name,\ Last_name,\ Title}((STUDENT \bowtie_{Student_ID=ENROLLS_IN.Student_ID} ENROLLS_IN) \bowtie_{Course_ID=ENROLLS_IN.Course_ID} COURSE)$$

This expression joins the three tables and then projects (selects) only the First_name and Last_name from the STUDENT table and the Title from the COURSE table. Note in Table 8.24, the same students' First_name, Last_name repeat for each different course Title.

Table 8.24: Results of the *equijoin join* operation for the STUDENT, ENROLLS_IN, and COURSE relations.

First_name	*Last_name*	*Title*
Sarah	Smith	Database Systems
Bojan	Jovansen	Circuit Analysis
Bojan	Jovansen	Advanced Circuit Analysis
Kosta	Dojcinoski	Linear Algebra
Kosta	Dojcinoski	Matrix theory and advanced vector spaces
David	Boskovski	Physics
David	Boskovski	Advanced Physics
Elena	Mitan	Academic writing
Elena	Mitan	Advanced Academic Writing
Anica	Stojnasen	Intro to Programming
Anica	Stojnasen	Advanced Programming
Anica	Stojnasen	Database Systems

Company System Example

req8.23: *List the products by warehouse location and stored in quantity on hand.*

To list the products (Product_name) by warehouse location and quantity, we start by matching the WAREHOUSE relation location and quantity on hand, from the STORED_IN relation. Next, this relation needs to perform a join with the PRODUCT relation to find the products.

This query uses two JOIN operations to link the three tables. It first joins PRODUCT and STORED_IN on Product_ID, and then joins the result with WAREHOUSE on Warehouse_ID. A projection operation then retrieves the desired attributes. The results are grouped by warehouse location for better readability.

Relational algebra expression using equijoin operation between the PRODUCT, STORED_IN, and WAREHOUSE relations is:

$$\pi_{P.Product_name,\ W.Location,\ SI.Quantity_on_hand}(PRODUCT \bowtie_{P.Product_ID=SI.Product_ID}(WAREHOUSE \bowtie_{W.Warehouse_ID=SI.Warehouse_ID} STORED_IN))$$

This expression first Equijoin the WAREHOUSE relation with the STORED_IN relation on the common attribute Warehouse_ID. The result is then Equijoin with the PRODUCT relation on the common attribute Product_ID. Finally, as shown in Table 8.25, it projects the requested attributes: Product_name, Location, and Quantity_on_hand.

Table 8.25: Results of the equijoin join operation for the PRODUCT, WAREHOUSE and STORED_IN relations.

Product_name	*Location*	*Quantity_on_hand*
Tavce Gravce	Skopje	21
Ajvar	Skopje	57
Ohrid Pearl	Skopje	9
Sirenje	Skopje	14
Selsko Meso	Skopje	18
Macedonian Ruby	Bitola	23
Mastika	Bitola	32
Vranec Wine	Bitola	72
Galichnik Wedding Costume	Tetovo	7
Filigree	Ohrid	6
Kajmak	Ohrid	12
Rakija	Ohrid	18

8.6.3 The Theta Join Operation

The Theta join operation is a generalized join operation, denoted as $R \bowtie_\theta S$, where θ (theta) represents any comparison condition, not limited to equality. The theta join produces all combinations of tuples from relations R and S that satisfy the specified theta condition, which can include any comparison operators such as =, ≠, <, >, ≤, ≥, or complex Boolean expressions combining multiple conditions with logical operators (∧, ∨, ~).

A Conditional join, also known as a Theta join, is a fundamental relational algebra operation that combines tuples from two relations based on a specified condition and can be expressed as:

$$R \bowtie_\theta S \equiv \sigma_\theta(R \times S)$$

Conceptually, a theta-join works in two steps:

Cartesian Product: It first computes the Cartesian product of the two relations.

Selection: Then, it applies a selection operation on the result of the Cartesian product, keeping only those tuples that satisfy the specified condition (θ).

Unlike equijoin, which only uses *equality* comparisons, Theta join provides maximum flexibility for expressing complex relationship conditions between relations. This makes it particularly useful for range queries, inequality comparisons, and complex analytical operations such as finding employees earning more than the average salary, or matching records within certain date ranges. Theta join serves as the most general form of join operation, with equijoin and natural join being special cases of theta join.

> **The Theta Join Operation**
>
> **R⋈θS** The **Theta join** operation produces all combinations of tuples from relations R and S, that satisfy the *specific join condition.*

University System Example

req8.24: *List all professors with faculty details on professors who were hired before 01 January 2005, and whose faculty's budget is greater than 1,000,000.*

In this case, we want to list combinations of professors and *faculties* where:

The professor's *Hire_date* is before '2005-01-01'.

The faculty's *Budget* is greater than 1,000,000.

Relational algebra expression using Theta join is:

$$\pi_{\text{P.Professor_ID, P.First_name, P.Last_name, P.Hire_date, F.Name, F.Location, F.Budget}} (\text{PROFESSOR} \bowtie_{\text{PROFESSOR.Faculty_ID = FACULTY.Faculty_ID} \wedge \text{PROFESSOR.Hire_date < '2005-01-01'} \wedge \text{FACULTY.Budget > 1000000}} \text{FACULTY})$$

This expression performs a Conditional Join (Theta-Join) between the PROFESSOR (aliased as P) and FACULTY (aliased as F) relations.

Theta-Join ($\bowtie_\theta$) step-by-step operations:

The *join condition (θ)* is a compound one:

Professor.Faculty_ID = Faculty.Faculty_ID: This ensures that we are linking professors to their actual affiliated faculty.

Professor.Hire_date < '2005-01-01': This filters professors who were hired before January 1, 2005.

Faculty.Budget > 1000000: This filters faculties that have a budget greater than 1,000,000.

The *join condition* combines tuples from PROFESSOR and FACULTY only if all three parts of this condition are met.

Projection (π): From the resulting joined relation, we project the specified attributes: *Professor_ID, First_name, Last_name, Hire_date* (from the PROFESSOR relation), and *Name, Location, Budget* (from the FACULTY relation).

Finally, as shown in Table 8.26, it lists all professors and faculties where the professor was hired before 01 January 2005, and the faculty's budget is greater than 1,000,000.

Table 8.26: Results of the **Theta join** operation for the PROFESSOR and FACULTY relations.

Professor_ID	*First_name*	*Last_name*	*Hire_date*	*Name*	*Location*	*Budget*
P002	Naum	Ohridski	15.01.2004	Electrical Engineering	East Wing	1200000
P003	Kiril	Prosvetitel	25.09.2001	Bioengineering	North Building	1100000
P004	Metodij	Prosvetitel	13.09.2002	Computer Science	Main Campus	1500000

Company System Example

req8.25: *List all of the employees with a hire date after 01 June 2021 that are working in a department with a budget of less than 400,000.*

The relational algebra expression that performs a Conditional Join (Theta-Join) between the EMPLOYEE and DEPARTMENT relations is:

$$\pi_{\text{First_name, Last_name, Hire_date, Dept_name, Budget}} \textbf{(EMPLOYEE} \bowtie_{\text{EMPLOYEE.Department_ID=DEPARTMENT.Department_ID} \ \wedge \ \text{EMPLOYEE.Hire_date>'2021-06-01'} \ \wedge \ \text{DEPARTMENT.Budget<400000}} \textbf{DEPARTMENT)}$$

Theta-Join ($\bowtie_\theta$) step-by-step operations:

The *join condition (θ)* is a compound one:

EMPLOYEE.Department_ID=DEPARTMENT.Department_ID: This forms an initial combination of every employee record with every department record.

EMPLOYEE.Hire_date>'2021−06−01': This filters for employees hired after June 1st, 2021.

DEPARTMENT.Budget < 400000: This filters for departments with a budget less than 400,000.

The combination of these conditions forms the "theta" condition for the join.

Projection (π): From the resulting joined relation, we project the specified attributes: First_name, Last_name, Hire_date, Dept_name, and Budget.

Finally, as shown in Table 8.27, it lists all of the employees with a hire date after 01 June 2021 who are working in a department with a budget of less than 400,000.

Table 8.27: Results of the **Theta join** operation for the EMPLOYEE and DEPARTMENT relations.

First_name	*Last_name*	*Hire_date*	*Dept_name*	*Budget*
Marija	Naumoska	10.05.2022	Marketing	300000

Anita	Toshevska	22.07.2023	Marketing	300000
Hristina	Miloseska	12.11.2021	Human Resources	200000
Biljana	Osmani	18.04.2023	Marketing	300000

8.6.4 The Outer Join Operation

Outer Joins

The main advantage of the Outer Join is that information is preserved; it means Outer Join preserves tuples that would have been lost by the other types of joins.

Left Outer Join (⟕): Include unmatched tuples from the left relation.

Right Outer Join (⟖): Include unmatched tuples from the right relation

Full Outer Join (⟗): Include unmatched tuples from both relations

> **R⟕S** The **Left Outer Join** is a join in which tuples from the relation R that do not have matching values in the common attributes of S, are also included in the result relation. Missing values in the second relation are set to null.

> **R⟖S** The **Right Outer Join** is a join in which tuples from the relation S that do not have matching values in the common attributes of R, are also included in the result relation. Missing values in the first relation are set to null.

> **R⟗S** The **Full Outer Join** is a join in which tuples keeps all tuples both relations R and S, padding tuples with nulls when no matching tuples are found.

University System Example

Left Outer Join (⟕) operation for the PROFESSOR and FACULTY relations.

req8.26: *List all professors and faculties, including those without a faculty affiliation.*

A Left Outer Join on PROFESSOR and FACULTY would show all professors, including those without a faculty affiliation (if any exist), along with their faculty details.

The relational algebra expression is

$$\textbf{PROFESSOR} ⟕_{\text{Professor.Faculty_ID = Faculty.Faculty_ID}} \textbf{FACULTY}$$

A Left Outer Join (⟕) between PROFESSOR and FACULTY will return all tuples from the PROFESSOR relation. If a professor does not have a matching Faculty_ID in the FACULTY relation, the faculty-related columns in the result will be filled with NULL values.

Table 8.28 includes all professors, even those not affiliated with a faculty (if any exist in the data).

Table 8.28: Results of the Left Outer Join operation for the PROFESSOR and FACULTY relations.

Professor_ID	*First_name*	*Last_name*	*Academic_title*	*Faculty_Name*
P001	Kliment	Ohridski	Full Professor	Computer Science
P002	Naum	Ohridski	Associate Professor	Electrical Engineering
P003	Kiril	Prosvetitel	Full Professor	Bioengineering
P004	Metodij	Prosvetitel	Full Professor	Computer Science
P005	Blaze	Koneski	Assistant Professor	Humanities
P006	Richard	Johnson	Associate Professor	Computer Science

Right Outer Join (⟖) operation for the PROFESSOR and FACULTY relations.

req8.27: *List all professors and faculties, including faculties without affiliated professors.*

The relational algebra expression is:

$$\textbf{PROFESSOR} \rtimes_{\text{PROFESSOR.Faculty_ID=FACULTY.Faculty_ID}} \textbf{FACULTY}$$

A *Right Outer Join* (⟖) between PROFESSOR and FACULTY will return **all tuples from the FACULTY relation** (the right side), along with matching professor data where it exists. If a faculty has no affiliated professors, the **professor-related columns** (Professor_ID, First_name, Last_name, Academic_title) in the result will be filled with NULL values.

Table 8.29: Results of the Right Outer Join operation for the PROFESSOR and FACULTY relations.

Professor_ID	*First_name*	*Last_name*	*Academic_title*	*Faculty_Name*
P001	Kliment	Ohridski	Full Professor	Computer Science
P002	Naum	Ohridski	Associate Professor	Electrical Engineering
P003	Kiril	Prosvetitel	Full Professor	Bioengineering
P004	Metodij	Prosvetitel	Full Professor	Computer Science
P005	Blaze	Koneski	Assistant Professor	Humanities
P006	Richard	Johnson	Associate Professor	Computer Science
NULL	NULL	NULL	NULL	Economy & Tourism

Full Outer Join (⟗) operation for the PROFESSOR and FACULTY relations.

req8.28: *List all professors and faculties, including faculties with and without affiliated professors.*

The relational algebra expression is:

$$\textbf{PROFESSOR} \rtimes_{\text{PROFESSOR.Faculty_ID=FACULTY.Faculty_ID}} \textbf{FACULTY}$$

A Full Outer Join (⟗) between PROFESSOR and FACULTY will return all tuples from both relations, combining matched tuples and including unmatched tuples from either side, as shown in Table 8.30.

Table 8.30: Results of the Full Outer Join operation for the PROFESSOR and FACULTY relations.

Professor_ID	*First_name*	*Last_name*	*Academic_title*	*Faculty_Name*
P001	Kliment	Ohridski	Full Professor	Computer Science
P002	Naum	Ohridski	Associate Professor	Electrical Engineering
P003	Kiril	Prosvetitel	Full Professor	Bioengineering
P004	Metodij	Prosvetitel	Full Professor	Computer Science
P005	Blaze	Koneski	Assistant Professor	Humanities
P006	Richard	Johnson	Associate Professor	Computer Science
NULL	NULL	NULL	NULL	Economy & Tourism

Company System Example

Left Outer Join (⟕) operation for the EMPLOYEE and WAREHOUSE relations.

req8.29: *List all employees who are handling the warehouses, considering employees without a warehouse*

The relational algebra expression for the Left Outer Join operation is:

$$\textbf{EMPLOYEE} ⟕_{\text{EMPLOYEE.Warehouse_ID=WAREHOUSE.Warehouse_ID}} \textbf{WAREHOUSE}$$

This expression performs a LEFT Outer join (⟕) on the EMPLOYEE and WAREHOUSE tables. The join condition is that the Warehouse_ID from the EMPLOYEE table must match the Warehouse_ID from the WAREHOUSE table. This ensures that all employees from the left table (EMPLOYEE) are included in the result set, with matching warehouse information from the right table (WAREHOUSE) where available. For employees without a matching warehouse, the warehouse attributes will be null.

The query will produce a resulting Table 8.31, which includes all employees from the EMPLOYEE table. The columns Warehouse_ID, Location, and Capacity from the WAREHOUSE table will be displayed as NULL for any employee who isn't associated with a warehouse.

Table 8.31: Results of the Left Outer Join operation for the EMPLOYEE and WAREHOUSE relations.

Employee_ID	*First_name*	*Last_name*	*Warehouse_ID*	*Location*	*Capacity*
E1	Zoran	Pejovski	W1	Skopje	10000
E2	Nikola	Delcevski	W1	Skopje	10000
E3	Marija	Naumoska	W2	Bitola	7500
E4	Anita	Toshevska	W2	Bitola	7500

E5	Naum	Stojanoski	W3	Ohrid	5000
E6	Hristina	Miloseska	W3	Ohrid	5000
E7	Dimitar	Tocko	W1	Skopje	10000
E8	Biljana	Osmani	W2	Bitola	7500

Right Outer Join (⟖) operation for the EMPLOYEE and WAREHOUSE relations.

req8.30: *List all employees who are handling the warehouses, including warehouses without employees.*

The relational algebra expression is:

$$\textbf{EMPLOYEE} ⟖_{\text{EMPLOYEE.Warehouse_ID=WAREHOUSE.Warehouse_ID}} \textbf{WAREHOUSE}$$

This expression performs a Right Outer join (represented by the symbol '⟖') on the EMPLOYEE and WAREHOUSE tables. The join condition is that the Warehouse_ID from the EMPLOYEE table must match the Warehouse_ID from the WAREHOUSE table. This operation includes all tuples from the right table (WAREHOUSE), and for each of these, it joins with matching tuples from the left table (EMPLOYEE). If a warehouse has no corresponding employee, the employee attributes in the result will be null. (Table 8.32)

Table 8.32: Results of the Right Outer Join operation for the EMPLOYEE and WAREHOUSE relations.

Employee_ID	***First_name***	***Last_name***	***Warehouse_ID***	***Location***	***Capacity***
E1	Zoran	Pejovski	W1	Skopje	10000
E2	Nikola	Delcevski	W1	Skopje	10000
E3	Marija	Naumoska	W2	Bitola	7500
E4	Anita	Toshevska	W2	Bitola	7500
E5	Naum	Stojanoski	W3	Ohrid	5000
E6	Hristina	Miloseska	W3	Ohrid	5000
E7	Dimitar	Tocko	W1	Skopje	10000
E8	Biljana	Osmani	W2	Bitola	7500
NULL	NULL	NULL	W4	Tetovo	1000

Full Outer Join (⟗) operation for the EMPLOYEE and WAREHOUSE relations.

req8.31: *List all employees who are handling the warehouses, consider employees without a warehouse, and warehouses without employees.*

The relational algebra expression is:

$$\textbf{EMPLOYEE} ⟗_{\text{EMPLOYEE.Warehouse_ID=WAREHOUSE.Warehouse_ID}} \textbf{WAREHOUSE}$$

This expression performs a FULL Outer join (⟗) on the EMPLOYEE and WAREHOUSE tables. This join type includes all tuples from both the left table (EMPLOYEE) and the right

table (WAREHOUSE). When a match is found based on the join condition (EMPLOYEE.Warehouse_ID = WAREHOUSE.Warehouse_ID), the attributes from both tables are combined. For tuples that do not have a match in the other table, their corresponding attributes in the result set will be filled with null values. (Table 8.33)

Table 8.33: Results of the FULL Outer Join operation for the EMPLOYEE and WAREHOUSE relations.

Employee_ID	*First_name*	*Last_name*	*Warehouse_ID*	*Location*	*Capacity*
E1	Zoran	Pejovski	W1	Skopje	10000
E2	Nikola	Delcevski	W1	Skopje	10000
E3	Marija	Naumoska	W2	Bitola	7500
E4	Anita	Toshevska	W2	Bitola	7500
E5	Naum	Stojanoski	W3	Ohrid	5000
E6	Hristina	Miloseska	W3	Ohrid	5000
E7	Dimitar	Tocko	W1	Skopje	10000
E8	Biljana	Osmani	W2	Bitola	7500
NULL	NULL	NULL	W4	Tetovo	1000

The result Table 8.33 includes all employees and all warehouses. For employees without an assigned warehouse (which do not exist in the provided data), the Warehouse_ID, Location, and Capacity columns would be NULL. For warehouses that have no associated employees (like warehouse W4), the employee information columns will be NULL.

8.6.5 The division Operation

The division operation (÷) is a specialized relational algebra operation used to find all tuples in one relation that are associated with every tuple in another relation, effectively answering "for all" types of queries. The notation R ÷ S requires that the attributes of S be a subset of the attributes of R, and the result contains tuples from R projected onto the attributes not in S, but *only for those tuples that have corresponding entries* for every tuple in S. Relational algebra treats the division operation as a derived operation because it corresponds directly to the universal quantifier (∀) in logic, enabling the formulation of queries that require completeness conditions to hold for all related tuples.

> **The division Operation**
>
> **R÷S** The **division** operation finds all tuples in one relation that are related to all tuples in another relation.

The Division operator is used to find tuples in one relation that are associated with all tuples in another relation. It is commonly used when we want to identify entities that satisfy certain conditions across multiple related data sets.

University System Example

req8.32: *Find students enrolled in all courses offered by the 'Computer Science' faculty.*

The division operation is used to find entities in one relation that are related to all entities in another relation. In this case, we are looking for students who are related to all of the "Computer Science" faculty courses.

Relational algebra expression using the Division operator would involve the following steps:

Selection (σ): Relation R1 that filters all tuples from the FACULTY table where the Name is "Computer Science".

$$R1 = \sigma_{Name='Computer\ Science'}\ FACULTY$$

Join (⋈): Join the filtered faculty result (R1) with the COURSE table to find all courses offered by that faculty, in relation R2.

$$R2 = R1 \bowtie_{Faculty_ID} COURSE$$

Projection (π): This selects, in relation S, only the Course_ID from R2. This gives us the set of all courses offered by the 'Computer Science' faculty — this is the divisor.

$$S = \pi_{Course_ID}(R2)$$

Projection (π): This selects, in relation R, the Student_ID and Course_ID pairs from the ENROLLS_IN table. This gives us every (student, course) enrolment pair — this is the dividend.

$$R = \pi_{Student_ID,\ Course_ID}(ENROLLS_IN)$$

Division (÷): Finally, in relation T, we perform a division of R by S.

$$T = R \div S$$

This will give us all the Student_IDs of students enrolled in all courses offered by the 'Computer Science' faculty.

The complete relational algebra expression is as follows:

$$\pi_{Student_ID,\ Course_ID}(ENROLLS_IN) \div \pi_{Course_ID}((\sigma_{Name='Computer\ Science'}\ FACULTY) \bowtie_{Faculty_ID} COURSE)$$

The result in Table 8.34 shows result Student_ID is S006. This is because student S006 is enrolled in CS101, CS201, and CS303. Considering that all these courses are from the Computer Science faculty (based on their course IDs), and that these are all the courses offered by the faculty, S006 is the only student who is enrolled in all of them.

Table 8.34: Results of the Division operation on students enrolled in all courses by the Computer Science faculty.

Student_ID
S006

Company System Example

req8.33: *Find the warehouse(s) that store ALL food products.*

The division operation is used to find entities in one relation that are related to all entities in another relation. In this case, we are looking for warehouses that stock every product in the 'Food' category.

Relational algebra expression using the Division operator would involve the following steps:

Selection (σ): Relation F that filters all tuples from the PRODUCT table where the Category is 'Food'.

$$F = \sigma_{Category='Food'}(PRODUCT)$$

Projection (π): This selects, in relation S, only the Product_ID from F. This gives us the set of all food product IDs — this is the divisor.

$$S = \pi_{Product_ID}(F)$$

Projection (π): This selects, in relation R, the Warehouse_ID and Product_ID pairs from the STORED_IN table. This gives us every (warehouse, product) storage pair — this is the dividend.

$$R = \pi_{Warehouse_ID,\ Product_ID}(STORED_IN)$$

Division (÷): Finally, in relation T, we perform a division of R by S.

$$T = R \div S$$

This will give us all the Warehouse_IDs of warehouses that store every food product.

The complete relational algebra expression is:

$$\pi_{Warehouse_ID,\ Product_ID}(STORED_IN) \div \pi_{Product_ID}(\sigma_{Category='Food'}(PRODUCT))$$

The result in Table 8.35 shows Warehouse_ID W1. The three food products are Ajvar (Product_ID 1), Tavce Gravce (Product_ID 2), and Selsko Meso (Product_ID 12). Checking the STORED_IN relation, all three are stored in warehouse W1 (Skopje). No other warehouse stores all three food products, so W1 is the only result.

Table 8.35: Results of the division operation for warehouses that store all food products.

Warehouse_ID
W1

req8.34: Find the suppliers that supply all products supplied by the suppliers located in 'Kavadarci'.

This relational algebra expression uses the division operation to find the Suppliers that supply products by all suppliers located in 'Kavadarci'.

Let's define the relations needed for this operation:

Selection (σ): Relation K1 filters the suppliers that are located in 'Kavadarci'.

K1 = $\sigma_{City='Kavadarci'}$(SUPPLIER)

Projection (π): This relation K contains the Supplier_ID of the Kavadarci-based suppliers.

K = $\pi_{Supplier_ID}$(K1)

Join and Projection (⋈, π): This relation Products_K contains the Product_IDs of all products supplied by the Kavadarci-based suppliers — this is the divisor.

Products_K = $\pi_{Product_ID}$(SUPPLIES $\bowtie_{Supplier_ID}$ K)

Projection (π): This relation R contains all (Supplier_ID, Product_ID) pairs from the SUPPLIES table — this is the dividend.

R = $\pi_{Supplier_ID,\ Product_ID}$(SUPPLIES)

Division (÷): Finally, in relation T, we divide R by Products_K to find all suppliers that supply every product supplied by the Kavadarci-based suppliers.

T = R ÷ Products_K

The complete relational algebra expression is:

$\pi_{Supplier_ID,\ Product_ID}$(SUPPLIES) ÷ $\pi_{Product_ID}$(SUPPLIES $\bowtie_{Supplier_ID}$ ($\pi_{Supplier_ID}$($\sigma_{City='Kavadarci'}$(SUPPLIER))))

The only supplier in Kavadarci is Tikves (S2). The products supplied by S2 are Product_IDs 2, 4, 6, 8, and 10. Checking the SUPPLIES relation, S2 is the only supplier that supplies all five of those products, so the result is S2.

Table 8.36: Results of the division operation for the SUPPLIES and SUPPLIER relations located in 'Kavadarci'.

Supplier_ID
S2

8.7 Extended Relational Algebra

Aggregation and grouping operations, part of the extended relational algebra, are used to perform **summation** or **aggregation** of data, similar to the totals at the end of the product supplies monthly report or the student transcript average score report. In some cases, **grouping** the data by Supplier_ID in the supplies relation can be beneficial for the subtotals report. Their operations cannot be performed using the basic relational algebra operations, so extended algebra operations must be used.

8.7.1 Aggregate Functions

As well as simply retrieving certain tuples and attributes of one or more relations, we often want to perform some form of summation or aggregation of data, similar to the totals at the bottom of a report, or some form of grouping of data, similar to subtotals in a report. These operations cannot be performed using the basic relational algebra operations considered above.

The main aggregate functions are:

- COUNT – returns the number of values in the associated attribute;
- SUM - returns the sum of the values in the associated attribute;
- AVG - returns the average of the values in the associated attribute;
- MIN - returns the smallest value in the associated attribute;
- MAX - returns the largest value in the associated attribute.

The aggregate functions

$\varphi_{<aggregate_function>}(R)$ The **aggregate function** list, the relation R to define a relation over the aggreage list. The function contains one or more (<aggregate_function>, <attribute>) pairs.

University System Example

req8.35a*: Count students enrolled in courses in the ENROLLS_IN relation.*

A relational algebra expression that uses an **aggregate** operation is:

$$\varphi_{COUNT(Student_ID)}(ENROLLS_IN)$$

This aggregate function counts the number of students (Student_ID) from the ENROLLS_IN relation; the outcome is given in Table 8.36 a).

req8.35b*: Count **unique** students enrolled in courses in the ENROLLS_IN relation.*

A relational algebra expression that counts unique students with an **aggregate** operation is:

$$\varphi_{COUNT}(\delta(\pi_{Student_ID}(ENROLLS_IN)))$$

After analyzing both **req8.35a) and b),** the results outcomes is given in Table 8.37

Table 8.37: Results of the **aggregate function, req8.35** a) and b).

req8.35 a)

Student_ID
12

req8.35 b)

Student_ID
6

Company System Example

req8.36*: Count employees and find average, minimum, and maximum salary in the EMPLOYEE relation.*

A relational algebra expression that uses an **aggregate** operation is:

$$\varphi_{COUNT(Employee_ID),SUM(Salary),AVG(Salary),MIN(Salary),MAX(Salary)}(EMPLOYEE)$$

This aggregate function counts the number of employees (Employee_ID) and calculates the total salary (function SUM), average salary (function AVG), minimum salary (function MIN), and maximum salary (function MAX) from the EMPLOYEE relation; the outcome is given in Table 8.38.

Table 8.38: Result of the **aggregate function** operation in EMPLOYEE relation.

COUNT(Employee_ID)	*SUM(Salary)*	*AVG(Salary)*	*MIN(Salary)*	*MAX(Salary)*
8	340000	42500	25000	60000

8.7.2 Grouping Operation

Grouping operations enable the partitioning of tuples into groups based on shared attribute values, typically to support aggregate computations across these groups.

The Grouping operations

$\gamma_{<aggregate_function>}(R)$ The grouping operation on relation R is grouping attributes and then applies the aggregate function list to define a new relation. The aggregate function contains one or more (<aggregate_function>, <attribute>) pairs.

The resulting relation contains the group attributes, along with the results of each of the aggregate functions.

University System Example

req8.37*: Count students enrolled in courses and then find the average, minimum, and maximum grade in ENROLLS_IN relation and group them by Student_ID.*

Relational algebra expression that uses grouping operation to count the number of courses per student and then find the average, minimum, and maximum grades for each student is:

$\gamma_{Student_ID,\ COUNT(Course_ID)\ as\ Courses_Enrolled,\ AVG(Grade)\ as\ Average_Grade,\ MIN(Grade)\ as\ Minimum_Grade,\ MAX(Grade)\ as\ Maximum_Grade}(ENROLLS_IN)$

The relational algebra expression would involve the following steps:

- **Grouping (γ):** represents the grouping and aggregation operation.
- **Student_ID** is the grouping attribute. This means the aggregate functions will be applied to each unique Student_ID.
- **COUNT(Course_ID)** counts the number of courses each student is enrolled in.
- **AVG(Grade), MIN(Grade),** and **MAX(Grade)** calculate the average, minimum, and maximum grades for each student, respectively.

Table 8.39: Result of the grouping and aggregation operation that counts the number of courses per student and then finds the average, minimum, and maximum grades for each student.

Student_ID	*Courses_Enrolled*	*Average_ Grade*	*Minimum_ Grade*	*Maximum_ Grade*
S001	1	9	9	9
S002	2	6	5	7
S003	2	6.5	5	8
S004	2	7	6	8

S005	2	8.5	7	10
S006	3	8	6	10

Company System Example

req8.38*: Find the number of employees working in each department and sum their salaries.*

The relational algebra expression to find the number of employees and the sum of their salaries for each Department_ID using only the EMPLOYEE table is:

$$\gamma_{\text{Department_ID, COUNT(Employee_ID) as NumEmployees, SUM(Salary) as TotalSalary}}(\text{EMPLOYEE})$$

Explanation of relational algebra expression that uses grouping operations:

- **Grouping (γ):** This is the grouping operator applied directly to the EMPLOYEE relation.
- **Department_ID**: This is the grouping attribute. The tuples will be grouped based on unique Department_ID values.
- **COUNT(Employee_ID) as NumEmployees** - This aggregate function counts the number of employees within each Department_ID group.
- **SUM(Salary) as TotalSalary** - This aggregate function calculates the sum of salaries for employees within each Department_ID group.

Table 8.40: Grouping employees working in each department and summing their salaries.

Department_ID	*NumEmployees*	*TotalSalary*
D1	3	145000
D2	3	120000
D3	2	75000

8.7.3 Sorting Operation

Sorting operations allow for ordering the output data based on specified attributes and criteria. Commonly, sorting arranges tuples in a relation according to one or more attributes in ascending or descending order.

> **The Sorting operations**
>
> $\tau_{sorting_attributes}(R)$ The **Sorting** operation on relation R allow for ordering the output data based on specified attributes and criteria.

University System Example

req8.39*: "Sort the professors by the hire date."*

The relational algebra expression to *sort the professors by the hire_date* is:

τ_{Hire_date}**(PROFESSOR)**

This expression will take the PROFESSOR relation and produce a new relation with the same schema but with the rows sorted in ascending order by the Hire_date attribute.

Table 8.41: Results of the sorted list of professors by hire date.

Professor_ID	*First_ name*	*Last_name*	*Academic_title*	*Hire_date*	*Faculty _ID*	*Heads_ Faculty_ID*
P003	Kiril	Prosvetitel	Full Professor	25.09.2001	F004	NULL
P004	Metodij	Prosvetitel	Full Professor	13.09.2002	F001	F001
P002	Naum	Ohridski	Associate Professor	15.01.2004	F002	F002
P001	Kliment	Ohridski	Full Professor	01.08.2007	F001	NULL
P005	Blaze	Koneski	Assistant Professor	19.10.2012	F005	F005
P006	Richard	Johnson	Associate Professor	02.09.2022	F001	NULL

Note: Office and Email attributes are omitted from Table 8.41 for display purposes; in a full implementation, they would appear between Last_name and Academic_title.

Company System Example

req8.40*: "Sort the WAREHOUSE by the capacity, starting from the smallest capacity."*

Because standard relational algebra is based on set theory, we use the Sort operator. The relational algebra expression to sort the WAREHOUSE by the Capacity, starting from the smallest capacity, is:

$\tau_{Capacity}$**(WAREHOUSE)**

This expression will take the WAREHOUSE relation and produce a new relation with the same schema but with the rows sorted in ascending order by the Capacity attribute, as shown in Table 8.42.

Table 8.42: Sorted list of WAREHOUSE by the Capacity in ascending order.

Warehouse_ID	*Location*	*Capacity*	*Department_ID*
W4	Tetovo	1000	D2
W3	Ohrid	5000	D3
W2	Bitola	7500	D2
W1	Skopje	10000	D1

req8.41*: "Sort the DEPARTMENT by the budget, starting from the biggest."*

The relational algebra expression to sort the DEPARTMENT by the Budget, starting from the biggest, is:

$\tau_{Budget\ \downarrow}$**(DEPARTMENT)**

This expression sorts the DEPARTMENT relation by the Budget attribute in descending order, indicated by a downward arrow or the 'desc' keyword in relational algebra, to prioritize higher budgets, as shown in Table 8.43.

Table 8.43: Sorted list of DEPARTMENT by the Budget in descending order.

Department_ID	*Dept_name*	*Location*	*Budget*
D1	Sales	Skopje	500000
D2	Marketing	Bitola	300000
D3	Human Resources	Ohrid	200000

8.8 Complex Relational Algebra

Complex relational algebra queries often involve multiple sequential operations, making the execution order crucial for optimization. Proper ordering and strategies for query operations are essential to ensure efficient query execution (Elmasri & Navathe, 2016). A **multi-step** query chain performs relational operations such as selection, projection, join, and set operations to progressively transform data until the final result is achieved (Date, 2003). For example, retrieving specific customer orders that meet certain criteria may require selecting relevant tuples, joining them with customer relations, projecting necessary attributes, and applying additional filtering or aggregation operations (Garcia-Molina et al., 2008). The sequence of these operations significantly impacts query performance, as different orders can produce varying intermediate result sizes and computational costs (Silberschatz et al., 2019). Therefore, query optimization aims to reorder and restructure **relational algebra expressions** to minimize execution time and resource use while maintaining logical equivalence and producing the same final results (Elmasri & Navathe, 2016).

A fundamental optimization principle involves performing **selection operations** as early as possible in the query execution sequence, which can help readers feel confident that their strategies lead to more efficient processing by reducing the number of tuples processed by subsequent operations (Codd, 1970; Date, 2003). Similarly, executing **projection operations** early eliminates unnecessary attributes from intermediate relations, decreasing data volume and memory requirements throughout the query processing pipeline (Elmasri & Navathe, 2016). The placement of **join operations** is particularly critical, as joins can generate large intermediate relations, especially when implemented conceptually as Cartesian products followed by selection (Silberschatz et al., 2019). Applying selection conditions before join operations—a technique commonly referred to as predicate **pushdown**—can significantly reduce the size of relations being joined and improve overall query performance (Garcia-Molina et al., 2008). Understanding algebraic properties such as commutativity and associativity enables query optimizers to explore alternative execution plans and select strategies that minimize disk I/O, memory usage, and computational overhead while ensuring correct and efficient query execution (Date, 2019; Elmasri & Navathe, 2016). Explaining how these properties influence plan selection helps readers connect theoretical concepts with practical optimization strategies.

8.8.1 Multi-Step Queries

Multi-step queries are essential in practical database applications because real-world information requirements rarely involve data from a single relation or simple filtering conditions (Elmasri & Navathe, 2016). Business intelligence, reporting systems, and data analysis tasks typically require integrating information distributed across multiple related relations, necessitating sequential operations to collect, filter, combine, and transform data into meaningful results (Date, 2003; Connolly & Begg, 2015). For example, in a sales report identifying top-performing products by warehouse location, multiple tables must be joined to link product sales data to the warehouses where those products are stored, as illustrated in the company system examples. Including specific examples of such queries helps readers understand their practical significance and application context.

Decomposing complex queries into discrete relational operations helps readers feel more in control and more confident in their ability to understand and manage them. This approach makes queries easier to understand, test, and maintain, while enabling database management systems to analyze and optimize each step independently (Silberschatz et al., 2019). This decomposition further supports the inspection of intermediate results during query development, promotes query reuse through subquery composition, and enables query optimizers to apply algebraic transformation rules that improve execution efficiency without altering the logical semantics or final results of the queries (Date, 2019).

University System Example

req8.42: *List students enrolled in courses taught by professors hired before 2005.*

To find students enrolled in courses taught by professors hired before 2005, we must link students to courses, courses to professors, and then filter based on the professor's hire date. Relational algebra expression, with step-by-step breakdown using multiple operations:

- **Selection (σ):** Filter the PROFESSOR table for those hired before January 1, 2005.
  $$R1 = \sigma_{Hire_date < '2005\text{-}01\text{-}01'}(PROFESSOR)$$
- **Join (⋈):** Join the filtered professors (R1) with the COURSE table to find the courses they teach.
  $$R2 = R1 \bowtie_{Professor_ID} COURSE$$
- **Join (⋈):** Join those courses (R2) with the ENROLLS_IN table to find the students enrolled in those specific courses.
  $$R3 = R2 \bowtie_{Course_ID} ENROLLS_IN$$
- **Join (⋈):** Join the result (R3) with the STUDENT table to retrieve student names.
  $$R4 = R3 \bowtie_{Student_ID} STUDENT$$
- **Projection (π):** Select the final columns to display (e.g., Student ID and Names).
  $$\pi_{Student_ID,\ First_name,\ Last_name}(R4)$$

Complete relational algebra expression that finds students enrolled in courses taught by professors hired before 2005 will be:

$$\pi_{Student_ID,\ First_name,\ Last_name}((((\sigma_{Hire_date < '2005\text{-}01\text{-}01'}(PROFESSOR)) \bowtie_{Professor_ID} COURSE) \bowtie_{Course_ID} ENROLLS_IN) \bowtie_{Student_ID} STUDENT)$$

Students enrolled in courses taught by professors hired before 2005 are given in Table **8.44.**

Table 8.44: Results of University example, showing the students enrolled in courses taught by professors hired before 2005

Student_ID	*First_name*	*Last_name*
S002	Bojan	Jovansen
S003	Kosta	Dojcinoski
S004	David	Boskovski

req8.43: *List all people (students and professors) with their names and email addresses.*

Relational algebra expression starts by decomposing a complex query into discrete relational operations and needs to ensure that the relations we are uniting are union-compatible. This means they must have the same number of attributes, and corresponding attributes must have compatible data types.

- **Natural join (⋈):** Relation **R1**, as a join of the STUDENT table with the EMAIL table to get the student's name and email.

  R1 = STUDENT $\bowtie_{\text{Student_ID}}$ EMAIL

- **Projection (π):** In relation **StudentsInfo** of the students' First_name, Last_name, and Email.

  StudentsInfo=$\pi_{\text{First_name, Last_name, Email}}$(R1)

- **Projection (π):** In relation **ProfessorsInfo** from the PROFESSOR relation, the attributes First_name, Last_name, and Email.

  ProfessorsInfo=$\pi_{\text{First_name, Last_name, Email}}$(PROFESSOR)

- **Union (∪)**: Finally, the union operation is performed on StudentsInfo and ProfessorsInfo.

  StudentsInfo ∪ ProfessorsInfo

A complete relational algebra expression that lists people with their names and email addresses is:

$\pi_{\text{First_name, Last_name, Email}}$ (STUDENT $\bowtie_{\text{Student_ID}}$ EMAIL) ∪ $\pi_{\text{First_name, Last_name, Email}}$ (PROFESSOR)

The results from the given relational algebra expression are shown in Table 8.45. Notice there are repeating names but different email addresses; this is because the same student (First_name and Last_name) has multiple email addresses (private and personal) in the EMAIL relation.

Table 8.45: Results of the University example with students and professors, with their names and email addresses

First_name	*Last_name*	*Email*
Anica	Stojnasen	anica.s@gmail.com
Anica	Stojnasen	anica.stojansen@univeristy.edu
Blaze	Koneski	blaze.koneski@univeristy.edu

Bojan	Jovansen	bojan.j@gmail.com
Bojan	Jovansen	bojan.jovansen@univeristy.edu
David	Boskovski	david.boskovski@univeristy.edu
Elena	Mitan	elena.mitan@gmail.com
Elena	Mitan	elena.mitan@univeristy.edu
Kiril	Prosvetitel	kiril.prosvetitel@univeristy.edu
Kliment	Ohridski	kliment.ohridski@univeristy.edu
Kosta	Dojcinoski	kosta.dojcinoski@univeristy.edu
Metodij	Prosvetitel	metodij.prosvetitel @univeristy.edu
Naum	Ohridski	naum.ohridski@univeristy.edu
Richard	Johnson	richard.johnson@univeristy.edu
Sarah	Smith	sarah.smith@gmail.com
Sarah	Smith	sarah.smith@univeristy.edu

Company System Example

req8.44: *List all employees who are managing the department names.*

This task requires a join operation between the EMPLOYEE and DEPARTMENT tables. The join condition is that the Manages_Dept_ID attribute in the EMPLOYEE table matches the Department_ID in the DEPARTMENT table. This operation will list the employees who are department managers and the departments they manage.

The relational algebra expression steps for this operation are:

- **Equijoin (⋈)**: Relation R1 as equijoin between the EMPLOYEE and DEPARTMENT relations, using the explicit condition Manages_Dept_ID = Department_ID to match each employee to the department they manage.

  $$\mathbf{R1 = EMPLOYEE} \bowtie_{\mathbf{Manages_Dept_ID = Department_ID}} \mathbf{DEPARTMENT}$$

- **Projection (π):** Select the applicable columns to display (e.g., employees and department names).

  $$\pi_{\mathbf{Employee_ID,First_name,\ Last_name,\ Dept_name}}\ \mathbf{(R1)}$$

Complete relational algebra expression that employees who manage their department are:

$$\pi_{\mathbf{Employee_ID,First_name,\ Last_name,\ Dept_name}}\mathbf{(EMPLOYEE} \bowtie_{\mathbf{Manages_Dept_ID = Department_ID}} \mathbf{DEPARTMENT)}$$

The result of this operation is a list of the employees who are department managers and the departments they manage, shown in Table **8.46.**

Table 8.46: Results of the employees that are managing the department names

Employee_ID	*First_name*	*Last_name*	*Dept_name*
E1	Zoran	Pejovski	Sales
E3	Marija	Naumoska	Marketing

E5	Naum	Stojanoski	Human Resources

req8.45: *Find all of the employees' first name, last name, and city who work in the same city as the department location.*

To find employees working in the same city as their department's location, you need to join the EMPLOYEE and DEPARTMENT tables and filter the results where the City in the EMPLOYEE table matches the Location in the DEPARTMENT table. Relational algebra expression, with step-by-step breakdown using multiple operations:

- **Natural join (⋈):** Relation R1 as natural join (⋈) on the EMPLOYEE and DEPARTMENT relations using their shared Department_ID attribute.

  $$\mathbf{R1 = EMPLOYEE \bowtie_{Department_ID} DEPARTMENT}$$

- **Selection (**σ): Relation R2 that filters the rows where the City column from the EMPLOYEE table is equal to the Location column from the DEPARTMENT table

  $$\mathbf{R2 = \sigma_{City=Location}(R1)}$$

- **Projection (**π**):** Select the applicable columns to display (First_name, Last_name, and City).

  $$\mathbf{\pi_{First_name,\ Last_name,\ City}\ (R2)}$$

Complete relational algebra expression that lists employees who work in the same city as the department location is:

$$\mathbf{\pi_{First_name,\ Last_name,\ City}(\sigma_{City=Location}(EMPLOYEE \bowtie_{Department_ID} DEPARTMENT))}$$

Result of this operation that lists all employees first name, last name, city that work in the same city as the department location, shown in Table 8.47.

Table 8.47: Result of the requirement for employees who work in the same city as the department location.

First_name	*Last_name*	*City*
Zoran	Pejovski	Skopje
Nikola	Delcevski	Skopje
Marija	Naumoska	Bitola
Anita	Toshevska	Bitola
Naum	Stojanoski	Ohrid
Hristina	Miloseska	Ohrid

req8.46: *Count employees who have a salary greater than 40000.*

To count the number of employees with a salary greater than 40,000, you need to apply a selection operation on the EMPLOYEE relation based on the salary condition and then count the resulting tuples. The relational algebra expression for this operation involves a selection (σ) and an aggregate function (φ), specifically COUNT, in these steps:

- **Selection (**σ): Relation R1 that filters all tuples from the EMPLOYEE relation where the Salary is greater than 40,000.

  $$R1=\sigma_{Salary>40000}(EMPLOYEE)$$
- **Projection (**π**):** Relation R2 that selects Employee_ID attribute of the resulting tuples.

  $$R2=\pi_{Employee_ID}(R1)$$
- **Aggregate function** (φ): That finally counts the employees in the relation.

  $$\varphi_{COUNT}(R2)$$

Complete relational algebra expression that counts the employees with a salary greater than 40,000 is:

$$\varphi_{COUNT}(\pi_{Employee_ID}(\sigma_{Salary>40000}(EMPLOYEE)))$$

Table 8.48: Result of counting the number of employees with a salary greater than 40,000

NumberOfEmployees
4

Company System Example

req8.47: *List the ID, first name, and last name of all employees who work in a warehouse located in 'Skopje'.*

Relational algebra expression starts by decomposing a complex query into discrete relational operations into these steps:

- **Selection (**σ): Relation R1 that filters the WAREHOUSE relation to find only those where the Location is 'Skopje'.

  $$R1=\sigma_{Location = 'Skopje'}WAREHOUSE$$
- **Natural Join (⋈):** Relation R2 that combines this filtered result with the EMPLOYEE relation using the common attribute Warehouse_ID.

  $$R2= R1 \bowtie_{Warehouse_ID} EMPLOYEE$$
- **Projection (**π**):** That finally selects only the specific columns needed (Employee_ID, First_name, Last_name, and Location).

  $$\pi_{Employee_ID,\ First_name,\ Last_name,\ Location}(R2)$$

Complete relational algebra expression that lists all employees who work in a warehouse located in 'Skopje' is:

$$\pi_{Employee_ID,\ First_name,\ Last_name,\ Location}((\sigma_{Location = 'Skopje'}WAREHOUSE) \bowtie EMPLOYEE)$$

Table 8.49: Result of the requirement for employees that work in a warehouse located in 'Skopje'.

Employee_ID	*First_name*	*Last_name*	*Location*
E1	Zoran	Pejovski	Skopje
E2	Nikola	Delcevski	Skopje

E7	Dimitar	Tocko	Skopje

8.8.2 Alternative approaches comparison

Complex relational algebra queries involve multiple sequential operations, making the execution order crucial for optimization, and proper ordering and strategies for query operations are essential to ensure efficient query execution (Elmasri & Navathe, 2016). A multi-step query chain performs relational operations such as selection, projection, join, and set operations to progressively transform data until the final result is achieved (Date, 2003). One such alternative involves using set difference operations rather than joins to find records that lack corresponding matches in related tables. Where an INNER JOIN retrieves only matching rows, a set difference (−) approach isolates records in one relation that have no counterpart in another, which is particularly useful for identifying customers who have never placed an order, products that have never been sold, or employees assigned to no active project (Garcia-Molina et al., 2008).

Company System Example

req8.48(req8.17): *List all cities where there are customers but no employees.*

This company system example on relational set difference (or Minus) operation has been demonstrated using set theory to subtract the set of employee cities from the set of customer cities. In section *8.5.3,* the relational algebra expression using the MINUS Operation was presented as:

$$\pi_{City}(CUSTOMER) - \pi_{City}(EMPLOYEE)$$

In this section of the alternative approach, using a join-based approach will be demonstrated. While the standard join-based approach, we utilize a Left Outer Join (⟕) between the CUSTOMER and EMPLOYEE relations on the City attribute. By filtering for records where the employee side of the join is null, we isolate cities unique to customers. In relational algebra, the Left Outer Join (⟕) retains all tuples from the left relation (CUSTOMER). We then select (σ) the rows where the corresponding employee data is null (ω), see section 7.3.7 Characteristics of Relations, and project (π) the City.

$$\pi_{City}(\sigma_{Employee_ID = \omega}(CUSTOMER \overset{}{\bowtie}_{CUSTOMER.City = EMPLOYEE.City} EMPLOYEE))$$

Based on the data provided in the company model, the Customer cities are: Skopje, Bitola, Ohrid, Prilep, Tetovo, Kumanovo, and Veles. On the other side, Employee cities: Skopje, Bitola, Ohrid, Prilep, Tetovo, showing all cities where there are customers but not employees is the same as in Table 8.50.

Table 8.50: Results of the ***Set Difference (or Minus)*** operation for the CUSTOMER and EMPLOYEE relations over City data.

City
Kumanovo
Veles

req8.49a: *Based on the orders, calculate the total cost for each customer per product.*

req8.49b: *Based on the orders, calculate the total cost for each product per customer.*

This company system example uses the ORDER and ORDER_LINE relation to perform a join to link customers to their purchased products and then apply aggregation and sorting.

Relational algebra expression starts by decomposing a complex query into discrete relational operations into these steps:

- **Natural Join (⋈):** Relation R that joins the relations ORDER and ORDER_LINE, using the common attribute Order_ID.

  R= ORDER ⋈ ORDER_LINE

- **Grouping operation (γ)**: Groups the data and calculates the sum of costs. To calculate the total cost, it uses the sum of the product of Unit_price and Quantity.

  In this step, we can have alternative approaches to grouping the attributes from the relation R.

  - First approach G1 of grouping first by Customer_ID and then Product_ID, as stated in **req8.49a**.

  G1= $\gamma_{\text{Customer_ID, Product_ID, SUM(Unit_price* Quantity) as TotalCost}}$ (R)

  - Second approach G2 of grouping first by Product_ID and then Customer_ID, as stated in **req8.49b**.

  G2= $\gamma_{\text{Product_ID, Customer_ID, SUM(Unit_price* Quantity) as TotalCost}}$ (R)

- **Sorting operation (τ):** Sorts the final results by TotalCost. Also, in this step, we have two alternative approaches to sorting:

  - First approach ascending order, by default, T1

  T1= $\tau_{\text{TotalCost}}$

  - Second approach descending order (↓), T2

  T2= $\tau_{\text{TotalCost}\downarrow}$

Considering the alternative approaches to grouping and sorting operations, we will present **two different** complete relational algebra expressions:

a. complete relational algebra expression that calculates the total cost for each customer per product, as specified in **req8.49a**, and sort the cost in descending order.

T2(G1(R)) ≡ $\tau_{\text{TotalCost}\downarrow}$ ($\gamma_{\text{Customer_ID, Product_ID, SUM(Unit_price* Quantity) as TotalCost}}$ (ORDER ⋈ ORDER_LINE))

The first approach that calculates the total cost for each customer per product and sorts the cost in descending order, from largest to smallest, is as given in Table 8.51.

Table 8.51: Resulting Table that calculates the total cost for each *customer per product*.

Customer_ID	*Product_ID*	*TotalCost*
C4	10	148,800
C6	7	55,800
C7	8	31,000
C1	9	23,250
C3	4	2,790
C2	11	2,232
C4	5	1,860
C3	12	1,736
C1	1	744
C2	3	682
C5	6	620

b. complete relational algebra expression. The relational algebra expression as specified in **req8.49b**, and sort the cost in ascending order.

T1(G2(R)) ≡ $\tau_{\text{TotalCost}}$ ($\gamma_{\text{Product_ID, Customer_ID, SUM(Unit_price* Quantity) as TotalCost}}$ (ORDER ⋈ ORDER_LINE))

The second approach that calculates the total cost for each *product per customer* and sorts the cost in ascending order, from smallest to largest, is as given in Table 8.52.

Table 8.52: Resulting Table that calculates the total cost for each *product per customer*.

Product_ID	*Customer_ID*	*TotalCost*
6	C5	620
3	C2	682
1	C1	744
12	C3	1,736.00
5	C4	1,860.00
11	C2	2,232.00
4	C3	2,790.00
9	C1	23,250.00
8	C7	31,000.00
7	C6	55,800.00
10	C4	148,800.00

University System Examples

req8.50: *List faculties with no current course enrollments.*

To identify faculties that currently have no students enrolled in any of their offered courses, we compare the set of all faculties against the set of faculties that have at least one enrollment record. Relational algebra expression uses the Set Difference or **Minus** (**-**), to find the "missing" faculties. Here is a step-by-step breakdown:

- **Identify Faculties with Enrollments** (R1): Join the COURSE table with the ENROLLS_IN table to find every course that has at least one student.

  $$\mathbf{R1 = COURSE} \bowtie_{\mathbf{Course_ID}} \mathbf{ENROLLS_IN}$$

- **Project Faculty IDs from Enrollments** (R2): Extract only the Faculty_ID from the previous result.

  $$\mathbf{R2} = \pi_{\mathbf{Faculty_ID}}\mathbf{(R1)}$$

- **Project All Faculty IDs** (R3): Get the complete list of all Faculty_IDs from the FACULTY table.

  $$\mathbf{R3} = \pi_{\mathbf{Faculty_ID}}\ \mathbf{(FACULTY)}$$

- **Set Difference** (R4): Subtract faculties with enrollments (R2) from all faculties (R3) to find those with zero enrollments.

  $$\mathbf{R4 = R3 - R2}$$

- **Final Join and Projection**: Join the result back with the FACULTY table to get the name and details.

  $$\pi_{\mathbf{Faculty_ID,Name}}\mathbf{(R4} \bowtie \mathbf{FACULTY)}$$

The complete relational algebra expression that finds faculties with no current course enrollments is

$$\pi_{\mathbf{Faculty_ID,Name}}\mathbf{((}\pi_{\mathbf{Faculty_ID}}\ \mathbf{(FACULTY) -}\ \pi_{\mathbf{Faculty_ID}}\ \mathbf{(COURSE} \bowtie_{\mathbf{Course_ID}} \mathbf{ENROLLS_IN))} \bowtie \mathbf{FACULTY)}$$

Table 8.53: The results of the University example show faculties with no current course enrollments.

Faculty_ID	*Name*
F003	Economy & Tourism

req8.51: *Find the most popular course (highest enrollment).*

To find the most popular course, we identify which Course_ID appears most frequently in the ENROLLS_IN table.

A relational algebra expression has to count enrollments and then use a self-comparison to find the maximum. Here is the step-by-step breakdown:

- **Grouping operation (γ)**: To c**ount enrollments per course** (R1) that groups the ENROLLS_IN table by Course_ID and counts the students by Student_ID.

  $$\mathbf{R1} = \gamma_{\mathbf{Course_ID,\ COUNT(Student_ID)\ as\ Count}}\ \mathbf{ENROLLS_IN}$$

- **Aggregate Function (φ)**: To f**ind the maximum enrollment** (R2), which determines the highest count value from **R1.**

  $$\mathbf{R2} = \varphi_{\mathbf{MAX(Count)}}\mathbf{(R1)}$$

- **Equijoin ($\bowtie_{<join_condition>}$):** To i**dentify the course name with that maximum value in** R3, by selecting the tuple from R1 where the count matches the maximum found in R2.

  $$\mathbf{R3 = R1 \bowtie_{Count = MAX(Count)} R2}$$

- **Natural Join($\bowtie$)** That joins **with course details** (R4): Join R3 with the COURSE table to retrieve the descriptive title.

  $$\mathbf{R4 = R3 \bowtie_{Course_ID} COURSE}$$

- **Projection (π):** That finally selects only the specific columns needed (COURSE_ID, Title, and the count).

  $$\mathbf{\pi_{Course_ID,\ Title,\ Count}(R4)}$$

The complete relational algebra expression that finds the most popular course is

$$\mathbf{\pi_{Course_ID,\ Title,\ Count}\ ((\gamma_{Course_ID,\ COUNT(Student_ID)\ as\ Count}\ ENROLLS_IN)\ \bowtie_{Count=MAX(Count)}\ (\varphi_{MAX(Count)}(\gamma_{Course_ID,\ COUNT(Student_ID)\ as\ Count}\ ENROLLS_IN))\ \bowtie_{Course_ID}\ COURSE)}$$

Table 8.54: Results of the University example show the most popular course

Course_ID	*Title*	*Count*
CS303	Database Systems	2

University System Example

req8.52 *Find TEXTBOOK Titles that are given for ASSIGNMENT.*

To find TEXTBOOK Titles that are given for ASSIGNMENT via intersection, the most logical intersection would be to find the titles of textbooks that are given for assignment, as these are the titles common to both the set of all textbook titles and the set of titles associated with assignments.

Relational algebra expression uses a set intersection operation (R∩S), which returns all tuples that are present in both relation R and relation S. Similar to the difference operation, both relations R and S must be union-compatible. To find the titles of textbooks that are given for assignment, we can perform an intersection between the set of all textbook titles and the set of titles that are part of an assignment. Here is the step-by-step breakdown of the Intersection Operation:

- **Projection (π):** Relation R that contains all distinct titles from the Textbook table.

  $$\mathbf{R=\pi_{Title}(TEXTBOOK)}$$

- **Equijoin ($\bowtie_{<join_condition>}$): Relation S with Titles Associated with Assignments.** This relation contains the titles of textbooks that have been assigned. We obtain this by **equijoin of** the Assignment table with the TEXTBOOK table on their common attribute ISBN, and then projecting only the Title attribute.

  $$\mathbf{S=\pi_{Title}(ASSIGNMENT \bowtie_{ASSIGNMENT.ISBN\ =\ TEXTBOOK.ISBN} TEXTBOOK)}$$

The complete relational algebra expression that finds the titles of textbooks that are given for assignment, we perform the intersection operation R∩S:

$$\mathbf{\pi_{Title}(TEXTBOOK) \cap \pi_{Title}(ASSIGNMENT \bowtie_{ASSIGNMENT.ISBN\ =\ TEXTBOOK.ISBN} TEXTBOOK)}$$

This expression means "find the titles that are common to both the set of all textbook titles and the set of titles linked to assignments." The results of the execution are given in Table 8.55.

Table 8.55: Results of the University example show the titles of textbooks that are given for assignment

Title
Introduction to Programming by Examples
Object-oriented programming with examples
Fundamentals of Database Systems
Electric Circuits
Complex circuit analysis and design
Linear Algebra Done Right
Matrix theory and advanced vector spaces
Physics for Scientists and Engineers
Quantum mechanics and advanced thermodynamics
Academic concepts
Research methodology and scientific writing

University System Example

req8.53 *Find the titles of textbooks that are not assigned.*

Scenario ***Set Difference (or Minus) Operation***: To find the titles of textbooks that are not assigned, we can perform a difference operation. First, we project all titles from the TEXTBOOK relation. In this case, we want to find textbook titles that are present in the TEXTBOOK table but do not appear in any ASSIGNMENT.

Relational algebra expression that demonstrates the step-by-step breakdown in the Set Difference Operation:

- **Projection (π):** R that identifies all Titles from the TEXTBOOK relation.
  $$R=\pi_{Title}(TEXTBOOK)$$
- **Equijoin ($\bowtie_{<join_condition>}$):** TEXTBOOK with ASSIGNMENT on ISBN, in relation S1.
  $$S1=TEXTBOOK\bowtie_{TEXTBOOK.ISBN = ASSIGNMENT.ISBN}ASSIGNMENT$$
- **Projection (π):** project the Title from the textbooks that are assigned, in relation S.
  $$S=\pi_{Title}(S1)$$
- **Set Difference (-): T**o find Titles that exist in TEXTBOOK but not in ASSIGNMENT:
  $$R-S$$

Combining these steps into a single expression to find the titles of textbooks that are not assigned, we get a relational algebra expression:

$$\pi_{Title}(TEXTBOOK)-\pi_{Title}(TEXTBOOK\bowtie_{TEXTBOOK.ISBN = ASSIGNMENT.ISBN}ASSIGNMENT)$$

This expression means: "Take all textbook titles and subtract the titles of textbooks that appear in any assignment, leaving only the titles of unassigned textbooks." The results of the execution are given in Table 8.56.

Table 8.56: Results of the University example show titles of textbooks that are not assigned

Title
Academic writing

8.8.3 Query Optimization Concepts

Query optimization concepts are essential in practical DBMS because inefficiently structured queries can lead to poor response times, excessive resource consumption, and overall system performance degradation, particularly when handling large datasets or concurrent user workloads (Elmasri & Navathe, 2016; Silberschatz et al., 2019). To address these challenges, DBMS employ query optimizers that analyze relational algebra expressions and generate efficient execution plans by exploiting algebraic equivalences and statistical metadata such as relation sizes, index availability, and data distribution (Garcia-Molina et al., 2008). The optimizer evaluates logically equivalent query formulations that produce identical results but differ significantly in execution cost, for example, by reordering commutative operations or restructuring join sequences to minimize intermediate relation sizes (Date, 2003). The ordering of operations plays a critical role in performance, as executing selection and projection operations early in the query plan reduces the volume of data processed by subsequent, more expensive operations such as joins and aggregations (Elmasri & Navathe, 2016). Selection pushdown filters irrelevant tuples as close to the data source as possible, while projection pushdown removes unnecessary attributes early, thereby lowering memory usage and data transfer overhead (Silberschatz et al., 2019). A solid understanding of these optimization principles enables database developers to formulate queries that enhance the optimizer's effectiveness, such as by using selective predicates and well-structured joins. Modern query optimizers further balance computational cost against memory usage and disk input/output operations to deliver faster query execution and improved scalability in relational database applications (Date, 2019).

Equivalent Expressions (Relational Algebra Transformations)

Query optimizers use a set of rules, based on the commutative, associative, and distributive properties of relational algebra, to transform the user's SQL query into equivalent forms. The goal is to find an equivalent expression that is cheaper to execute.

Key Equivalence Rules:

Rule Commutativity of Join

The order of joining two relations does not affect the result:

R⋈S ≡ S ⋈ R.

University System Example

Joining STUDENT with ENROLLS_IN yields the same result as joining ENROLLS_IN with STUDENT.

STUDENT ⋈ ENROLLS_IN ≡ ENROLLS_IN ⋈ STUDENT

Company System Example

Below is an example using the EMPLOYEE and WAREHOUSE tables from the COMPANY database.

EMPLOYEE ⋈ WAREHOUSE ≡ WAREHOUSE ⋈ EMPLOYEE

Associativity of Join

The grouping of three or more relations in a join can be rearranged:

(R⋈S) ⋈ T ≡ R ⋈(S ⋈T).

Algebraic equivalences allow query transformation without changing semantics. Key equivalences include: cascade of selections (σ[cond1](σ[cond2](R)) = σ[cond1 AND cond2](R)) allows combining multiple WHERE conditions; commutativity of selection (σ[cond](R) = σ[cond](R) regardless of condition order); commutativity of joins (R ⋈ S = S ⋈ R) allows reordering joins; and pushing selections before joins (σ[cond](R ⋈ S) can become σ[cond](R) ⋈ S if the condition involves only R's attributes). This last transformation is critical because selecting rows early reduces the data volume for the following operations, dramatically improving performance.

University System Example

req8.54 *Find students enrolled in the course*.

When joining STUDENT, ENROLLS_IN, and COURSE, the optimizer can choose to join STUDENT and ENROLLS_IN first, or ENROLLS_IN and COURSE first.

- **Grouped Left: (STUDENT⋈ENROLLS_IN)⋈ COURSE**

  First, find students in their enrollment records via Student_ID.

  Then, adding course descriptive data via Course_ID.

- **Grouped Right: ENROLLS_IN⋈ (STUDENT⋈COURSE)**

  First, links enrollment records to course titles via Course_ID.

  Then, link those enrollment records to the student details via Student_ID.

Regardless of the join grouping used, the resulting data from the University System database is shown in Table 8.57.

Table 8.57: Results of University students enrolled in the course.

Student_ID	***First_name***	***Last_name***	***Course Title***
S001	Sarah	Smith	Database Systems

S002	Bojan	Jovansen	Circuit Analysis
S002	Bojan	Jovansen	Advanced Circuit Analysis
S003	Kosta	Dojcinoski	Linear Algebra
S003	Kosta	Dojcinoski	Matrix theory and advanced vector spaces
S004	David	Boskovski	Physics
S004	David	Boskovski	Advanced Physics
S005	Elena	Mitan	Academic writing
S005	Elena	Mitan	Advanced Academic Writing
S006	Anica	Stojnasen	Intro to Programming
S006	Anica	Stojnasen	Advanced Programming
S006	Anica	Stojnasen	Database Systems

Company System Example

req8.55 *Find all orders and product quantities*

Below is an example using the ORDER, ORDER_LINE, and PRODUCT relations from the COMPANY system.

To list the products contained in each order, we join **ORDER** (R), **ORDER_LINE** (S), and **PRODUCT** (T).

- **Grouped Left: (ORDER ⋈ ORDER_LINE)⋈ PRODUCT**

  First, link orders to their line items via Order_ID.

  Then, link the resulting items to their specific product details via Product_ID.

- **Grouped Right: ORDER ⋈ (ORDER_LINE ⋈ PRODUCT)**

  First, link line items to product details via Product_ID.

  Then, link those detailed items to their parent orders via Order_ID

Regardless of the join grouping used, the resulting data from the Company System database is, as shown in Table 8.58:

Table 8.58: Results of Company System orders and products quantity

Order_ID	*Product_ID*	*Product_name*	*Quantity*
Order1	1	Ajvar	2
Order2	3	Rakija	1
Order3	4	Mastika	3
Order4	5	Sirenje	4
Order5	6	Kajmak	1
Order6	7	Ohrid Pearl	6
Order7	8	Macedonian Ruby	1

Order8	9	Filigree	5
Order9	11	Vranec Wine	2
Order10	12	Selsko Meso	2
Order11	10	Galichnik Wedding Costume	2

Cascading of Selection

A conjunctive selection (AND) can be split into a sequence of selections:

$$\sigma_{A1 \wedge A2}(R) \equiv \sigma_{A1}(\sigma_{A2}(R)).$$

University System Example

Conjunctive (Combined): $\sigma_{City='Ohrid' \wedge Age>25}(STUDENT) \equiv$

Cascaded (Sequence): $\sigma_{City='Ohrid'}(\sigma_{Age>25}(STUDENT)).$

Company System Example

Here is an example using the CUSTOMER table, filtering by Street, City, and Zip_code.

Conjunctive (Combined): $\sigma_{Street='Partizanski\ Odredi\ 10' \wedge City='Skopje' \wedge Zip_code='1000'}(CUSTOMER) \equiv$

Cascaded (Sequence): $\sigma_{Street='Partizanski\ Odredi\ 10'}(\sigma_{City='Skopje'}(\sigma_{Zip_code='1000'}(CUSTOMER)))$.

Commutativity of Selection

In relational algebra, the Commutativity of Selection principle states that when applying multiple filters (selections) to a relation, the sequence in which they are applied does not change the final result set.

The order of multiple selections does not matter, but the cost does:

$$\sigma_{A1}(\sigma_{A2}(R)) \equiv \sigma_{A2}(\sigma_{A1}(R)).$$

The order in which relational operations, particularly joins, are executed influences query performance, as database optimizers use algorithms such as dynamic programming and heuristics to determine the optimal join sequence (Elmasri & Navathe, 2016; Silberschatz et al., 2019). Understanding this process can help database professionals feel more confident in tuning query performance and making informed design choices. Although query writers may focus on syntax or indexing, actual performance is determined by the execution plan generated by the database optimizer. The main challenge in this process is selecting an efficient join order when multiple relations are affected. For a query referencing N relations, the number of possible join sequences grows factorially, creating a complex optimization problem that modern database systems must resolve efficiently (Garcia-Molina et al., 2008).

University System Example

Join ordering example scenario (STUDENT, ENROLLS_IN, COURSE):

req8.56 *Find the names of students enrolled in "Intro to Programming" (CS101).*

Consider a university database with three relations: **STUDENT**, **ENROLLS_IN**, and **COURSE**. Retrieving students enrolled in "Intro to Programming" demonstrates that changing the order of operations can significantly affect query speed, underscoring the importance of operation ordering in query optimization.

- **Inefficient Expression (first Join, then Select):**

$$(\text{STUDENT} \bowtie \text{ENROLLS_IN}) \bowtie \sigma_{\text{Title='Intro to Programming'}}(\text{COURSE})$$

In an **inefficient query expression**, the database system first joins the **STUDENT** and **ENROLLS_IN** relations. Because **ENROLLS_IN** references many students, this join produces a large intermediate result containing approximately 50,000 tuples. If this join is conceptualized as a Cartesian product followed by a join condition, the intermediate data volume can grow even larger before filtering is applied. Understanding how join algorithms work can help database professionals feel more confident in choosing the right approach to handle large datasets. Only after generating this substantial intermediate relation does the query apply a selection condition to isolate tuples related to the course "Intro to Programming" from the **COURSE** relation. This approach wastes computational resources by materializing large temporary structures, consuming excessive memory, CPU time, and disk I/O, only to eliminate most irrelevant data.

- **Optimized Expression (First Select, then Join):**

$$((\sigma_{\text{Title='Intro to Programming'}}(\text{COURSE})) \bowtie_{\text{Course_ID}} \text{ENROLLS_IN}) \bowtie_{\text{Student_ID}} \text{STUDENT}$$

In contrast, the **optimized query expression** applies the selection condition as early as possible by first filtering **COURSE** to a single tuple corresponding to the desired course. This approach helps database professionals feel motivated to adopt early filtering strategies for better performance. By isolating the relevant course at the very beginning, the query eliminates all unrelated course records before any join operations are performed. This early filtering significantly reduces the relation size, thereby drastically decreasing the size of intermediate results when joined with the **ENROLLS_IN** table. This join significantly reduces the size of the intermediate result by retaining only enrollment records associated with the selected course. Finally, this reduced enrollment set is joined with the **STUDENT** relation using the **Student_ID** attribute. Because the input to this join is already small, the database system can efficiently retrieve the corresponding student records without generating large intermediate relations. This optimized execution plan minimizes memory usage, disk I/O, and CPU overhead, clearly demonstrating how early filtering and careful operation ordering can dramatically improve query performance and scalability.

The complete optimized relational algebra expression is:

$$\pi_{\text{Student_ID, First_name, Last_name, Course Title}}(((\sigma_{\text{Title='Intro to Programming'}}(\text{COURSE})) \bowtie_{\text{Course_ID}} \text{ENROLLS_IN}) \bowtie_{\text{Student_ID}} \text{STUDENT})$$

Table 8.59. Results of University students enrolled in "Intro to Programming."

Student_ID	*First_name*	*Last_name*	*Course Title*
S006	Anica	Stojnasen	Intro to Programming

The optimizer employs dynamic programming or randomized search algorithms to explore join orders and uses cost models (based on disk I/O, CPU cycles, and memory) to estimate the cheapest execution.

Company System Example

This example demonstrates how Operation Ordering, specifically Selection Pushing, significantly impacts performance by reducing the size of intermediate results during joins. In relational database management systems, performing a selection (filtering) as early as possible is a fundamental optimization technique.

req8.57 *Find the names and quantities of all products ordered by the customer 'Ivanovska'.*

In this case, multiple relations from the Company system relational model must be combined, including CUSTOMER, ORDER, ORDER_LINE, and PRODUCT. Although the logical requirements of the query are straightforward, the execution efficiency depends critically on the sequence in which selection and join operations are applied. Recognizing the importance of join sequence can empower database professionals to manage complex data relationships more effectively, leading to more efficient query execution.

- **Inefficient Expression (first Join, then Select):**

In the **inefficient approach**, the database system first joins all four relations—CUSTOMER, ORDER, ORDER_LINE, and PRODUCT—to construct an intermediate relation that contains every customer, all their orders, the associated order lines, and the corresponding product information. Only after this large intermediate result is materialized is the selection condition applied to filter tuples where the customer's last name is 'Ivanovska':

$$\sigma_{\text{Last_name = 'Ivanovska'}}(\text{CUSTOMER} \bowtie \text{ORDER} \bowtie \text{ORDER_LINE} \bowtie \text{PRODUCT})$$

This strategy is highly inefficient because it forces the system to process and store a vast amount of irrelevant data before eliminating almost all of it. The joins generate large temporary relations, leading to excessive CPU usage, memory consumption, and disk I/O.

- **Optimized Expression (first Select, then Join):**

In contrast, the **optimized approach** applies the selection operation as early as possible:

$$((\sigma_{\text{Last_name = 'Ivanovska'}}\text{CUSTOMER}) \bowtie \text{ORDER}) \bowtie \text{ORDER_LINE} \bowtie \text{PRODUCT}$$

By filtering the **CUSTOMER** relation first, the system reduces it to a single tuple corresponding to the customer Ivanovska. Subsequent joins are therefore performed using a drastically smaller input, significantly reducing the number of tuple comparisons. Since join operations are computationally expensive—often approaching $O(n \times m)$ in the worst case—reducing the number of participating tuples early yields substantial savings in processing time and memory usage. This example clearly illustrates how selection pushdown and proper join ordering lead to more efficient and scalable query execution.

The complete optimized relational algebra expression is:

$$\pi_{\text{Product_name, Quantity}}(((\sigma_{\text{Last_name = 'Ivanovska'}}\text{CUSTOMER}) \bowtie \text{ORDER}) \bowtie \text{ORDER_LINE} \bowtie \text{PRODUCT})$$

Results of the names and quantities of all products ordered by the customer 'Ivanovska' (Customer **C1**) has two orders: **Order1** and **Order8**, are show in Table 8.60.

Table 8.60. Results of Company system for products ordered by the customer 'Ivanovska'

Product_name	***Quantity***
Ajvar	2
Filigree	5

8.8.4 Performance Selection and Projection Pushdown Principles

The principle of selection and projection pushdown is one of the most fundamental and widely applied optimization techniques in relational database query processing, highlighting its critical role in query optimization. Its core objective is to reduce the volume of data flowing through a query execution plan by filtering tuples and eliminating unnecessary attributes as early as possible in the execution sequence (Silberschatz et al., 2019). Instead of allowing large intermediate relations to propagate through costly operations such as joins, aggregations, and sorting before applying restrictions, modern query optimizers push selection and projection operations closer to the data source. This transformation converts a data-intensive execution strategy into an efficient pipeline that processes only relevant data from the outset (Date, 2003). The benefits of pushdown optimization are cumulative: reducing intermediate result sizes lowers memory usage, minimizes disk input/output, and significantly decreases CPU overhead in subsequent operations (Garcia-Molina et al., 2008). For example, in a query involving multiple joins, applying a highly selective predicate to the largest relation before joining can reduce the number of processed tuples by orders of magnitude, leading to substantial performance gains (Elmasri & Navathe, 2016). Early projection also contributes by removing unnecessary attributes, reducing tuple width, and improving cache utilization and data transfer efficiency (Date, 2019). In addition, pushdown strategies enable the optimizer to better leverage physical storage structures, such as indexes and data partitions, often replacing full-table scans with more efficient indexed access paths (Silberschatz et al., 2019). These techniques are vital components of cost-based query optimization and directly impact database system performance (Connolly & Begg, 2015).

Why it works: Reducing the number of rows *before* performing expensive operations like joins or sorting drastically shrinks the dataset that needs to be processed.

Selection Pushdown (σ)

Selection pushdown operates on the principle that filtering operations should execute as close as possible to the initial data access point—essentially at the bottom of the query execution tree, where data first emerges from storage. This positioning allows the database to apply predicates immediately after reading data pages from disk, fundamentally altering what enters the processing pipeline. The immediate benefits are threefold: reduced physical I/O as fewer data pages require loading into memory, dramatically smaller intermediate result sets passed to expensive downstream operations, and critically, the opportunity for index utilization. When selection predicates are pushed down, the optimizer can determine whether indexes exist on filtered columns and substitute costly full-table scans with efficient

index seeks. A filter on a student's age, for instance, becomes vastly more efficient when applied via an index at the storage layer rather than scanning and filtering the entire table in memory.

University System Example

req8.58 *Find all professors from the Computer Science faculty, hired before 2010.*

The relational algebra must combine information from the **PROFESSOR** and **FACULTY** relations. Although the logical conditions of the query are simple, the efficiency of its execution depends heavily on where the selection predicates are applied within the query plan.

- **Inefficient Plan (Filter after Join):**

In the **inefficient execution plan**, the database system first performs a join between **PROFESSOR** and **FACULTY**, producing an intermediate relation that includes all professors along with their associated faculty information. This intermediate result may contain a large number of tuples, many of which are irrelevant to the final query result. Only after this costly join operation are the selection conditions applied—first filtering tuples for *Faculty_ID = 'F001'* (Computer Science), and then further restricting the result to professors hired before *January 1, 2010*:

$$\sigma_{\text{Faculty_ID='F001'} \wedge \text{Hire_date} < \text{'2010-01-01'}} (\textbf{PROFESSOR} \bowtie \textbf{FACULTY})$$

This approach wastes computational resources by joining many tuples that will ultimately be discarded, increasing memory usage, CPU cost, and disk I/O.

- **Optimized Plan (Selection Pushdown):**

In the **optimized execution plan**, the selection predicates are pushed down and applied directly to the **PROFESSOR** relation before any join is performed:

$$(\sigma_{\text{Faculty_ID='F001'} \wedge \text{Hire_date} < \text{'2010-01-01'}} \textbf{PROFESSOR}) \bowtie \textbf{FACULTY}$$

By filtering **PROFESSOR** first—potentially using indexes on *Faculty_ID* or *Hire_date*—the number of tuples is reduced dramatically, for example from 100 professors to only two or three. The subsequent join with **FACULTY** is therefore executed on a much smaller input, significantly reducing processing time and resource consumption. This example clearly demonstrates how selection pushdown improves query performance by eliminating irrelevant data as early as possible.

The complete optimized relational algebra expression is:

$$\pi_{\text{First_name, Last_name, Hire_date, Faculty_ID}} (\sigma_{\text{Faculty_ID='F001' U A2 Hire_date} < \text{'2010-01-01'}} \textbf{PROFESSOR}) \bowtie \textbf{FACULTY}$$

Table 8.61. Results of University professors from the Computer Science faculty hired before 2010.

First_name	*Last_name*	*Hire_date*	*Faculty_ID*
Kliment	Ohridski	01.08.2007	F001
Metodij	Prosvetitel	13.09.2002	F001

Company System Example

req8.59 *Find employees who are handling 'Pending' orders*

The relational algebra must combine information from the **EMPLOYEE** and **ORDER** relations. The **EMPLOYEE** relation stores employee details, while the **ORDER** relation records customer orders and their current processing status. Although the query's logical requirement is straightforward, its execution efficiency depends heavily on where the selection condition on the order status is applied.

- **Inefficient Plan (Filter after Join):**

In the **inefficient execution plan**, the database system first performs a join between **EMPLOYEE** and **ORDER**, producing an intermediate relation that includes every employee paired with their associated orders, regardless of order status. Only after this potentially large intermediate result is generated is the selection condition applied to retain orders with *Status = 'Pending'*:

$$\sigma_{\text{Status = 'Pending'}}(\text{EMPLOYEE} \bowtie \text{ORDER})$$

This approach is inefficient because it forces the system to process many tuples that do not satisfy the selection condition. As a result, unnecessary comparisons are performed, memory consumption increases, and additional disk I/O may be required to store large intermediate results, most of which are eventually discarded.

- **Optimized Plan (Selection Pushdown):**

In contrast, the **optimized execution plan** applies the selection operation directly to the **ORDER** relation before performing the join:

$$\text{EMPLOYEE} \bowtie (\sigma_{\text{Status = 'Pending'}}\ \text{ORDER})$$

By filtering the **ORDER** relation first, only orders with a *'Pending'* status participate in the join. This significantly reduces the number of tuples involved, especially if pending orders represent a small fraction of all orders. The subsequent join with **EMPLOYEE** is therefore faster and more resource-efficient.

The complete optimized relational algebra expression is:

$$\pi_{\text{Employee_ID, First_name, Last_name, Order_ID, Status}}(\text{EMPLOYEE} \bowtie (\sigma_{\text{Status = 'Pending'}}\ \text{ORDER}))$$

This example clearly illustrates how selection pushdown improves query performance by eliminating irrelevant data early in the execution process. There are only two orders with a 'Pending' status: **Order3** and **Order9**, as shown in Table 8.62.

Table 8.62. Results of Company system to identify employees are handling 'Pending' orders

Employee_ID	*First_name*	*Last_name*	*Order_ID*	*Status*
E1	Zoran	Pejovski	Order9	Pending
E3	Marija	Naumoska	Order3	Pending

Projection Pushdown (π)

Projection pushdown extends optimization beyond row filtering to column elimination, removing unnecessary attributes from relations as soon as they are not needed in the query plan. By performing projections before resource-intensive operations like joins, the database reduces the width of records flowing through the execution pipeline, yielding substantial performance improvements. Narrower records consume less memory in temporary buffers, reduce the data volume transferred across networks in distributed architectures, and accelerate operations that depend on record width—particularly sorting and hash-based join algorithms. For example, in distributed data warehouses, early projection minimizes network transfer costs by sending only relevant columns across nodes. When a query requires only three columns from a twenty-column table, early projection means subsequent join operations process perhaps one-sixth the data volume, with proportional improvements in speed and resource utilization. This technique proves especially valuable for analytical queries that access wide tables, where most columns remain unused.

University System Example

req8.60 *Find the names and grades of all students in the course CS303.*

The relational algebra must combine information from the **STUDENT** and **ENROLLS_IN** relations. The **STUDENT** relation typically contains many attributes—such as street address, city, date of birth, and contact details—most of which are irrelevant to the final query result. The efficiency, therefore, depends on whether these unnecessary attributes are carried through expensive operations such as joins.

- **Inefficient Plan (Projection late):**

In the **inefficient execution plan**, the database system first performs a join between **STUDENT** and **ENROLLS_IN**, producing a large intermediate relation that includes all student attributes along with enrollment data. Only after this join does the system apply the selection condition to restrict the results to *Course_ID = 'CS303'*, followed by a projection to retain only the student's first name, last name, and grade:

$$\pi_{\text{S.Student_ID, S.First_name, S.Last_name, E.Course_ID, E.Grade}} (\sigma_{\text{Course_ID='CS303'}} (\text{STUDENT} \bowtie \text{ENROLLS_IN}))$$

This approach is inefficient because large and irrelevant columns—such as *Street*, *City*, and *Birth_date*—are unnecessarily propagated through the join, increasing memory usage, disk I/O, and processing time.

- **Optimized Plan (Projection Pushdown):**

In the **optimized execution plan**, **projection pushdown** is applied to remove unneeded attributes as early as possible. Instead of joining the full tables and then projecting, we project the specific columns needed from each table *before* the join occurs.

- **Projection (π):** In relation R1 project only the attributes required for the join and final output are retained from **STUDENT**:

$$\text{R1} = \pi_{\text{Student_ID, First_name, Last_name}} (\text{STUDENT})$$

- **Selection (**σ): In relation R2 the **ENROLLS_IN** relation is filtered for *CS303* and projected to keep only the student identifier and grade:

$$R2 = \pi_{Student_ID,\ Grade}\ (\sigma_{Course_ID='CS303'}\ (ENROLLS_IN))$$

- **Join (⋈):** Finally, in relation **R3** join is performed on these reduced relations, yielding a small and efficient intermediate result:

$$R3 = R1 \bowtie_{Student_ID} R2$$

The complete optimized expression is:

$$\pi_{First_name,\ Last_name,\ Grade}\ (STUDENT \bowtie_{Student_ID}\ (\sigma_{Course_ID='CS303'}\ (ENROLLS_IN)))$$

This example demonstrates how projection pushdown improves performance by reducing tuple width early, leading to more efficient joins and overall query execution.

Table 8.63. Results of the University names and grades of all students in the course CS303.

First_name	*Last_name*	*Grade*
Sarah	Smith	9
Anica	Stojnasen	6

Company System Example

Projection pushes down the required columns (and the necessary join keys, like IDs) onto the individual tables *before* the join, thus narrowing the relations that are being joined.

req8.61 *List the products by warehouse only in case the Quantity_on_hand is bigger than three times the Reorder_level in the STORED_IN relation.*

To identify products in stock that significantly exceed their reorder levels, we use Projection Pushdown to reduce the size of the data stream before joining the product descriptions with the inventory data.

- **Inefficient Plan (Projection late):**

In this plan, the database engine joins the entire STORED_IN table with the entire PRODUCT table (carrying over columns like Price, Weight, Description, and Reorder_level) before finally filtering the rows and selecting the relevant columns.

$$\pi_{Product_name,\ Warehouse_ID,\ Quantity_on_hand,\ Reorder_level}$$
$$(\sigma_{Quantity_on_hand>3\times Reorder_level}(STORED_IN \bowtie PRODUCT))$$

- **Optimized Plan (Projection Pushdown):**

This plan "pushes" the selection and projection operations down to the individual tables. Selection and Projection Pushdown on STORED_IN filters inventory first and keeps only the necessary attributes (Product_ID, Warehouse_ID, Quantity_on_hand, and Reorder_level). Projection Pushdown on PRODUCT with attributes Product_ID and Product_name from the other side, finally, after that, joins the relations.

$$\pi_{Product_name,\ Warehouse_ID,\ Quantity_on_hand,\ Reorder_level}\ ((\pi_{Product_ID,\ Warehouse_ID,\ Quantity_on_hand,\ Reorder_level}$$
$$(\sigma_{Quantity_on_hand>3\times Reorder_level}\ STORED_IN)) \bowtie (\pi_{Product_ID,\ Product_name}\ PRODUCT))$$

Table 8.64. Results of the company system products by warehouse, only in case the Quantity_on_hand is bigger than three times the Reorder_level.

Product_name	*Warehouse_ID*	*Quantity_on_hand*	*Reorder_level*
Ajvar	W1	57	10
Tavce Gravce	W1	21	6
Rakija	W3	18	4
Filigree	W3	6	1
Vranec Wine	W2	72	10

8.9 Summary

Relational algebra is an essential procedural query language that serves as the foundation for all relational database systems. It includes a core set of operations such as selection (σ), projection (π), join (⋈), and division (÷). These operations encourage practice in writing queries that effectively utilize these concepts. Set-theoretic operations, based on fundamental mathematical set theory, include union, intersection, and difference for combining or comparing relations, along with the Cartesian product for generating all possible tuple combinations. Binary operations like join and division are essential for combining related data across tables, forming the core of relational query processing. The chapter also covered aggregate functions such as count, sum, average, minimum, and maximum, which are often used with grouping to produce statistical insights from large datasets.

Beyond cataloging operations, the chapter emphasized relational algebra's role as a universal language for describing database operations, transcending vendor-specific implementations and providing a common vocabulary for designing, comparing, and analyzing database systems. This universality proves invaluable for understanding how query optimizers transform high-level SQL statements into efficient execution plans. A crucial mathematical property highlighted throughout is that relational algebra operations always produce relations as output, enabling complex queries to be constructed by composing simpler operations in a modular fashion. The chapter reinforced these theoretical concepts through progressively challenging practice problems, ranging from simple single-operation queries to complex multi-step transformations, each accompanied by complete solutions and explanations that illustrate both the mechanical application of operators and the strategic reasoning behind operation ordering.

The methodologies and principles presented in this chapter provide the basis for user-friendly database query languages that implement relational algebra operations. When you write a SELECT statement with JOINs and WHERE clauses, you are using **relational algebra** concepts like selection, projection, and joins. This connection between theory and practice establishes relational algebra not merely as academic knowledge but as essential conceptual infrastructure that enables database professionals to think precisely about data operations, optimize query performance, and design more efficient database schemas that anticipate common access patterns and query requirements.

Review Questions

8.1. List and briefly describe all relational algebra operation categories covered in this chapter. Distinguish between unary operations (Selection, Projection, Rename, Distinct), set operations (Union, Intersection, Set Difference, Cartesian Product), binary relational operations (Natural Join, Equijoin, Theta Join, Outer Join, Division), and aggregate/sorting operations. For each category, name the operations and explain what type of input relation(s) each requires.

8.2. Union Compatibility Requirements: a) Define union compatibility: what conditions must two relations satisfy for set operations (Union, Intersection, Set Difference) to be applied? b) Provide one example of two university-domain relations that are union-compatible and one example of two relations that are not, explaining your reasoning in terms of attribute count and domain compatibility. c) Does union compatibility imply that attribute names must be identical? Why or why not?

8.3. Selection vs. Projection Distinction: a) Clearly distinguish the Selection operation from the Projection operation in relational algebra. b) How does Selection filter a relation (which dimension – rows or columns) compared to Projection? c) What effect does each operation have on the schema (attributes) and cardinality (number of tuples) of the result relation? d) Provide a relational algebra expression using the University STUDENT relation that applies both a Selection and a Projection, and describe the result.

8.4. Join Operations Comparison: a) Compare and contrast the Cartesian Product, Natural Join, Equijoin, and Theta Join operations. b) For each operation, describe the join condition (if any), the schema of the result, and the typical size of the result relation relative to the input relations. c) Under what circumstances is each join type most appropriate? d) Provide a concrete example where using a Cartesian Product instead of a Natural Join would produce incorrect or unmanageably large results.

8.5. Division Operation Application: a) Describe what the Division operation computes b) Provide a realistic business scenario where Division is the most appropriate operation (e.g., finding all students enrolled in ALL required courses, or all employees who have used ALL pieces of company equipment). c) Explain why Division is considered a derived operation and show how it can be expressed using Projection, Cartesian Product, and Set Difference.

8.6. Aggregate Functions with Grouping: a) Explain how aggregate operations differ from other relational algebra operations: in what sense do they not return a relation in the strict relational model? b) List the five standard aggregate functions (COUNT, SUM, AVG, MIN, MAX) and describe what each computes. c) Provide a complete relational algebra expression with grouping that answers the following question on the University database: “For each faculty, find the number of affiliated professors and the average number of years since hire.”.

8.7. Sorting operations: a) Explain the Sorting operation in relational algebra: what does it add to a relation that the relational model does not otherwise guarantee? b) Write a

complete relational algebra expression to compute grade statistics by course from the University database, producing Course_ID, Title, Credits, and the average grade per course, sorted by average grade in descending order. c) Can the result of a sort operation be used as input to another set operation, such as Union? Explain why or why not.

8.8. Multi-step query: a) What is the primary benefit of decomposing a complex query into discrete intermediate relational algebra operations? How does this relate to query optimization? b) Write a multi-step relational algebra expression to find University students (First_name, Last_name, Course_Title, Credits, Faculty_ID) who have a recorded attendance status of 'Present'. Label each intermediate result. c) Write a multi-step relational algebra expression for the Company database to find employees (Employee_ID, First_name, Last_name, City) whose orders have a status of 'Canceled'.

8.9. Selection Pushdown Optimization: a) Explain the principle of Selection Pushdown: why does applying selection operations as early as possible in the query execution plan significantly improve performance? b) Write a relational algebra expression for the Company database to find employees who live in 'Skopje' and have a salary greater than 50,000. First write the naïve (unoptimized) expression, then rewrite it with selection pushdown applied, labeling the intermediate result at each step. c) Quantify the performance benefit conceptually: if EMPLOYEE has 10,000 rows and only 200 live in Skopje, how does pushdown reduce intermediate result size?

8.10. Projection Pushdown Benefits: a) Describe three specific performance benefits of applying Projection as early as possible in the query execution plan (projection pushdown). b) Why is early projection particularly important in distributed database systems, where intermediate results may need to be transmitted across a network? c) Provide a relational algebra example using both selection and projection pushdown on a multi-table University query, showing the optimized order of operations.

8.11 Query Optimization Strategy: a) Given a query joining four tables (STUDENT, ENROLLS_IN, COURSE, FACULTY) with predicates age > 20 and Faculty name = 'Humanities', write the relational algebra expression applying all applicable optimization principles (selection pushdown, projection pushdown, join ordering). b) Justify each operation ordering decision based on intermediate result size reduction and index utilization potential. c) Identify which intermediate result is likely to be smallest, and explain how this knowledge should inform join ordering.

8.12. a) Describe three common mistakes students and developers make when writing relational algebra expressions or SQL queries that violate optimization principles. For each mistake, provide a concrete example and explain the performance consequence. b) Summarize the specific principles of selection pushdown and projection pushdown, distinguishing between their mechanisms and the types of cost they reduce. c) How do these two pushdown principles interact when both are applied to the same query?

■ ■ ■ ■ ■ ■

Exercises

8.1. Relational Model Example – Library Management System: Consider the following requirements for a Library Management System: "A library has books, members, and staff.

Each book has an ISBN, title, author, publication year, and category. Members have a membership ID, name, address, phone numbers (multivalued), and join date. Staff members have employee IDs, names, positions, and hire dates." a) Identify all relations in this description. b) Determine the key attributes for each relation. c) Write the relational model with entities and their attributes. d) Write relational algebra expressions to list all books in the 'Fiction' category, and to project the ISBN, title, and author columns from the BOOK relation.

8.2. Relational Algebra Analysis – Library Borrowing: Using the Library Management System from Exercise 8.1, and the additional requirements: "Members borrow books. A member can borrow multiple books, and each book can be borrowed by different members over time. Each borrowing transaction must be handled by exactly one staff member." a) Identify the participating (associative) entity and write the relational model. b) Write the relational model specifying cardinality for each relationship. c) Write a relational algebra expression using a Natural Join between BORROWING, MEMBER, and BOOK to list all current borrowings, including the member's name and the book title. d) Apply projection to return only Member_name and Book_title in the final result.

8.3. Weak Entity Recognition – Hospital Management System: Consider the Hospital Management System requirement: "A hospital has patients and rooms. Each patient can have multiple visits. Each visit belongs to exactly one patient and cannot exist without a patient. For each visit, we track the visit number, admission date, discharge date, and diagnosis." a) Write the relational model of the Hospital Management System, identifying VISIT as a weak entity. b) Identify the partial key for the weak entity. c) Write the relational model of the weak entity with its identifying relationship. d) Write a relational algebra expression using Set Difference to find all patients who have had at least one visit (i.e., appear in the VISIT relation), and using Selection to find all visits with a diagnosis of 'Pneumonia'.

8.4. Cardinality and Participation – Relational Algebra Constraints: For each of the following business scenarios, write the relational model with appropriate constraints and provide a relational algebra expression demonstrating the relationship: a) "Each employee works for exactly one department, and each department has multiple employees." Write a Natural Join expression to list all employees with their department name. b) "Students can enroll in multiple courses, and courses can have multiple students enrolled." Write a Projection on the enrollment relation to list unique Student_ID and Course_ID pairs. c) "Every order must be placed by exactly one customer, but customers may not have placed any orders yet." Write a Left Outer Join to list all customers, including those without orders. d) "An employee may manage at most one department, but not every employee is a manager." Write a Selection to find all employees whose Employee_ID appears as a Manager_ID in the DEPARTMENT relation.

8.5. Online Bookstore System – Relational Model and Algebra: Write the relational model for an Online Bookstore System with the following requirements: "The bookstore sells books to customers. Customers have IDs, names, email addresses, and shipping addresses. Books have ISBNs, titles, authors, prices, and stock quantities. Customers place orders containing multiple order items. Each order item references one book. Each book is published by one publisher." a) Identify all entities (strong and weak) and write the relational

model. b) Write relational algebra expressions: use Selection and Projection to list books with a stock quantity less than 5, showing only ISBN, title, and stock quantity; use Natural Join to find all orders with the customer name and book titles ordered; use Set Difference to find customers who have never placed an order. c) Write the relational model examples with the entities and their attributes.

8.6. Restaurant Management System – Relational Model and Algebra: Write the relational model for the Restaurant Management System: "The restaurant has tables, customers, and menu items. Customers make reservations for tables. Menu items have names, descriptions, categories, and prices. Customers order menu items during their visit." a) Identify all entities, classify as strong or weak, and write the relational model. b) Write relational algebra expressions: use Selection to find all reservations for a party size greater than 4; use Projection on the RESERVATION relation to list reservation date, time, and table number, sorted by reservation date descending; use Natural Join to list orders with the customer name and menu item name. c) Write the relational model examples with the entities and their attributes.

8.7. University Course Scheduling System – Relational Model and Algebra: Write the relational model for the University Course Scheduling System: "The university offers courses taught by professors in classrooms during time slots. A course can have multiple sections, each taught by one professor in one classroom. Students enroll in specific course sections." a) Identify and handle the complex many-to-many-to-many relationships; consider which entities should be weak. b) Write relational algebra expressions: use Natural Join to find all course sections with professor name, classroom, and time slot; use Selection and Projection to list students enrolled in a specific course (e.g., Course_ID = 'CS101'), showing Student_ID and Section_ID; use Division to find students enrolled in ALL sections of a specific course. c) Write the relational model with entities and their attributes.

8.8. Hospital Management System (Extended) – Relational Model and Algebra: Write the relational model for the extended Hospital Management System with doctors, nurses, patients, departments, admissions, treatments, and equipment. a) Identify all relations and their key attributes. b) Write relational algebra expressions: use Natural Join to find all admissions with patient name and treating department name; use Selection to find all patients admitted since January 1 of the current year, then apply Projection to return only Patient_name, Admission_date, and Diagnosis; use Set Difference to find departments that have doctors but no assigned equipment; use Natural Join and Selection to list all treatments performed using MRI equipment (Equipment name = 'MRI Scanner'), showing doctor name, patient name, treatment date, and procedure. c) Write the relational model examples with the entities and their attributes.

8.9. E-Commerce Platform – Relational Model and Algebra: Write the relational model for the E-Commerce Platform connecting customers, sellers, products, orders, and warehouses. a) Identify all relations and their key attributes. b) Write relational algebra expressions: use Natural Join to find all shipments with the seller name, warehouse location, and product names shipped; use Selection and Projection to find all products stored in a warehouse with location 'Skopje', showing Product_name, Category, and Stock_quantity; use Intersection to find product IDs that appear in both the PRODUCT relation and the SHIPMENT_ITEM relation (i.e., products that have been shipped at least once); use

Aggregation to find the total quantity shipped per product, grouped by Product_ID. c) Write the relational model examples with the entities and their attributes.

8.10. Airline Reservation System – Relational Model and Algebra: Write the relational model for the Airline Reservation System with airports, aircraft, flights, passengers, reservations, tickets, crew, and frequent flyer programs. a) Identify all relations and their key attributes. b) Write relational algebra expressions: use Natural Join to find all flight instances with departure airport name, arrival airport name, scheduled departure time, and aircraft model; use Selection and Projection to list all passengers on a flight with Flight_number = 'MK101' for a specific date, showing Passenger_name, Seat_number, and Fare_class; use Natural Join and Aggregation to generate a frequent flyer report showing Passenger_name, Membership_number, Tier_level, and Total_miles, sorted by Total_miles descending; use Division to find passengers who have flown on ALL aircraft types operated by the airline. c) Write the relational model examples with the entities and their attributes.

References and Recommended Readings

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM, 13*(6), 377-387.

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of Database Systems* (7th ed.). Pearson.

Hristoski, I., & Dimovski, T. (2022). *Dizajn na relaciski bazi na podatoci, Tom 1* [Design of Relational Databases, Volume 1]. University "St. Kliment Ohridski", Bitola. (In Macedonian)

Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database Systems: The Complete Book* (2nd ed.). Prentice Hall.

Date, C. J. (2003). *An Introduction to Database Systems* (8th ed.). Addison-Wesley.

Practical Design Methodologies

Batini, C., Ceri, S., & Navathe, S. B. (1992). *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings.

Simsion, G. C., & Witt, G. C. (2005). *Data Modeling Essentials* (3rd ed.). Morgan Kaufmann.

Connolly, T. M., & Begg, C. E. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson.

Date, C. J. (2019). *Database Design and Relational Theory: Normal Forms and All That Jazz* (2nd ed.). O'Reilly Media.

Codd, E. F. (1990). *The Relational Model for Database Management: Version 2*. Addison-Wesley.

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database System Concepts* (7th ed.). McGraw-Hill.

Chapter 9

9. SQL Implementation

9.1 Introduction: The Information Age Resolution

SQL (Structured Query Language) is the dominant language for querying relational databases, enabling efficient retrieval, manipulation, and management of data. For example, when an academic advisor needs to identify all students enrolled in "Database Systems," SQL can be used to construct a multi-table query spanning three interconnected tables. The process includes querying the COURSE table to obtain the course identifier, querying the ENROLLS_IN table to retrieve enrollment records for the course, and referencing the STUDENT table to obtain the names and details of enrolled students. This example demonstrates SQL's capacity to join relational data and generate meaningful insights. By implementing SQL, abstract data relationships are transformed into useful information for academic reporting, business intelligence, and administrative decision-making. This chapter discusses how SQL queries facilitate the transition from raw data storage to practical information retrieval in real-world contexts.

This scenario highlights the role of databases as the underlying infrastructure of contemporary society. Online searches, purchases, banking transactions, flight bookings, and social media interactions all depend on database systems. Students encounter databases daily when they register for courses, check their grades, or access university portals for various academic activities. Universities manage records for millions of students. Corporations oversee complex supply chains, customer relationships, and financial operations. Hospitals maintain patient histories and coordinate care across departments. Governments process tax records, issue licenses, and administer public services (Elmasri & Navathe, 2016).

This chapter examines how SQL acts as the foundation of modern database systems. We start with the basic ideas behind SQL, then show how it works in practice as the standard way to interact with databases. Using examples from University and Company databases, we see how SQL connects the design of a database to its real-life use. SQL lets us define data, update it, ask questions, and keep information secure, regardless of the database system we use.

9.2 Why SQL Matters for Databases

SQL (Structured Query Language) is an effective tool for DBMS that can extract the most suitable information in the information age. Its significance lies in providing a standardized, declarative approach to database operations that abstracts away the complexities of data storage, retrieval, and manipulation. Unlike procedural programming languages that require users to specify how operations are executed, SQL allows users to declare what data is needed, leaving execution strategy decisions to the database management system. This declarative property enables database systems to apply sophisticated query optimization techniques transparently, making SQL both powerful and efficient (Elmasri & Navathe, 2016).

The progression from conceptual database design to a working SQL-based database follows a structured, multi-level abstraction process that can make the process appear clearer and more manageable. It begins with requirements analysis, where business needs and rules are identified, and continues with conceptual modeling using Entity–Relationship (ER) diagrams. ER modeling captures entities, attributes, and relationships at a high level, focusing on data semantics rather than implementation details, thereby facilitating communication among technical and non-technical stakeholders (Connolly & Begg, 2015).

The conceptual model is then transformed into a logical design grounded in the relational model, originally introduced by Codd in 1970. In this phase, entities become relations, attributes become typed columns, and relationships are implemented using primary and foreign keys. The relational model delivers basic concepts such as normalization, functional dependencies, and referential integrity, providing a strict theoretical basis for database design (Date, 2003; Elmasri & Navathe, 2016). This foundation can help users feel more confident in their understanding of database reliability and correctness.

SQL serves as the critical implementation bridge, translating logical designs into physical database structures using Data Definition Language (DDL) commands. The mapping from logical to physical includes several essential transformations: entity sets become database tables created with CREATE TABLE statements; attributes map to columns with appropriate data types (VARCHAR for text, INT for integers, DATE for temporal data, DECIMAL for exact numeric values); primary keys are enforced through PRIMARY KEY constraints providing unique identification of rows; and foreign keys implement relationships through FOREIGN KEY constraints that maintain referential integrity by assuring such values in one table correspond to valid values in referenced tables. This systematic progression from conceptual to logical to physical design confirms that databases are built on solid basic principles while remaining practical, performant, and maintainable.

> **Definition: SQL (Structured Query Language)**
>
> SQL is a standardized programming language used to manage, query, and manipulate data stored in relational database management systems.

This definition shows several important SQL characteristics:

SQL is fundamentally a declarative language, meaning users specify what data they want rather than how to retrieve it. When writing a SQL query, you describe the desired outcome, such as "retrieve all students enrolled in Database Systems", without detailing the step-by-

step procedures for accessing storage structures, navigating indexes, or joining tables. The database management system's query optimizer handles these implementation details automatically, determining the most efficient execution path. This declarative approach considerably simplifies database queries, letting users focus on business logic rather than technical complexity. Even complex operations involving multiple tables, aggregations, and filtering conditions can be expressed concisely through declarative statements, making SQL accessible to both technical developers and non-technical analysts who need to gain meaningful insights from organizational data.

It operates as an international standard maintained by ANSI (American National Standards Institute) and ISO (International Organization for Standardization), ensuring consistency among different database platforms. This standard process means that core SQL syntax remains largely uniform across Oracle, MySQL, PostgreSQL, Microsoft SQL Server, and other relational database systems. Consequently, database professionals can transfer their SQL knowledge across multiple platforms with minimal retraining, and organizations can migrate between database vendors without completely rewriting application code. While vendor-specific extensions and proprietary features exist, the core SQL standard provides a common language that promotes interoperability, reduces vendor lock-in, and facilitates collaboration between database professionals worldwide. This portability keeps SQL skills universally valuable across industries and technologies.

SQL provides a comprehensive set of operations for managing the entire data lifecycle in relational databases. Beyond simple data retrieval (SELECT statements), SQL provides Data Definition Language (DDL) commands for creating and modifying database structures such as tables, indexes, and views; Data Manipulation Language (DML) commands for inserting, updating, and deleting records; Data Control Language (DCL) for managing user permissions and security; and Transaction Control Language (TCL) for guaranteeing data integrity through commit and rollback operations. This complete functionality means SQL serves as a unified interface for all database transactions, starting with initial schema design through ongoing maintenance, querying, reporting, and administration. Database professionals rely on SQL not simply as a query tool but as a complete database management solution that handles everything from structural definition to security enforcement and transactional consistency.

9.3 Introduction to SQL

9.3.1 History of SQL

SQL's origins go back to the early 1970s at IBM's San Jose Research Laboratory (now IBM Almaden Research Center), where Donald D. Chamberlin and Raymond F. Boyce developed SEQUEL (Structured English Query Language) as a data manipulation language for IBM's experimental System R database management system. System R was a landmark research project that demonstrated the practical feasibility of relational database systems based on Edgar F. Codd's relational model, first introduced in 1970 (Date, 2003). Due to trademark considerations, SEQUEL was later renamed SQL, marking the start of its path toward becoming a global standard.

The first commercial implementation of SQL appeared in 1979, when Relational Software, Inc. (later Oracle Corporation) released Oracle V2, which preceded IBM's commercial offerings and established SQL as the dominant interface for relational databases (Garcia-Molina et al., 2008). The growing adoption of SQL led to its official standardization, beginning with the publication of the first SQL standard by the American National Standards Institute (ANSI) in 1986, followed by adoption by the International Organization for Standardization (ISO) in 1987 (Date, 2003).

Subsequent revisions significantly expanded SQL's expressive power and scope. SQL-92 introduced substantial enhancements to data definition and query capabilities, while later standards incorporated object-relational features, triggers, recursion, and developed analytical functionality. More recent versions added support for XML, JSON, temporal data management, and graph-oriented queries, representing the evolving requirements of modern data-driven applications (Elmasri & Navathe, 2016; Silberschatz et al., 2019). Throughout its evolution, SQL has maintained backward compatibility while evolving with new paradigms in database management, ensuring its ongoing importance in both academic and industrial areas.

9.3.2 SQL Importance and Database Implementations

SQL's importance in database management resides in its role as a standardized, declarative interface that enables developers, data analysts, and database administrators to interact with relational database systems using a common language. This standardization facilitates portability of skills, decreases training costs, and allows applications to be developed independently of specific database vendors, while still benefiting from relational theory and optimization mechanisms provided by database management systems (Elmasri & Navathe, 2016; Date, 2003). Although SQL standards establish a common foundation, individual database systems implement SQL with varying levels of compliance and often extend the language with proprietary features adapted to specific environments and use cases (Silberschatz et al., 2019).

Commercial enterprise systems, such as Microsoft SQL Server, exemplify this approach by largely following ANSI SQL standards while extending functionality through vendor-specific languages, such as Transact-SQL (T-SQL). These extensions support advanced querying, transaction management, and tight integration with enterprise application ecosystems, enabling SQL Server to serve as a complete platform for large-scale and mission-critical applications (Silberschatz et al., 2019). In contrast, MySQL represents a widely accepted open-source database system that prioritizes simplicity, performance, and ease of deployment, particularly in web-based and lightweight application environments. Although historically more limited in advanced SQL features, modern versions of MySQL increasingly support core SQL constructs, narrowing the gap with enterprise systems (Elmasri & Navathe, 2016).

The existence of diverse SQL implementations shows varying architectural goals, performance requirements, and business models. Enterprise-oriented systems value robustness, scalability, and advanced analytical functions, whereas open-source systems stress accessibility, flexibility, and community-based development. Regardless of these differences, all SQL-based systems lean on the same relational foundations, assuring that

SQL is still a unifying and enduring technology of modern database management (Date, 2003).

SQL's set-based operations provide an additional important benefit over procedural approaches. Rather than processing rows individually in loops, SQL operations operate on entire sets of data simultaneously, making use of the database engine's bulk-processing optimizations. This set-based thinking requires a conceptual shift for developers accustomed to imperative programming but brings substantial benefits in code conciseness, clarity, and performance. A complex operation that might require dozens of lines of procedural code with nested loops and conditional statements often reduces to a single SQL statement with appropriate joins and aggregate functions. This conciseness improves maintainability, as the intent becomes immediately clear from the query format, and reduces the exposure area for bugs.

9.3.3 SQL Command Syntax

SQL commands follow a structured syntax that governs how statements are written and executed. SQL itself is generally case-insensitive with respect to keywords, meaning SELECT, select, and SeLeCt are functionally equivalent. However, convention dictates that reserved words be written in uppercase to improve readability and distinguish them from user-defined identifiers such as table and column names. Reserved words such as SELECT, FROM, WHERE, INSERT, UPDATE, and DELETE have special meanings within SQL and cannot be used as names for database objects without special delimiters. Each SQL statement must be terminated with a semicolon (;) to mark its completion, allowing the database system to distinguish between multiple commands submitted sequentially. This semicolon is especially important when executing batch scripts or stored procedures that contain multiple statements.

Although SQL is a free-format language, an SQL statement or set of statements is more readable if indentation and lineation are used. For example:

- Each clause in a statement should begin on a new line.
- The beginning of each clause needs to align with the beginning of the other clauses.
- If a clause has several parts, they should each appear on a separate line and be indented under the start of the clause to show the relationship.

Throughout this chapter, we use the following extended form of the Backus-Naur Form (BNF) notation to define SQL statements:

- Upper-case letters are used to represent reserved words and must be spelled exactly as shown.
- Lower-case letters are used to represent user-defined words;
- a vertical bar (|) indicates a choice among alternatives; for example, a | b |c;
- Curly braces indicate a required element; for example, {a};
- Square brackets indicate an optional element; for example, [a];
- An ellipsis (…) is used to indicate optional repetition of an item zero or more times.

For example, **{a|b} (c...)** means either a or b followed by zero or more repetitions of c separated by commas.

In practice, the DDL statements are used to create the database structure (that is, the tables) and the access mechanisms (that is, what each user can legally access), and then the DML statements are used to populate and query the tables.

Following consistent syntax conventions optimizes code clarity, facilitates debugging, and ensures compatibility between different database platforms.

9.3.4 Database Variables and Data Types

In SQL, variables serve as named storage locations that temporarily hold values during query execution, stored procedure execution, or script processing. However, their scope is limited to the batch, procedure, or function in which they are declared, supporting encapsulation and reducing the risk of naming conflicts (Silberschatz et al., 2019). Due to SQL's declarative nature, variables play a secondary role compared to procedural programming languages and are primarily used within procedural extensions such as T-SQL (SQL Server), PL/SQL (Oracle), or MySQL's stored procedure language. Every variable must be declared with an explicit data type, which defines both the kind of data it can store and the associated memory requirements (Silberschatz et al., 2019).

Variables support more advanced database logic, including storing intermediate computation results, implementing conditional control flow, managing iteration within procedures, and passing parameters between stored procedures and functions. They are also fundamental to dynamic SQL, where query strings are constructed and executed at runtime. Although variables are essential for procedural database programming, many data retrieval tasks do not require them, as SQL's set-oriented operations allow direct manipulation of entire relations without intermediate storage.

Attribute data types in SQL define the nature of data that columns can store, establishing both storage requirements and valid operations on that data. Choosing appropriate data types is a critical design decision, as it affects indexing behavior, implicit type conversions, and long-term system maintainability. Modern relational database systems support a wide range of data types, including numeric, character, temporal, and specialized types for semi-structured, spatial, and complex data. Thoughtful data type selection can help ensure efficient, portable database designs that can advance alongside application requirements (Date, 2003; Elmasri & Navathe, 2016).

The hierarchical organization of database data types shows a structured taxonomy that helps developers understand relationships among types and select appropriate representations for different data requirements. At the highest level, SQL data types are commonly divided into exact numeric types, such as integers and fixed-point decimals, which represent values precisely, and approximate numeric types, such as floating-point numbers, which trade exact precision for a wider range and computational efficiency. This distinction is especially important in domains like financial systems, where exact representation is mandatory, versus scientific and analytical applications, where approximation is often acceptable (Elmasri & Navathe, 2016; Date, 2003). Numeric types further differ based on whether fractional values are supported and on the range of

representable values, while character data types are typically classified into fixed-length and variable-length forms, each having effects on storage optimization and performance (Silberschatz et al., 2019). Temporal data types distinguish between date-only, time-only, and combined date–time values, with additional variations supporting time zone awareness. Binary data types parallel character types in structure, but store raw byte arrays rather than encoded textual data. This structured classification of data types provides a conceptual framework that supports effective database design and productive data management (Elmasri & Navathe, 2016).

The common categories of database data types include:

- Numeric Data Types Used for storing numbers.
    - Integer Whole numbers (e.g., INT, BIGINT).
    - Decimal/Fixed-Point: Precise decimal numbers, suitable for financial data (e.g., DECIMAL, NUMERIC).
    - Floating-Point Approximate decimal numbers, often used for scientific calculations (e.g., FLOAT, DOUBLE).
- Text/String Data Types Used for storing alphanumeric characters.
    - CHAR Fixed-length strings (e.g., a two-letter state code).
    - VARCHAR Variable-length strings, for data like names or addresses.
    - TEXT/BLOB Large objects for storing large amounts of text (e.g., a product description) or binary data (e.g., an image file).
- Date and Time Data Types Used for temporal information.
    - DATE Stores a calendar date (year, month, day).
    - TIME Stores a time of day.
    - DATETIME/TIMESTAMP Stores a combination of date and time, often used for logging events.
- Boolean Data Type Stores logical values, typically TRUE or FALSE, often used for flags or binary conditions.

Note that the exact syntax and implementation of these data types can vary across database management systems (DBMSs), including MySQL, PostgreSQL, Oracle, and SQL Server.

9.4 Creating a Database

A database is an organized collection of structured data stored electronically, designed to support efficient retrieval, management, and updating of information. Creating a database involves establishing the basic structure where data will reside, including defining tables, relationships, data types, and constraints that ensure data integrity and consistency. DBMS provides the tools and interfaces necessary to create, maintain, and control access to databases through structured commands and administrative functions (Coronel & Morris, 2017). The creation process entails careful planning to determine appropriate schema

designs, naming conventions, storage parameters, and security procedures. Once established, databases serve as centralized repositories that multiple users and applications can access simultaneously while maintaining data accuracy and blocking unauthorized access. Understanding the fundamentals of database creation is essential for building robust information systems that support organizational operations and decision-making.

9.4.1 DDL Overview and Schema Concepts

Data Definition Language (DDL) comprises SQL commands that define, modify, and drop database structures, serving as the basis for all data operations. DDL operations are typically privileged actions requiring appropriate administrative rights, as they essentially change the database schema and can have domino effects on dependent objects, applications, and stored data. The primary DDL commands include CREATE for establishing new database objects, ALTER for modifying existing structures, DROP for removing objects, and TRUNCATE for removing all data while preserving structure. SQL uses a hierarchical organizational model in which a catalog is the highest-level container, typically corresponding to a database server instance, and contains multiple schemas.

> **Database Schema**
>
> A database schema is the logical structure that defines how data is organized within a database, including tables, columns, data types, relationships, constraints, and the rules governing data storage and access.

A schema is a database object that acts as a container. Schemas logically group related objects into a single container that keeps them separate from other database objects in other containers (schemas). Schemas are database oriented. This allows for the same schema in a different database, and they will work independently. When creating a new object in SQL, such as tables, views, stored procedures, etc., if you do not specify a schema for that object, it will be placed in the default logical schema for that SQL instance. In most cases, that default is the “dbo” (Database Owner) schema. Note that this applies specifically to SQL Server; in MySQL, databases themselves serve as schema containers with no equivalent default schema name, while in PostgreSQL the default schema is 'public'.

Before creating tables, we must understand constraints, the rules enforced by a database to maintain data integrity. Constraints include PRIMARY KEY for unique row identification, FOREIGN KEY for referential integrity, UNIQUE for ensuring distinct values, CHECK for ensuring domain constraints enforcing valid value ranges, and NOT NULL for mandatory fields. Modern databases also support views (virtual tables derived from queries), assertions (cross-table constraints, though rarely implemented), and triggers (stored procedures automatically executed in response to data changes). However, the use of triggers requires careful consideration, as it introduces complexity and can affect performance.

9.4.2 Mapping the Relational Model to SQL DDL Commands

The process of mapping a relational model to SQL Data Definition Language (DDL) commands involves systematically translating the conceptual and logical structures of the relational model, derived from Entity-Relationship (ER) diagrams, into executable SQL syntax, as shown in Table 9.1.

Table 9.1: Mapping Table: Relational Model Elements to SQL DDL Commands

Relational Model Element	Description in the Relational Model	SQL DDL Command
Relation (Entity)	A set of tuples representing an entity type	CREATE TABLE [table_name].
Attribute	A property of an entity with a domain	Column definition: [column_name] [data_type] [constraints]
Simple Attribute	Single-valued atomic attribute	Direct column: [column_name] [data_type]
Composite Attribute	An attribute composed of multiple parts.	Multiple columns: Street VARCHAR(100), City VARCHAR(50).
Multivalued Attribute	Attribute with multiple values	Separate table with foreign key (e.g., CREATE TABLE EMPLOYEE_PHONE (Employee_ID VARCHAR(10), Phone_number VARCHAR(20))
Derived Attribute	Computed from other attributes	Not stored; use a VIEW or a computed column. (e.g., Age from Birth_date).
Primary Key	Unique identifier for tuples.	PRIMARY KEY ([column(s)]) (e.g., Faculty_ID).
Composite Primary Key	Multiple attributes as key	PRIMARY KEY ([col1], [col2]) (e.g., Product_ID, Warehouse_ID in STORED_IN).
Foreign Key	Attribute referencing another relation's primary key.	FOREIGN KEY ([column]) REFERENCES table
Domain Constraint	Restriction on attribute values.	CHECK ([condition]) or data type limits (e.g., CHECK (Budget > 0)).
Nullability	Whether the attribute can be null.	[NOT] NULL (e.g., Dept_name VARCHAR(50) NOT NULL).
Default Value	Preset value for attribute.	DEFAULT [value] (e.g., Join_date DATE DEFAULT GETDATE() in SQL Server; use CURDATE() in MySQL).
Unique Constraint	Ensures uniqueness (non-key).	UNIQUE ([column]) (e.g., Email VARCHAR(100) UNIQUE).
Index	Optimizes access.	CREATE INDEX [index_name] ON table
View	Virtual relation derived from base relations.	CREATE VIEW [view_name] AS [query]
Assertion	Cross-table constraint.	CREATE ASSERTION [name] CHECK
Trigger	Procedural code on events.	CREATE TRIGGER [name] ON [table] FOR [event] AS [code]

Definition: Data Definition Language (DDL)

DDL is a subset of SQL commands used to define, modify, and delete database structures such as tables, indexes, and schemas, including CREATE, ALTER, and DROP statements.

This ensures the database schema accurately represents entities, attributes, relationships, and constraints while enforcing data integrity and normalization.

Key steps include:

1. **Identifying Entities and Relations**: Each entity in the relational model (e.g., DEPARTMENT or FACULTY from the provided case studies) is mapped to a table using the CREATE TABLE statement. The table name reflects the entity name.
2. **Defining Attributes as Columns**: Attributes become columns, with appropriate data types assigned based on their domain (e.g., VARCHAR for strings, DECIMAL for monetary values, DATE for temporal data). Defaults, nullability (NOT NULL), and domain constraints (e.g., CHECK) are specified to match attribute properties.
3. **Specifying Keys**: Primary keys, which uniquely identify tuples, are declared using the PRIMARY KEY clause. Composite keys list multiple columns. Foreign keys, representing relationships, use FOREIGN KEY ... REFERENCES to link tables, enforcing referential integrity (e.g., cascading updates/deletes if needed).
4. **Handling Relationships**: One-to-many relationships map to foreign keys on the "many" side. Many-to-many requires intermediate tables with composite keys. Weak entities inherit keys from strong entities.
5. **Adding Constraints and Indexes**: Extra constraints, such as UNIQUE or CHECK, enforce business rules. Indexes (CREATE INDEX) improve query performance for commonly accessed columns.
6. **Validation and Implementation**: The DDL is executed in a DBMS such as SQL Server or MySQL, with adjustments for dialect differences (e.g., backticks in MySQL for reserved words).

Constraints in SQL represent declarative rules that enforce data integrity, ensuring database consistency by preventing invalid data entry and maintaining relationships between tables. These constraints operate automatically: the database management system rejects any operation that violates them, thereby embedding business rules directly into the database schema rather than relying solely on application-level validation. SQL provides several constraint categories, each serving specific integrity purposes.

9.4.3 Creating Database and Tables

Creating a database begins with the CREATE DATABASE command, which establishes a new catalog to house all related schemas and objects.

CREATE DATABASE *databasename*;

Once created, the USE command makes it the active context.

USE *databasename*;

The CREATE TABLE statement then defines the storage structure, specifying column names, data types, nullability, and constraints that enforce business rules and maintain data integrity. The syntax follows the pattern CREATE TABLE table_name (column1 datatype constraints, column2 datatype constraints, ..., table_constraints), where column-level constraints apply to individual columns and table-level constraints can span multiple columns, particularly for composite primary keys or cross-column checks.

```
CREATE TABLE table_name (
  column1 datatype,
  column2 datatype,
  column3 datatype,
   ....);
```

Indexes, while not strictly part of the table definition, are important for query performance and are created with CREATE INDEX index_name ON table_name (column_list), with options for unique, clustered, and non-clustered indexes (SQL Server) and full-text indexes for text search.

```
CREATE INDEX index_name
  ON table_name (column1, column2, ...);
```

The ALTER TABLE command enables modifying existing structures without losing data, which is essential for evolving schemas. ADD COLUMN introduces new attributes to tables, allowing schema expansion. DROP COLUMN removes obsolete columns, but it permanently deletes the column's data across all rows. MODIFY COLUMN or ALTER COLUMN changes data types or constraints on existing columns, useful when initial design choices turn out insufficient. However, type conversions may fail if existing data cannot be converted to the new type, requiring careful planning and potentially data migration.

```
ALTER TABLE table_name

  ADD column_name datatype;

ALTER TABLE table_name

  DROP COLUMN column_name;

ALTER TABLE table_name

  ALTER/MODIFY COLUMN column_name datatype;
```

Constraint management through ADD CONSTRAINT and DROP CONSTRAINT enables flexible modification of integrity rules. Adding constraints retroactively validates existing data; if violations exist, the operation fails until data is corrected. For example, adding a CHECK constraint ensuring positive salaries fails if negative values already exist. Dropping constraints removes enforcement, occasionally necessary during bulk operations or when business rules change, though this temporarily reduces data integrity protection.

```
ALTER TABLE table_name
  ADD CONSTRAINT constraint_name constraint_type (column_name);


ALTER TABLE table_name
  DROP CONSTRAINT constraint_name;
```

The distinction between DROP TABLE and TRUNCATE TABLE is critical for database administrators. DROP TABLE permanently removes the entire table structure and all contained data, deleting metadata from the database catalog and releasing storage space. This irreversible operation completely removes the table, requiring recreation to restore functionality. In contrast, TRUNCATE TABLE removes all rows while preserving the table

structure, indexes, and constraints, functioning as a fast alternative to DELETE for complete data removal. TRUNCATE executes faster than DELETE because it deallocates entire data pages rather than logging individual row deletions, and it resets auto-increment counters to initial values. However, TRUNCATE cannot be rolled back in many systems and cannot include WHERE clauses for selective deletion, keeping it suitable only for complete table clearing during data refresh or testing.

```
DROP TABLE table_name;

TRUNCATE TABLE table_name;
```

Knowing these operations helps database professionals choose appropriate methods for schema evolution and data management tasks, managing performance, safety, and functionality requirements in production environments.

Named Constraints

The CREATE TABLE statements throughout this chapter use the CONSTRAINT keyword to assign explicit names to every constraint. Rather than declaring constraints inline, such as Faculty_ID VARCHAR(255) PRIMARY KEY, each constraint is written as CONSTRAINT constraint_name constraint_type (column). Naming constraints make violation error messages immediately identifiable, allow specific constraints to be dropped or modified by name using ALTER TABLE, and document the purpose of each rule directly in the schema. This is professional practice in database development and is required in all implementations in this course.

University Database System

The University database system demonstrates a comprehensive academic data management structure that models the detailed relationships among faculties, students, professors, courses, and supporting entities.

Based on the Relational database schema, in section 7.4.2 and the Complete Relational Model Schema of the University System, presented in subsection 7.6.4, we identify the database schema.

The FACULTY table serves as a core entity and central reference point for both students and professors, defining the academic divisions of the university.

```
CREATE TABLE FACULTY (
    Faculty_ID  VARCHAR(255)    NOT NULL,
    Name        VARCHAR(255)    NOT NULL,
    Budget      DECIMAL(10, 2),
    Location    VARCHAR(255),
    CONSTRAINT PK_FACULTY PRIMARY KEY (Faculty_ID),
    CONSTRAINT CHK_FACULTY_Budget CHECK (Budget >= 0)
);
```

The FACULTY table stores information about academic divisions within the university. Faculty_ID is declared NOT NULL and designated as the primary key through CONSTRAINT PK_FACULTY, guaranteeing that every faculty record is uniquely identifiable. Name is also NOT NULL since every faculty must have a name. Budget and Location are optional at the

database level, allowing records to be created before all administrative details are finalized. CONSTRAINT CHK_FACULTY_Budget enforces a domain rule that budget values cannot be negative, preventing invalid financial data from entering the system. Budget uses DECIMAL(10,2) to store monetary values precisely with two decimal places, while text attributes use VARCHAR(255) for flexible length storage.

The STUDENT table captures learner information, including Student_ID (primary key), First_name, Last_name, Birth_date, Age, and address components (Street, City, Country, Zip_code), with a foreign key Faculty_ID linking students to their enrolled faculty.

```
CREATE TABLE STUDENT (
    Student_ID  VARCHAR(255)   NOT NULL,
    First_name  VARCHAR(255)   NOT NULL,
    Last_name   VARCHAR(255)   NOT NULL,
    Birth_date  DATE   NOT NULL,
    Age         INT,
    Street      VARCHAR(255),
    City        VARCHAR(255),
    Country     VARCHAR(255),
    Zip_code    VARCHAR(10),
    Faculty_ID  VARCHAR(255)   NOT NULL,
    CONSTRAINT PK_STUDENT PRIMARY KEY (Student_ID),
    CONSTRAINT FK_Student_Faculty FOREIGN KEY (Faculty_ID) REFERENCES
FACULTY(Faculty_ID),
    CONSTRAINT CHK_Student_Age CHECK (Age >= 16 AND Age <= 100)
);
```

CONSTRAINT PK_STUDENT designates Student_ID as the unique identifier for each student record. First_name, Last_name, and Birth_date are declared NOT NULL as they are mandatory identification attributes — no student record should exist without them. Faculty_ID is also NOT NULL because every student must belong to exactly one faculty, reflecting the total participation constraint established in the ER model. CONSTRAINT FK_Student_Faculty enforces referential integrity, ensuring that no student can be assigned to a faculty that does not exist in the FACULTY table. Address components (Street, City, Country, Zip_code) are optional, allowing partial records to be created. Age is stored as an optional integer derived from Birth_date for administrative convenience; CONSTRAINT CHK_Student_Age restricts it to a realistic range of 16 to 100, preventing obviously invalid values from entering the system.

An EMAIL table demonstrates a multi-valued attribute implementation, using a composite primary key (Student_ID, Email) to associate multiple email addresses with a single student while maintaining referential integrity through a foreign key to STUDENT.

```
CREATE TABLE EMAIL (
    Student_ID  VARCHAR(255)   NOT NULL,
    Email       VARCHAR(255)   NOT NULL,
    CONSTRAINT PK_EMAIL PRIMARY KEY (Student_ID, Email),
```

CONSTRAINT FK_Email_Student FOREIGN KEY (Student_ID) REFERENCES STUDENT(Student_ID));

The EMAIL table handles the multivalued Email attribute from the ER model by storing each email address as a separate row. Both Student_ID and Email are declared NOT NULL since neither component of a composite key can be null. CONSTRAINT PK_EMAIL defines a composite primary key on (Student_ID, Email), ensuring that the same email address cannot be recorded twice for the same student while still allowing a student to have multiple distinct addresses — one row per address. CONSTRAINT FK_Email_Student enforces referential integrity, guaranteeing that email entries can only exist for students already present in the STUDENT table. Deleting a student without first removing their email records would violate this constraint, so cascade behaviour should be considered when managing student deletions in the application.

The PROFESSOR table models faculty members with Professor_ID (primary key), First_name, Last_name, Office, Email, Hire_date, Academic_title (e.g., Assistant Professor, Associate Professor, Full Professor), Faculty_ID indicating departmental affiliation, and Heads_Faculty_ID to represent deans or department heads, indicating which faculty the professor heads, through a foreign key referencing the FACULTY table.

```
CREATE TABLE PROFESSOR (
    Professor_ID     VARCHAR(255)    NOT NULL,
    First_name       VARCHAR(255)    NOT NULL,
    Last_name        VARCHAR(255)    NOT NULL,
    Office           VARCHAR(50),
    Email            VARCHAR(255)    NOT NULL,
    Hire_date        DATE    NOT NULL,
    Academic_title   VARCHAR(255)    NOT NULL,
    Faculty_ID       VARCHAR(255)    NOT NULL,
    Heads_Faculty_ID VARCHAR(255),
    CONSTRAINT PK_PROFESSOR PRIMARY KEY (Professor_ID),
    CONSTRAINT FK_Professor_Faculty FOREIGN KEY (Faculty_ID) REFERENCES FACULTY(Faculty_ID),
    CONSTRAINT FK_Professor_HeadsFaculty FOREIGN KEY (Heads_Faculty_ID) REFERENCES FACULTY(Faculty_ID),
    CONSTRAINT UQ_Professor_Email UNIQUE (Email),
    CONSTRAINT CHK_Professor_AcademicTitle CHECK (Academic_title IN ('Assistant Professor', 'Associate Professor', 'Full Professor'))
);
```

CONSTRAINT PK_PROFESSOR uniquely identifies each professor record. First_name, Last_name, Email, Hire_date, and Academic_title are declared NOT NULL as they are mandatory professional attributes. Faculty_ID is NOT NULL because every professor must be affiliated with exactly one faculty, reflecting total participation from the ER model; CONSTRAINT FK_Professor_Faculty enforces this referential link. Heads_Faculty_ID allows NULL because not all professors head a faculty — only deans hold this role — reflecting partial participation; CONSTRAINT FK_Professor_HeadsFaculty ensures that when a value is present, it references a valid faculty. Both foreign keys point to FACULTY(Faculty_ID) since

Heads_Faculty_ID identifies which faculty the professor leads, not another professor. CONSTRAINT UQ_Professor_Email ensures no two professors share the same email address, enforcing it as an alternate key. CONSTRAINT CHK_Professor_AcademicTitle restricts the title to the three valid academic ranks used in the university system.

The COURSE table defines academic offerings with Course_ID (primary key), Title, Credits, Description, Faculty_ID indicating the offering faculty, and Professor_ID for the instructor, establishing many-to-one relationships with both FACULTY and PROFESSOR.

```
CREATE TABLE COURSE (
    Course_ID    VARCHAR(255)    NOT NULL,
    Title        VARCHAR(255)    NOT NULL,
    Credits      INT             NOT NULL,
    Description  VARCHAR(MAX),
    Faculty_ID   VARCHAR(255)    NOT NULL,
    Professor_ID VARCHAR(255)    NOT NULL,
    CONSTRAINT PK_COURSE PRIMARY KEY (Course_ID),
    CONSTRAINT FK_Course_Faculty FOREIGN KEY (Faculty_ID) REFERENCES
FACULTY(Faculty_ID),
    CONSTRAINT FK_Course_Professor FOREIGN KEY (Professor_ID) REFERENCES
PROFESSOR(Professor_ID),
    CONSTRAINT CHK_Course_Credits CHECK (Credits >= 2 AND Credits <= 10)
);
```

Note: VARCHAR(MAX) is SQL Server syntax. In MySQL, replace it with TEXT for the Description column.

CONSTRAINT PK_COURSE uniquely identifies each course record. Title and Credits are NOT NULL as they are essential academic attributes — a course cannot exist without a name and credit value. Faculty_ID and Professor_ID are both NOT NULL because every course must be offered by exactly one faculty and taught by exactly one professor, reflecting the total participation constraints established in the ER model; CONSTRAINT FK_Course_Faculty and CONSTRAINT FK_Course_Professor enforce these referential links, preventing courses from referencing non-existent faculties or professors. Description is optional, allowing course records to be created before full descriptions are finalized. CONSTRAINT CHK_Course_Credits restricts credit values to the valid range of 2 to 10, consistent with the ECTS-based credit system used in the university.

The PREREQUISITE table implements a many-to-many self-relationship within COURSE, using a composite primary key (Course_ID, Prerequisite_Course_ID) where both columns reference COURSE.Course_ID, enabling complex prerequisite chains where advanced courses require completion of foundational courses.

```
CREATE TABLE PREREQUISITE (
    Course_ID              VARCHAR(255)    NOT NULL,
    Prerequisite_Course_ID  VARCHAR(255)    NOT NULL,
    CONSTRAINT PK_PREREQUISITE PRIMARY KEY (Course_ID,
Prerequisite_Course_ID),
```

```
    CONSTRAINT FK_Prerequisite_Course FOREIGN KEY (Course_ID) REFERENCES COURSE(Course_ID),
    CONSTRAINT FK_Prerequisite_RequiredCourse FOREIGN KEY (Prerequisite_Course_ID) REFERENCES COURSE(Course_ID),
    CONSTRAINT CHK_Prerequisite_NoCycle CHECK (Course_ID <> Prerequisite_Course_ID)
);
```

Both Course_ID and Prerequisite_Course_ID are declared NOT NULL since neither component of a composite key can be null. CONSTRAINT PK_PREREQUISITE defines a composite primary key on both columns, ensuring each course-prerequisite pair is recorded only once and preventing duplicate prerequisite entries. Both CONSTRAINT FK_Prerequisite_Course and CONSTRAINT FK_Prerequisite_RequiredCourse reference COURSE(Course_ID), implementing the recursive many-to-many self-relationship — a course can require multiple prerequisites, and a single course can be a prerequisite for multiple other courses, with both sides always pointing to valid entries in the COURSE table. CONSTRAINT CHK_Prerequisite_NoCycle adds an important domain rule preventing a course from being declared as its own prerequisite, which would be semantically meaningless and could cause infinite loops in prerequisite resolution logic.

The ENROLLS_IN table models student course registration through a many-to-many relationship between STUDENT and COURSE, with a composite primary key (Student_ID, Course_ID) and additional attributes Enrollment_date and Grade tracking academic progress.

```
CREATE TABLE ENROLLS_IN (
    Student_ID      VARCHAR(255)   NOT NULL,
    Course_ID       VARCHAR(255)   NOT NULL,
    Enrollment_date  DATE NOT NULL DEFAULT GETDATE(),
    Grade   INT,
    CONSTRAINT PK_ENROLLS_IN PRIMARY KEY (Student_ID, Course_ID),
    CONSTRAINT FK_EnrollsIn_Student FOREIGN KEY (Student_ID) REFERENCES STUDENT(Student_ID),
    CONSTRAINT FK_EnrollsIn_Course FOREIGN KEY (Course_ID) REFERENCES COURSE(Course_ID),
    CONSTRAINT CHK_EnrollsIn_Grade CHECK (Grade >= 5 AND Grade <= 10)
);
```

Note: DEFAULT GETDATE() is SQL Server syntax. In MySQL, replace it with DEFAULT CURDATE().

CONSTRAINT PK_ENROLLS_IN defines a composite primary key on (Student_ID, Course_ID), ensuring a student can be enrolled in the same course only once. CONSTRAINT FK_EnrollsIn_Student and CONSTRAINT FK_EnrollsIn_Course enforce referential integrity, guaranteeing that enrollment records can only reference existing students and courses. Both Student_ID and Course_ID are NOT NULL, as they form the composite key. Enrollment_date is NOT NULL with a DEFAULT GETDATE() clause, automatically recording the current date when a new enrollment is inserted without an

explicit date value. Grade allows NULL because a student is first enrolled before receiving a grade — the value is populated later when academic results are recorded. CONSTRAINT CHK_EnrollsIn_Grade restricts grade values to the range 5–10, consistent with the European grading scale used in the university system, where 5 represents a failing grade, and 10 is the maximum.

The CLASS_SESSION table represents individual class meetings with a composite primary key (Course_ID, Session_number, Year), including attributes for Room, Semester, Time, Day, and Duration, supporting detailed schedule management.

```
CREATE TABLE CLASS_SESSION (
    Course_ID   VARCHAR(255)   NOT NULL,
    Session_number  INT   NOT NULL,
    Year  INT  NOT NULL,
    Room    VARCHAR(50),
    Semester   VARCHAR(50),
    Time  TIME,
    Day    VARCHAR(20),
    Duration   INT,
    CONSTRAINT  PK_CLASS_SESSION  PRIMARY  KEY  (Course_ID,  Session_number,
Year),
    CONSTRAINT  FK_ClassSession_Course  FOREIGN  KEY  (Course_ID)  REFERENCES
COURSE(Course_ID),
    CONSTRAINT CHK_ClassSession_SessionNumber CHECK (Session_number >= 1),
    CONSTRAINT CHK_ClassSession_Year CHECK (Year >= 2000),
    CONSTRAINT CHK_ClassSession_Duration CHECK (Duration > 0)
);
```

Note: Time and Year may conflict with reserved words or built-in functions in some database systems. In MySQL, enclose them in backticks: `Time`, `Year`.

CONSTRAINT PK_CLASS_SESSION defines a three-column composite primary key on (Course_ID, Session_number, Year). Including Year is essential — the same session number recurs across academic years, so (Course_ID, Session_number) alone would not be unique across time. CONSTRAINT FK_ClassSession_Course ensures every class session belongs to a valid course. All three primary key components are NOT NULL as they collectively identify each session. Scheduling attributes Room, Semester, Time, Day, and Duration are optional at the database level, allowing session records to be created before scheduling details are confirmed. CONSTRAINT CHK_ClassSession_SessionNumber enforces that session numbers begin at 1, preventing zero or negative values. CONSTRAINT CHK_ClassSession_Year sets a lower bound of the year 2000, preventing obviously invalid year entries. CONSTRAINT CHK_ClassSession_Duration ensures session duration is always a positive value, as a class meeting of zero or negative minutes is meaningless.

The ATTENDS table links students to specific class sessions, using a composite primary key (Student_ID, Course_ID, Session_number, Year) with an Attendance attribute (Present, Absent, Excused) for tracking participation.

```
CREATE TABLE ATTENDS (
    Student_ID      VARCHAR(255)    NOT NULL,
    Course_ID       VARCHAR(255)    NOT NULL,
    Session_number  INT  NOT NULL,
    Year    INT   NOT NULL,
    Attendance      VARCHAR(20),
    CONSTRAINT      PK_ATTENDS      PRIMARY     KEY     (Student_ID,    Course_ID,
Session_number, Year),
    CONSTRAINT FK_Attends_Student FOREIGN KEY (Student_ID) REFERENCES
STUDENT(Student_ID),
    CONSTRAINT      FK_Attends_ClassSession     FOREIGN     KEY     (Course_ID,
Session_number,  Year)  REFERENCES  CLASS_SESSION(Course_ID,  Session_number,
Year),
    CONSTRAINT CHK_Attends_Attendance CHECK (Attendance IN ('Present', 'Absent',
'Excused'))
);
```

CONSTRAINT PK_ATTENDS defines a four-column composite primary key on (Student_ID, Course_ID, Session_number, Year), uniquely identifying each student's attendance record at a specific class session in a specific academic year. All four components are NOT NULL as they collectively form the key. CONSTRAINT FK_Attends_Student links each attendance record to a valid student. CONSTRAINT FK_Attends_ClassSession is a composite foreign key referencing all three columns of CLASS_SESSION's primary key — this is necessary because a class session is itself identified by a three-column key, and the entire combination must be referenced together to maintain referential integrity. Attendance allows NULL to support pre-recording of session records before attendance is confirmed, for example, when sessions are scheduled in advance. CONSTRAINT CHK_Attends_Attendance restricts the attribute to the three valid status values — 'Present', 'Absent', and 'Excused' — preventing arbitrary strings from entering the system and ensuring consistent reporting across all attendance records.

Next, the TEXTBOOK table catalogs course materials with ISBN (primary key), Title, Author, Edition, Publisher, and Available_copies.

```
CREATE TABLE TEXTBOOK (
    ISBN    VARCHAR(20)    NOT NULL,
    Title     VARCHAR(255)    NOT NULL,
    Author    VARCHAR(255)    NOT NULL,
    Edition    VARCHAR(50),
    Publisher   VARCHAR(255),
    Available_copies INT  DEFAULT 0,
    CONSTRAINT PK_TEXTBOOK PRIMARY KEY (ISBN),
    CONSTRAINT CHK_Textbook_Copies CHECK (Available_copies >= 0)
);
```

CONSTRAINT PK_TEXTBOOK designates ISBN as the primary key, which is appropriate since the International Standard Book Number is a globally unique identifier assigned to every published book, making it a natural and stable key. Title and Author are declared NOT

NULL as they are essential identifying attributes — a textbook record without a title or author would be meaningless. Edition uses VARCHAR(50) to accommodate ordinal values such as '1st', '3rd', or '7th ed.', and is optional since some works do not carry edition markings. Publisher is also optional at the database level, allowing records to be created before all cataloguing details are available. Available_copies carries a DEFAULT 0, automatically setting new textbook entries to zero copies until stock is recorded, and CONSTRAINT CHK_Textbook_Copies ensures the value never falls below zero — a library cannot hold a negative number of physical copies.

Finally, the ASSIGNMENT table creates a many-to-many relationship between professors, courses, and textbooks through a composite primary key (Professor_ID, Course_ID, ISBN), it tracks which textbooks are assigned to which courses by which instructors.

```
CREATE TABLE ASSIGNMENT (
    Professor_ID    VARCHAR(255)    NOT NULL,
    Course_ID       VARCHAR(255)    NOT NULL,
    ISBN            VARCHAR(20)     NOT NULL,
    CONSTRAINT PK_ASSIGNMENT PRIMARY KEY (Professor_ID, Course_ID, ISBN),
    CONSTRAINT FK_Assignment_Professor FOREIGN KEY (Professor_ID)
REFERENCES PROFESSOR(Professor_ID),
    CONSTRAINT FK_Assignment_Course FOREIGN KEY (Course_ID) REFERENCES
COURSE(Course_ID),
    CONSTRAINT FK_Assignment_Textbook FOREIGN KEY (ISBN) REFERENCES
TEXTBOOK(ISBN)
);
```

CONSTRAINT PK_ASSIGNMENT defines a three-column composite primary key on (Professor_ID, Course_ID, ISBN), implementing the ternary relationship between PROFESSOR, COURSE, and TEXTBOOK established in the ER model. All three columns are NOT NULL since each component is essential — an assignment must always identify a professor, a course, and a textbook simultaneously. The composite key ensures that the same textbook cannot be assigned to the same course by the same professor more than once, while still allowing the same textbook to be assigned to different courses or the same course to use multiple textbooks. Every column serves a dual role: each is simultaneously part of the primary key and a foreign key referencing its parent table. CONSTRAINT FK_Assignment_Professor, CONSTRAINT FK_Assignment_Course, and CONSTRAINT FK_Assignment_Textbook collectively enforce referential integrity across all three dimensions, ensuring that assignments can only reference professors, courses, and textbooks that already exist in the database.

🏢 *Company/Business Database*

The Company database system models a business organization's operational structure, encompassing employees, departments, products, customers, orders, and logistics.

Based on the Relational database schema, in section 7.4.2 and the Complete Relational Model Schema of the Company System, presented in subsection 7.6.8, identify the database schema:

The DEPARTMENT table serves as an organizational unit, with Department_ID (primary key), Dept_name, Budget, and Location, representing corporate sectors, such as Sales, Marketing, or Human Resources.

```
CREATE TABLE DEPARTMENT (
    Department_ID   VARCHAR(255)   NOT NULL,
    Dept_name       VARCHAR(255)   NOT NULL,
    Budget          DECIMAL(10, 2),
    Location        VARCHAR(255),
    CONSTRAINT PK_DEPARTMENT PRIMARY KEY (Department_ID),
    CONSTRAINT CHK_Department_Budget CHECK (Budget >= 0)
);
```

CONSTRAINT PK_DEPARTMENT uniquely identifies each department record. Dept_name is declared NOT NULL since every department must have a name — a nameless department record would be meaningless for organisational management. Budget and Location are optional at the database level, allowing department records to be created before financial allocations or physical locations are finalised. CONSTRAINT CHK_Department_Budget enforces the domain rule that a department's budget cannot be negative, preventing invalid financial data from entering the system. Budget uses DECIMAL(10,2) to store monetary values precisely with two decimal places, consistent with the University FACULTY table, which models the same concept.

The WAREHOUSE table models storage facilities with Warehouse_ID (primary key), Location, and Capacity, representing physical inventory sites.

```
CREATE TABLE WAREHOUSE (
    Warehouse_ID    VARCHAR(255)    NOT NULL,
    Location        VARCHAR(50)     NOT NULL,
    Capacity        INT,
    Department_ID   VARCHAR(255)    NOT NULL,
    CONSTRAINT PK_WAREHOUSE PRIMARY KEY (Warehouse_ID),
    CONSTRAINT FK_Warehouse_Department FOREIGN KEY (Department_ID)
REFERENCES DEPARTMENT(Department_ID),
    CONSTRAINT CHK_Warehouse_Capacity CHECK (Capacity > 0)
);
```

CONSTRAINT PK_WAREHOUSE uniquely identifies each warehouse facility. Location is declared NOT NULL since a warehouse must always have a physical location — a storage facility without an address is operationally meaningless. Department_ID is NOT NULL because every warehouse is overseen by exactly one department, reflecting the total participation constraint established in the ER model; CONSTRAINT FK_Warehouse_Department enforces this referential link, preventing warehouse records from referencing non-existent departments. Capacity is optional at the database level, allowing warehouse records to be created before a formal capacity assessment is completed, but CONSTRAINT CHK_Warehouse_Capacity ensures that once a value is entered, it is always positive — a warehouse cannot have zero or negative storage capacity.

The EMPLOYEE table captures workforce data with Employee_ID (primary key), First_name, Last_name, Email, Hire_date, Salary, and crucially, Department_ID (foreign key to DEPARTMENT) and Manages_Dept_ID to indicate which employees are department managers, along with Warehouse_ID for employees assigned to warehouse operations. This creates a complicated network of relationships showing employment, management, and assignment structures.

```
CREATE TABLE EMPLOYEE (
    Employee_ID     VARCHAR(255)    NOT NULL,
    First_name      VARCHAR(255)    NOT NULL,
    Last_name       VARCHAR(255)    NOT NULL,
    Email           VARCHAR(255)    NOT NULL,
    Hire_date       DATE    NOT NULL,
    Salary          DECIMAL(10,2)   NOT NULL,
    Street          VARCHAR(255),
    City            VARCHAR(255),
    Zip_code        VARCHAR(10),
    Country         VARCHAR(255),
    Department_ID   VARCHAR(255)    NOT NULL,
    Manages_Dept_ID VARCHAR(255),
    Warehouse_ID    VARCHAR(255),
    CONSTRAINT PK_EMPLOYEE PRIMARY KEY (Employee_ID),
    CONSTRAINT FK_Employee_Department FOREIGN KEY (Department_ID)
REFERENCES DEPARTMENT(Department_ID),
    CONSTRAINT FK_Employee_ManagesDept FOREIGN KEY (Manages_Dept_ID)
REFERENCES DEPARTMENT(Department_ID),
    CONSTRAINT FK_Employee_Warehouse FOREIGN KEY (Warehouse_ID)
REFERENCES WAREHOUSE(Warehouse_ID),
    CONSTRAINT UQ_Employee_Email UNIQUE (Email),
    CONSTRAINT CHK_Employee_Salary CHECK (Salary > 0)
);
```

CONSTRAINT PK_EMPLOYEE uniquely identifies each employee record. First_name, Last_name, Email, Hire_date, and Salary are declared NOT NULL as they are mandatory workforce attributes — no employee record should be created without them. Department_ID is NOT NULL because every employee must belong to exactly one department, reflecting total participation from the ER model; CONSTRAINT FK_Employee_Department enforces this referential link. Manages_Dept_ID allows NULL because only department managers hold this role — most employees do not manage any department, reflecting partial participation; CONSTRAINT FK_Employee_ManagesDept ensures that when a value is present, it references a valid department. Both management and membership foreign keys reference DEPARTMENT(Department_ID) since they represent two distinct relationships with the same table. Warehouse_ID allows NULL as not all employees are assigned to warehouse operations; CONSTRAINT FK_Employee_Warehouse enforces referential integrity when a warehouse assignment exists. CONSTRAINT UQ_Employee_Email ensures no two employees share the same email address. CONSTRAINT CHK_Employee_Salary

prevents zero or negative salary values from entering the system. Address components are optional, allowing records to be created before all personal details are available.

The PRODUCT table defines sellable items with Product_ID (primary key), Product_name, Description, Category, Price, and Weight, forming the core of the company's inventory catalog.

```
CREATE TABLE PRODUCT (
    Product_ID      INT   NOT NULL,
    Product_name    VARCHAR(255)    NOT NULL,
    Description     VARCHAR(MAX),
    Category        VARCHAR(255)    NOT NULL,
    Price           DECIMAL(10,2)   NOT NULL,
    Weight          DECIMAL(8,2),
    CONSTRAINT PK_PRODUCT PRIMARY KEY (Product_ID),
    CONSTRAINT CHK_Product_Price CHECK (Price > 0),
    CONSTRAINT CHK_Product_Weight CHECK (Weight > 0)
);
```

Note: VARCHAR(MAX) is SQL Server syntax. In MySQL, replace it with TEXT for the Description column.

CONSTRAINT PK_PRODUCT uses an INT primary key, appropriate since product identifiers in this system are numeric values. Product_name and Category are declared NOT NULL — every product in the catalogue must have a name and belong to a category for inventory management and reporting to function correctly. Price is NOT NULL since no product can be listed without a price; CONSTRAINT CHK_Product_Price ensures the value is always positive, preventing free or negatively priced items from entering the catalogue. Description is optional, allowing products to be added to the catalogue before full descriptions are written. Weight is optional at the database level to accommodate products where weight may not yet be assessed, but CONSTRAINT CHK_Product_Weight ensures that once a value is provided, it is always positive, as a product cannot have zero or negative physical weight. Weight uses DECIMAL(8,2) to store values in kilograms with two decimal places, supporting both lightweight accessories and heavier items in the product range.

The STORED_IN table implements a many-to-many relationship between PRODUCT and WAREHOUSE, using a composite primary key (Product_ID, Warehouse_ID) with Quantity_on_hand tracking current inventory levels and Reorder_level defining the minimum stock threshold that triggers replenishment at each location.

```
CREATE TABLE STORED_IN (
    Product_ID   INT   NOT NULL,
    Warehouse_ID   VARCHAR(255)   NOT NULL,
    Quantity_on_hand    INT   NOT NULL DEFAULT 0,
    Reorder_level   INT   NOT NULL,
    CONSTRAINT PK_STORED_IN PRIMARY KEY (Product_ID, Warehouse_ID),
    CONSTRAINT FK_StoredIn_Product FOREIGN KEY (Product_ID) REFERENCES
PRODUCT(Product_ID),
```

```
    CONSTRAINT FK_StoredIn_Warehouse FOREIGN KEY (Warehouse_ID)
REFERENCES WAREHOUSE(Warehouse_ID),
    CONSTRAINT CHK_StoredIn_Quantity CHECK (Quantity_on_hand >= 0),
    CONSTRAINT CHK_StoredIn_ReorderLevel CHECK (Reorder_level >= 0)
);
```

CONSTRAINT PK_STORED_IN defines a composite primary key on (Product_ID, Warehouse_ID), ensuring each product-warehouse combination is recorded only once while allowing the same product to be stored across multiple warehouses. CONSTRAINT FK_StoredIn_Product and CONSTRAINT FK_StoredIn_Warehouse enforce referential integrity, guaranteeing that inventory records can only reference products and warehouses that already exist. Quantity_on_hand is declared NOT NULL with a DEFAULT 0, automatically initialising stock at zero when a new product-warehouse record is first created before physical stock arrives. Reorder_level is NOT NULL because every stored product must have a defined replenishment threshold — without it, the inventory management system cannot trigger restocking alerts. CONSTRAINT CHK_StoredIn_Quantity and CONSTRAINT CHK_StoredIn_ReorderLevel both enforce non-negative values, as neither current stock nor a reorder threshold can be meaningfully negative.

The SUPPLIER table maintains vendor information with Supplier_ID (primary key), Supplier_name, First_name, Last_name, Email, Street, City, Country, and Zip_code.

```
CREATE TABLE SUPPLIER (
    Supplier_ID     VARCHAR(10)     NOT NULL,
    Supplier_name   VARCHAR(100)    NOT NULL,
    First_name      VARCHAR(50),
    Last_name       VARCHAR(50),
    Email           VARCHAR(100),
    Street          VARCHAR(100),
    City            VARCHAR(50),
    Country         VARCHAR(50),
    Zip_code        VARCHAR(10),
    CONSTRAINT PK_SUPPLIER PRIMARY KEY (Supplier_ID),
    CONSTRAINT UQ_Supplier_Email UNIQUE (Email)
);
```

CONSTRAINT PK_SUPPLIER uniquely identifies each vendor record. Supplier_name is declared NOT NULL as it is the primary business identifier — a supplier record without a company name would be unmanageable in procurement operations. Contact person attributes First_name and Last_name are optional, reflecting that some vendor relationships may be managed at the company level without a designated individual contact. Email is optional but CONSTRAINT UQ_Supplier_Email ensures that when a value is provided no two suppliers share the same email address, preventing contact confusion in procurement communications. Address components Street, City, Country, and Zip_code are optional at the database level, allowing supplier records to be created before all vendor details are confirmed. Phone numbers are not stored directly in this table — consistent with the relational model established in Chapter 7, they are handled as a multivalued attribute in the separate SUPPLIER_PHONE table, which is presented later in this section.

The SUPPLIES table connects suppliers to products they provide, using a composite primary key (Supplier_ID, Product_ID) with Supply_date and Unit_cost attributes tracking procurement details.

```
CREATE TABLE SUPPLIES (
    Product_ID      INT     NOT NULL,
    Supplier_ID     VARCHAR(10)     NOT NULL,
    Supply_date     DATE    NOT NULL,
    Unit_cost       DECIMAL(10,2)   NOT NULL,
    CONSTRAINT PK_SUPPLIES PRIMARY KEY (Product_ID, Supplier_ID),
    CONSTRAINT FK_Supplies_Product FOREIGN KEY (Product_ID) REFERENCES
PRODUCT(Product_ID),
    CONSTRAINT FK_Supplies_Supplier FOREIGN KEY (Supplier_ID) REFERENCES
SUPPLIER(Supplier_ID),
    CONSTRAINT CHK_Supplies_UnitCost CHECK (Unit_cost > 0)
);
```

CONSTRAINT PK_SUPPLIES defines a composite primary key on (Product_ID, Supplier_ID), establishing that each supplier can supply each product at most once in this relation — one procurement record per product-supplier combination. CONSTRAINT FK_Supplies_Product and CONSTRAINT FK_Supplies_Supplier enforce referential integrity, ensuring that supply records can only reference products and suppliers that already exist in the database. Both Product_ID and Supplier_ID are NOT NULL as they form the composite key. Supply_date is declared NOT NULL because every procurement record must identify when the supply arrangement was established — a supply record without a date provides no useful information for procurement tracking or reporting. Unit_cost is NOT NULL since a supply relationship always involves a price; CONSTRAINT CHK_Supplies_UnitCost ensures it is always positive, preventing free or negatively priced procurement entries from corrupting cost analysis and financial reporting.

The CUSTOMER table stores client data with Customer_ID (primary key), First_name, Last_name, Email, Join_date tracking relationship inception, and address components Street, City, Country, and Zip_code.

```
CREATE TABLE CUSTOMER (
    Customer_ID     VARCHAR(10)   NOT NULL,
    First_name      VARCHAR(50)     NOT NULL,
    Last_name       VARCHAR(50)     NOT NULL,
    Email           VARCHAR(100),
    Join_date       DATE  NOT NULL DEFAULT GETDATE(),
    Street          VARCHAR(100),
    City            VARCHAR(50),
    Country         VARCHAR(50),
    Zip_code        VARCHAR(10),
    CONSTRAINT PK_CUSTOMER PRIMARY KEY (Customer_ID),
    CONSTRAINT UQ_Customer_Email UNIQUE (Email)
);
```

Note: DEFAULT GETDATE() is SQL Server syntax. In MySQL, replace it with DEFAULT CURDATE().

CONSTRAINT PK_CUSTOMER uniquely identifies each customer record. First_name and Last_name are declared NOT NULL as they are essential identification attributes — a customer record without a name is operationally useless for order processing and communication. Email is optional, but CONSTRAINT UQ_Customer_Email ensures that when a value is provided, no two customers share the same email address, which is important for account management and marketing communications. Join_date is declared NOT NULL with DEFAULT GETDATE(), automatically recording the date a customer relationship begins when no explicit value is supplied — this supports customer loyalty analysis and relationship tracking. Address components are optional at the database level, allowing customer records to be created before full address details are collected, for example, during an initial registration. Phone numbers are not stored directly in this table — they are handled as a multivalued attribute in the separate CUSTOMER_PHONE table, presented later in this section.

The ORDER table captures purchase requests with Order_ID (primary key), Status tracking fulfilment progress (Pending, Processing, Shipped, Canceled), Order_date, Required_date, Shipped_date, Employee_ID (foreign key to EMPLOYEE identifying who processed the order), and Customer_ID (foreign key to CUSTOMER).

```
CREATE TABLE [ORDER] (
    Order_ID        VARCHAR(10)     NOT NULL,
    Status      VARCHAR(50)     NOT NULL DEFAULT 'Pending',
    Order_date      DATE    NOT NULL,
    Required_date   DATE,
    Shipped_date    DATE,
    Employee_ID     VARCHAR(255)    NOT NULL,
    Customer_ID     VARCHAR(10)     NOT NULL,
    CONSTRAINT PK_ORDER PRIMARY KEY (Order_ID),
    CONSTRAINT FK_Order_Employee FOREIGN KEY (Employee_ID) REFERENCES EMPLOYEE(Employee_ID),
    CONSTRAINT FK_Order_Customer FOREIGN KEY (Customer_ID) REFERENCES CUSTOMER(Customer_ID),
    CONSTRAINT CHK_Order_Status CHECK (Status IN ('Pending', 'Processing', 'Shipped', 'Canceled'))
);
```

Note: In SQL Server, use [ORDER]; in MySQL, use `ORDER` (backticks) to avoid conflict with the reserved keyword.

CONSTRAINT PK_ORDER uniquely identifies each purchase request using a VARCHAR identifier consistent with the order naming convention in the system. Order_date is NOT NULL since every order must record when it was placed. Status is NOT NULL with DEFAULT 'Pending', automatically assigning new orders the initial fulfilment state; CONSTRAINT CHK_Order_Status restricts it to the four valid values used throughout the system, preventing arbitrary strings from entering the fulfilment pipeline. Employee_ID is NOT NULL

because every order must be processed by an assigned employee; CONSTRAINT FK_Order_Employee enforces this referential link. Customer_ID is NOT NULL because every order must be placed by an existing customer; CONSTRAINT FK_Order_Customer enforces this referential link. Required_date and Shipped_date are optional — the required delivery date may not be specified at order creation, and the shipped date is only populated once the order is dispatched.

The ORDER_LINE table provides order detail lines with a composite primary key (Order_ID, Line_number), including Unit_price, Quantity, Discount, and Product_ID referencing the ordered product, allowing calculation of individual line totals.

```
CREATE TABLE ORDER_LINE (
    Order_ID       VARCHAR(10)    NOT NULL,
    Line_number    VARCHAR(10)    NOT NULL,
    Unit_price     DECIMAL(10,2)  NOT NULL,
    Quantity       INT            NOT NULL,
    Discount       DECIMAL(4,2)   NOT NULL DEFAULT 0,
    Product_ID     INT            NOT NULL,
    CONSTRAINT PK_ORDER_LINE PRIMARY KEY (Order_ID, Line_number),
    CONSTRAINT FK_OrderLine_Order FOREIGN KEY (Order_ID) REFERENCES
[ORDER](Order_ID),
    CONSTRAINT FK_OrderLine_Product FOREIGN KEY (Product_ID) REFERENCES
PRODUCT(Product_ID),
    CONSTRAINT CHK_OrderLine_UnitPrice CHECK (Unit_price > 0),
    CONSTRAINT CHK_OrderLine_Quantity CHECK (Quantity > 0),
    CONSTRAINT CHK_OrderLine_Discount CHECK (Discount >= 0 AND Discount <= 1)
);
```

Note: In MySQL, use `ORDER` (backticks) in the FK_OrderLine_Order reference.

CONSTRAINT PK_ORDER_LINE defines a composite primary key on (Order_ID, Line_number), uniquely identifying each line within an order — the same line identifier, such as 'Line1' can appear across different orders but must be unique within a single order. CONSTRAINT FK_OrderLine_Order enforces that every order line belongs to a valid existing order; CONSTRAINT FK_OrderLine_Product ensures every line references a valid product. Product_ID is NOT NULL since an order line without a product reference is meaningless. Unit_price and Quantity are NOT NULL; CONSTRAINT CHK_OrderLine_UnitPrice and CONSTRAINT CHK_OrderLine_Quantity ensure both are always positive. Discount is NOT NULL with DEFAULT 0, automatically setting new lines to no discount; CONSTRAINT CHK_OrderLine_Discount restricts values to the range 0 to 1, representing 0% to 100% discount expressed as a decimal fraction consistent with the values used throughout the system.

The three phone tables implement the multivalued Phone attribute for EMPLOYEE, CUSTOMER, and SUPPLIER, respectively, following the same pattern as the EMAIL table in the University system. Each stores phone numbers as separate rows linked to their parent entity through a foreign key, with the composite primary key preventing duplicate phone entries for the same entity.

```
CREATE TABLE EMPLOYEE_PHONE (
    Employee_ID    VARCHAR(255)    NOT NULL,
    Phone_number    VARCHAR(20)    NOT NULL,
    CONSTRAINT    PK_EMPLOYEE_PHONE    PRIMARY    KEY    (Employee_ID,
Phone_number),
    CONSTRAINT    FK_EmployeePhone_Employee    FOREIGN    KEY    (Employee_ID)
REFERENCES EMPLOYEE(Employee_ID)
);
```

CONSTRAINT PK_EMPLOYEE_PHONE defines a composite primary key on (Employee_ID, Phone_number), ensuring the same phone number cannot be recorded twice for the same employee while allowing each employee to have multiple distinct numbers. CONSTRAINT FK_EmployeePhone_Employee enforces referential integrity, guaranteeing that phone records can only exist for employees already present in the EMPLOYEE table.

```
CREATE TABLE CUSTOMER_PHONE (
    Customer_ID    VARCHAR(10)    NOT NULL,
    Phone_number    VARCHAR(20)    NOT NULL,
    CONSTRAINT    PK_CUSTOMER_PHONE    PRIMARY    KEY    (Customer_ID,
Phone_number),
    CONSTRAINT    FK_CustomerPhone_Customer    FOREIGN    KEY    (Customer_ID)
REFERENCES CUSTOMER(Customer_ID)
);
```

CONSTRAINT PK_CUSTOMER_PHONE defines a composite primary key on (Customer_ID, Phone_number), allowing each customer to have multiple contact numbers while preventing duplicate entries. CONSTRAINT FK_CustomerPhone_Customer ensures that phone records can only reference customers who already exist in the CUSTOMER table.

```
CREATE TABLE SUPPLIER_PHONE (
    Supplier_ID    VARCHAR(10)    NOT NULL,
    Phone_number    VARCHAR(20)    NOT NULL,
    CONSTRAINT PK_SUPPLIER_PHONE PRIMARY KEY (Supplier_ID, Phone_number),
    CONSTRAINT    FK_SupplierPhone_Supplier    FOREIGN    KEY    (Supplier_ID)
REFERENCES SUPPLIER(Supplier_ID)
);
```

CONSTRAINT PK_SUPPLIER_PHONE defines a composite primary key on (Supplier_ID, Phone_number), supporting multiple contact numbers per vendor while preventing duplicate entries. CONSTRAINT FK_SupplierPhone_Supplier ensures that phone records can only reference suppliers already present in the SUPPLIER table. In all three phone tables, Phone_number uses VARCHAR(20) to accommodate varying international formats, including country codes, area codes, and extension numbers.

This complete schema enables tracking the complete business flow from supplier procurement through inventory control to customer order fulfillment, supporting complex queries across organizational, product, and customer domains. The result is a database schema ready for data population via DML.

Attribute constraints and defaults operate at the column level, defining permissible values and behaviors. Domain constraints restrict attribute values to valid ranges or sets, implemented through CHECK constraints that specify Boolean conditions. In the University database, the FACULTY table enforces CHECK (Budget >= 0), ensuring budgets cannot be negative, while COURSE specifies CHECK (Credits >= 3 AND Credits <= 8), limiting credit hours to reasonable academic values. The Company database applies CHECK (Salary > 0) on EMPLOYEE and CHECK (Quantity_on_hand >= 0) on STORED_IN to prevent nonsensical negative inventories.

Default values provide automatic column population when insertions omit values:

University Database System

```
CREATE TABLE ENROLLS_IN (
    …
    Enrollment_date DATE DEFAULT GETDATE(),
    …
```

Company/Business Database

```
CREATE TABLE ORDER (
    …
    Status VARCHAR(50) DEFAULT 'Pending',
    …
```

automatically timestamps *enrollments* or marks Orders with *status* Pending.

Nullability constraints control whether columns can contain NULL values; NOT NULL enforces mandatory fields, such as STUDENT.First_name or EMPLOYEE.Email, while allowing NULL for optional attributes like STUDENT.Zip_code or EMPLOYEE.Warehouse_ID.

Key and referential integrity constraints maintain entity identification and relationship consistency.

Note: The following code snippets are simplified illustrative examples used to demonstrate constraint concepts. They use abbreviated syntax for clarity and do not include the full named constraint notation applied in the complete CREATE TABLE statements presented earlier in this section.

Primary keys uniquely identify tuples, enforced through PRIMARY KEY constraints that automatically imply UNIQUE and NOT NULL: CREATE TABLE STUDENT (Student_ID VARCHAR(255) PRIMARY KEY) assures each student has a distinct, non-null identifier. *Composite* primary keys span multiple columns: CREATE TABLE ENROLLS_IN (Student_ID VARCHAR(255), Course_ID VARCHAR(255), PRIMARY KEY (Student_ID, Course_ID)) uniquely identifies enrollments by student-course combination. *Foreign keys* establish referential integrity, ensuring values in one table match existing primary key values in another: FOREIGN KEY (Faculty_ID) REFERENCES FACULTY(Faculty_ID) in the STUDENT table prevents assigning students to non-existent faculties.

Cascade options define referential actions that can be added to foreign key constraints to control what happens when a referenced record is deleted or updated. In the schemas

implemented above, no explicit cascade option is specified, so the default ON DELETE RESTRICT behaviour applies — for example, deleting a WAREHOUSE record is blocked as long as STORED_IN records still reference it. If ON DELETE CASCADE were applied to FK_StoredIn_Warehouse, deleting a warehouse would automatically remove all its STORED_IN inventory records. ON DELETE SET NULL would instead preserve those records while setting their Warehouse_ID to NULL. Understanding these options allows designers to choose the referential action that best reflects business requirements when extending the schema.

Unique constraints enforce distinctness for non-key columns, enabling alternate keys: UNIQUE (Email) on PROFESSOR ensures no duplicate email addresses without making email the primary key. Assertions represent cross-table constraints that check conditions across multiple relations, though they are rarely implemented in practice; instead, CHECK constraints, triggers, or application logic verify complex rules such as "total enrollment per course cannot exceed room capacity" or "employee salary must not exceed department budget." These mechanisms jointly ensure databases maintain semantic correctness, automatically enforcing business rules and preventing data anomalies that might compromise system integrity.

This mapping preserves the relational model's set-based nature, preventing anomalies through normalization (e.g., 3NF), described in Chapter 10.

9.5 From Relational Algebra to SQL DML Queries

SQL's query capabilities are grounded in relational algebra, a formal mathematical system that defines operations on relations. Data Manipulation Language (DML) constitutes the subset of SQL commands responsible for inserting, updating, deleting, and querying data within database tables, representing the most frequently executed category of SQL operations. Unlike DDL, which modifies the structure, DML operates on the actual data content, making it central to day-to-day database operations. The importance of DML goes beyond simple CRUD (Create, Read, Update, Delete) operations; modern DML includes sophisticated query constructs that leverage set theory, predicate logic, and relational algebra to retrieve precisely the data needed for business operations and analytics. The conceptual basis for SQL queries lies in relational algebra, a mathematical schema defining operations on relations (tables). Each relational algebra operation has a corresponding SQL construct: the Selection operation (σ) filters rows based on conditions, mapped to SQL's WHERE clause; the Projection operation (π) selects specific columns, mapped to SQL's SELECT clause; the Cartesian Product operation (×) combines all rows from multiple tables, mapped to SQL's CROSS JOIN; the Join operation (⋈) combines related rows, mapped to SQL's various JOIN types; the Union operation (∪) combines results from multiple queries, mapped to SQL's UNION operator; and the Set Difference operation (-) identifies rows in one set but not another, mapped to SQL's EXCEPT or MINUS.

> **Definition: Data Manipulation Language (DML)**
>
> DML is a subset of SQL commands used to retrieve, insert, update, and delete data within database tables, primarily including SELECT, INSERT, UPDATE, and DELETE statements.

This mapping shows that SQL queries actually execute sophisticated mathematical operations on datasets. Query optimization, performed by the database engine, translates SQL's declarative specification (what data to retrieve) into an optimal execution plan (how to retrieve it), applying algebraic equivalences to minimize computational cost. For example, pushing selection operations down before joins (selecting rows early reduces the data volume for subsequent joins), reordering joins based on selectivity (joining smaller result sets first), and choosing appropriate join algorithms (nested loop, hash join, merge join) based on data properties and available indexes.

9.5.1 Working with Data in a Database

Working with data in databases involves three fundamental operations that manipulate table content: INSERT for adding new records, UPDATE for modifying existing records, and DELETE for removing records. These Data Manipulation Language (DML) operations form the core of database transactions, authorizing applications to create, modify, and delete data while maintaining data integrity through constraint enforcement and transaction management.

The INSERT statement adds new rows to tables, supporting two primary forms: single-row insertion using the VALUES clause and bulk insertion using SELECT statements. Single-row insertion follows the syntax:

```
INSERT INTO table_name (column1, column2, column3, ...)
  VALUES (value1, value2, value3, ...);
```

The column list specifies which attributes receive values, and the value list provides corresponding data. It is optional if values are provided for all columns in definition order, but explicit lists improve maintainability and protect against structure changes.

University Database System

For example, a SQL query to insert a statement in the FACULTY table:

```
INSERT INTO FACULTY (Faculty_ID, Name, Budget, Location)
  VALUES ('F001', 'Computer Science', 1500000.00, 'Main Campus');
```

adds a new faculty.

SQL query to insert student records:

```
INSERT INTO STUDENT (Student_ID, First_name, Last_name, Birth_date, Age,
Street, City, Country, Zip_code, Faculty_ID)
  VALUES ('S001', 'Sarah', 'Smith', '1995-08-25', 30, 'Jane Sandanski 32', 'Ohrid',
'Macedonia', '6000', 'F001');
```

Company/Business Database

SQL query to insert department records:

```
INSERT INTO DEPARTMENT (Department_ID, Dept_name, Budget, Location)
  VALUES ('D1', 'Sales', 500000.00, 'Skopje');
```

SQL query to insert employee records:

```
INSERT INTO EMPLOYEE (Employee_ID, First_name, Last_name, Email, Hire_date,
Salary, City, Department_ID)
  VALUES ('E1', 'Zoran', 'Pejovski', 'zoran.pejovski@email.com', '2020-01-15',
60000.00, 'Skopje', 'D1');
```

The UPDATE statement modifies existing records, allowing users to correct errors, reflect changes in real-world entities, or adjust values based on new information. Using syntax UPDATE table_name SET column1 = value1, column2 = value2 WHERE condition. The SET clause specifies which columns to change and their new values, while the WHERE clause determines affected rows. The WHERE clause is critical; omitting it updates all rows, often causing unintended widespread changes.

```
UPDATE table_name
  SET column1 = value1, column2 = value2, ...
  [WHERE <condition>];
```

University Database System

```
UPDATE PROFESSOR
  SET Academic_title = 'Full Professor'
  WHERE Hire_date < '2005-01-01';
```

This promotes to Full Professor all professors hired before January 1, 2005, reflecting a seniority-based academic rank advancement policy.

Company/Business Database

Updates can reference other tables in SQL Server using the FROM clause:

```
UPDATE EMPLOYEE
  SET E.Salary = E.Salary * 1.10
  FROM EMPLOYEE E
  JOIN DEPARTMENT D
  ON E.Department_ID = D.Department_ID
  WHERE D.Dept_name = 'Sales';
```

Updating employee salaries based on the department.

The DELETE statement removes unwanted or obsolete records, maintaining database accuracy and relevance. The DELETE statement removes rows using the DELETE FROM table_name WHERE condition syntax.

```
DELETE FROM table_name
  [WHERE <condition>];
```

The WHERE clause enables selective deletion; DELETE FROM ENROLLS_IN WHERE Grade = 5 removes failing enrollments while preserving passing ones.

University Database System

```
DELETE FROM ENROLLS_IN
  WHERE Grade = 5;
```

Without a WHERE clause, DELETE removes all rows while preserving the table structure, though TRUNCATE performs this more efficiently. All three operations should occur within transactions to preserve consistency, allowing rollback if errors occur and guaranteeing atomicity when multiple related changes must succeed or fail together, protecting database soundness during complex operations.

9.5.2 Basic SQL Queries

The purpose of the SELECT statement is to retrieve and display data from one or more database tables. This is a powerful SQL command capable of performing the equivalent of the projection operation (π) and selection operation (σ), filtering tuples (rows) based on a predicate condition, retaining only those satisfying the criteria. The order of clauses in the SELECT statement cannot be changed. The two mandatory clauses are the first two SELECT and FROM; the rest of them are optional, as the statement syntax shows:

```
SELECT [DISTINCT|ALL] {*|[<column1> [AS newName]] [,...]}
  FROM TableName [alias] [,…]
  [WHERE <condition>]
  [GROUP BY <grouping condition> [,...]]
  [HAVING <grouping condition>]
  [ORDER BY <column1> [,...]];
```

The sequence of **processing** in the SELECT statement is:

FROM: specifies the table or tables to be used
WHERE: filters the rows subject to some condition
GROUP BY: forms groups of rows with the same column value
HAVING: filters the groups subject to some condition
SELECT: specifies which columns are to appear in the output
ORDER BY: specifies the order of the output

The SELECT clause specifies which columns to retrieve, supporting column names, expressions, aggregate functions, and asterisk (*) for all columns, though SELECT * is discouraged in production code as it couples queries to table structure and retrieves potentially unnecessary data.

<column1> can represent a column name or an expression; the TableName is the name of the existing table or view that you have to access.

The DISTINCT Operation, as described in *subsection 8.4.4,* eliminates duplicates from results.

The FROM clause identifies source tables, allowing multiple tables to be joined.

The WHERE clause filters rows based on complex Boolean expressions combining conditions with AND, OR, and NOT operators, comparison operators (=, <>, <, <=, >, >=), and special operators (LIKE for pattern matching, BETWEEN for ranges, IN for set membership, IS NULL for null checking).

The ORDER BY clause sorts results: ORDER BY Last_name, First_name sorts by last name, then first name for ties, with DESC keyword for descending order: ORDER BY Budget DESC.

Throughout this chapter, we present numbered relational SQL queries (**query9.1**, **query9.2**, etc.) that demonstrate practical applications of each operation. Each query requirement is followed by its relational SQL statement and results.

University Database System

query9.1: *List all faculties with a budget greater than 1,000,000.*

The query selects the Faculty_ID, Name, Budget, and Location columns from the FACULTY table and filters rows where the budget is greater than one million. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
  Faculty_ID,
  Name,
  Budget,
  Location
  FROM  FACULTY
  WHERE Budget > 1000000.00;
```

The database query optimizer analyzes WHERE clause predicates to determine efficient execution plans, potentially using indexes to avoid full-table scans.

The relational algebra expression defined in req8.1 maps directly to the mentioned SQL statement. The results of the selection operation match the output of the SQL query 9.1, as shown in Figure 9.1.

	Faculty_ID	Name	Budget	Location
1	F001	Computer Science	1,500,000	Main Campus
2	F002	Electrical Engineeri	1,200,000	East Wing
3	F004	Bioengineering	1,100,000	North Building

Figure 9.1: Result output of the SQL query 9.1

query9.2: *List all students from Ohrid who are older than 21.*

The query selects all columns from the STUDENT table, filtering for rows where the age is greater than 21 and the city is Ohrid. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT *
  FROM STUDENT
  WHERE Age > 21 AND City = 'Ohrid';
```

Note: SELECT * is used here intentionally to return all student attributes, consistent with the relational algebra selection operation (σ) which preserves the complete schema of the original relation. In production code, it is recommended to explicitly list only the required columns for better performance and maintainability.

The relational algebra expression defined in req8.2 maps directly to the mentioned SQL statement. The results of the selection operation match the output of the SQL query 9.2, as shown in Figure 9.2.

	Student_ID	First_name	Last_name	Birth_date	Age	Street	City	Country	Zip_code	Faculty_ID
1	S001	Sarah	Smith	1995-08-25	30	Jane Sandanski 32	Ohrid	Macedonia	6000	F001
2	S005	Elena	Mitan	2003-07-01	22	Turisticka 22	Ohrid	Macedonia	6000	F005

Figure 9.2: Result output of the SQL query 9.2

Company/Business Database

query9.3: *List all departments with a budget greater than 300,000.*

The SQL query selects the Department_ID, Dept_name, Location, and Budget columns from the DEPARTMENT table, filtering rows where the Budget column is greater than 300,000. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
  Department_ID,
  Dept_name,
  Location,
  Budget
  FROM    DEPARTMENT
  WHERE   Budget > 300000.00;
```

The relational algebra expression defined in req8.3 maps directly to the given SQL statement. The results of the selection operation match the output of the SQL query 9.3, as shown in Figure 9.3.

	Department_ID	Dept_name	Location	Budget
1	D1	Sales	Skopje	500,000

Figure 9.3: Result output of the SQL query 9.3

query9.4: *List all warehouses with a capacity of less than or equal to 5,000, located in Ohrid.*

The SQL query selects the Warehouse_ID, Location, Capacity and Department_ID columns from the WAREHOUSE table, filtering rows with capacity less than or equal to 5,000 and located in Ohrid. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
  Warehouse_ID,
  Location,
  Capacity,
  Department_ID
  FROM    WAREHOUSE
  WHERE   Capacity <= 5000.00 AND Location = 'Ohrid';
```

The relational algebra expression defined in **req8.4** maps directly to the given SQL statement. The results of the **selection** operation match the output of the SQL query 9.4, as shown in Figure 9.4.

	Warehouse_ID	Location	Capacity	Department_ID
1	W3	Ohrid	5,000	D3

Figure 9.4: Result output of the SQL query 9.4

University Database System

query9.5: *List the professor ID, first name, last name, academic title, hire date, and Faculty ID from the PROFESSOR table.*

The SQL query selects the Professor_ID, First_name, Last_name, Academic_title, Hire_date and Faculty_ID columns from the PROFESSOR table. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
  Professor_ID,
  First_name,
  Last_name,
  Academic_title,
  Hire_date,
  Faculty_ID
  FROM PROFESSOR;
```

The relational algebra expression in **req8.5** maps directly to the given SQL statement. The projection results match the output of SQL query 9.5, as shown in Figure 9.5.

	Professor_ID	First_name	Last_name	Academic_title	Hire_date	Faculty_ID
1	P001	Kliment	Ohridski	Full Professor	2007-08-01	F001
2	P002	Naum	Ohridski	Associate Professor	2004-01-15	F002
3	P003	Kiril	Prosvetitel	Full Professor	2001-09-25	F004
4	P004	Metodij	Prosvetitel	Full Professor	2002-09-13	F001
5	P005	Blaze	Koneski	Assistant Professor	2012-10-19	F005
6	P006	Richard	Johnson	Associate Professor	2022-09-02	F001

Figure 9.5: Result output of the SQL query 9.5

Company/Business Database

query9.6: *List the customers by first name, last name, city and join date.*

The SQL query selects the First_name, Last_name, City and Join_date columns from the CUSTOMER table. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
  First_name,
  Last_name,
  City,
  Join_date
  FROM CUSTOMER;
```

The relational algebra expression defined in **req8.6** maps directly to the given SQL statement. The results of the SQL query 9.6 are shown in Figure 9.6.

	First_name	Last_name	City	Join_date
1	Ana	Ivanovska	Skopje	2022-01-10
2	Petar	Petrovski	Bitola	2022-03-15
3	Marija	Georgievska	Ohrid	2022-05-20
4	Nikola	Nikolovski	Prilep	2022-07-25
5	Elena	Trajkovska	Tetovo	2022-09-30
6	Goran	Stojanovski	Kumanovo	2023-01-05
7	Ivana	Jovanovska	Veles	2023-03-12

Figure 9.6: Result output of the SQL query 9.6

Renaming of the table names, attributes and results with shorter names by creating an alias for each feature to avoid repeated typing of long names. The alias can follow the keyword AS, as shown in query 9.8, or it can follow the relation name directly. We can use this alias-naming or renaming mechanism in any SQL query to specify tuple variables for every table in the WHERE clause, whether or not the same relation.

University Database System

query9.7: *Retrieve from the PROFESSOR table the first name, last name, and academic title, renaming the output columns as Acc_first_name, Acc_last_name, and Academic_Title.*

The SQL query selects the First_name, renamed as *Acc_first_name,* Last_name, renamed as *Acc_last_name*, and *Academic_title, renamed* as *Academic_Title* columns from the PROFESSOR table. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
  First_name AS Acc_first_name,
  Last_name AS Acc_last_name,
  Academic_title AS Academic_Title
  FROM PROFESSOR;
```

The relational algebra expression defined in **req8.7** maps directly to the given SQL statement. The results of the **rename** SQL query 9.7 are shown in Figure 9.7.

	Acc_first_name	Acc_last_name	Academic_Title
1	Kliment	Ohridski	Full Professor
2	Naum	Ohridski	Associate Professor
3	Kiril	Prosvetitel	Full Professor
4	Metodij	Prosvetitel	Full Professor
5	Blaze	Koneski	Assistant Professor
6	Richard	Johnson	Associate Professor

Figure 9.7: Result output of the SQL query 9.7

Company/Business Database

query9.8: *Retrieve from the PRODUCT table the product name, description, category, price, and weight, renaming the output columns as Item_name, Item_description, Category, Price, and Weight.*

The SQL query selects the Product_name, renamed as *Item_name,* Description, renamed as *Item_description,* Category, Price, and Weight columns from the PRODUCT table. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
  Product_name AS Item_name,
  Description AS Item_description,
  Category,
  Price,
  Weight
  FROM PRODUCT;
```

The relational algebra expression defined in **req8.8** maps directly to the given SQL statement. The results of the **rename** SQL query 9.8 are shown in Figure 9.8.

	Item_name	Item_description	Category	Price	Weight
1	Ajvar	Traditional Macedonian spread	Food	372	0.5
2	Tavce Gravce	Baked beans dish	Food	434	0.7
3	Rakija	Fruit brandy	Beverage	682	1
4	Mastika	Mastic-flavored liqueur	Beverage	930	1
5	Sirenje	White cheese	Dairy	465	0.4
6	Kajmak	Creamy dairy product	Dairy	620	0.3
7	Ohrid Pearl	Jewelry	Accessories	9,300	0.1
8	Macedonian Ruby	Gemstone	Accessories	31,000	0.05
9	Filigree	Traditional craft	Souvenir	4,650	0.2
10	Galichnik Wedding Cost	Traditional clothing	Apparel	74,400	2.5
11	Vranec Wine	Red wine	Beverage	1,116	1.5
12	Selsko Meso	Village-style meat dish	Food	868	0.8

Figure 9.8: Result output of the SQL query 9.8

9.5.3 Substring Pattern Matching, Concatenation, and Arithmetic Operations

Pattern matching in SQL enables flexible text searching through the LIKE operator, which accepts wildcard characters for partial matches. The percent sign (%) represents zero or more characters, while the underscore (_) matches exactly one character. For example, WHERE Last_name LIKE '%ski' finds students whose surnames end with the last characters 'ski', while WHERE Title LIKE '%Programming%' finds all Programming courses. The NOT LIKE negates patterns: WHERE Country NOT LIKE 'Macedonia' considers foreign students, not from 'Macedonia'.

The IN operator tests whether a value matches any element in a specified list, simplifying queries requiring multiple OR conditions. The IN operator accepts subqueries, enabling dynamic lists. For example, in companies, the ORDER relation WHERE Status IN ('Processing', 'Pending') identifies the specific statuses of the orders that are in a transition state.

The BETWEEN operator simplifies range queries by testing whether values fall within inclusive bounds. WHERE Salary BETWEEN 30000 AND 50000 selects employees with a

salary range from 30000 through 50000, equivalent to Salary >= 30000 AND Salary <= 50000, but more readable.

The ANY operator, when combined with subqueries, enables comparisons against sets of values returned by nested SELECT statements. WHERE Department_ID = ANY (SELECT Department_ID FROM DEPARTMENT WHERE Dept_name IN ('Sales', 'Marketing')) finds Department_ID of an employee that matches any ID returned by the subquery (Sales or Marketing), the condition is met.

University Database System

query9.9: *Find Programming courses.*

The SQL query selects the Course_ID, Title, Credits, and Description columns from the COURSE table, and filters only the courses that have the keyword 'Programming' in the title. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
  Course_ID,
  Title,
  Credits,
  Description
  FROM COURSE
  WHERE Title LIKE '%Programming%';
```

The results of the SQL query 9.9 are shown in Figure 9.9.

	Course_ID	Title	Credits	Description
1	CS101	Intro to Programming	5	Fundamentals of coding
2	CS201	Advanced Programming	6	Object-oriented programming and data structures

Figure 9.9: Result output of the SQL query 9.9

query9.10: *Find students with last names ending in 'ski'.*

The SQL query selects the Student_ID, First_name, Last_name, and City columns from the STUDENT table and filters students whose surnames end with the suffix 'ski'. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT Student_ID, First_name, Last_name, City
  FROM STUDENT
  WHERE Last_name LIKE '%ski';
```

The results of the SQL query 9.10 are shown in Figure 9.10.

	Student_ID	First_name	Last_name	City
1	S003	Kosta	Dojcinoski	Resen
2	S004	David	Boskovski	Skopje

Figure 9.10: Result output of the SQL query 9.10

query9.11: *Find the Student_ID, first name, last name, age, and city only for foreign students. Format output as "Student_ID, First_name Last_name is Age old and is from City".*

```
SELECT
    Student_ID,
    CONCAT(First_name,' ',Last_name,' is ',Age,' old and is from ', City) as Student_Description
        FROM STUDENT
        WHERE Country NOT LIKE 'Macedonia';
```

The results of the SQL query 9.11 are shown in Figure 9.11.

	Student_ID	Student_Description
1	S002	Bojan Jovansen is 23 old and is from Malmö
2	S006	Anica Stojnasen is 22 old and is from Malmö

Figure 9.11: Result output of the SQL query 9.11

Company/Business Database

query9.12: *List Employee_ID, first name, last_name, and date of employment, only for employees that are hired in 2022.*

The SQL query selects the Employee_ID, First_name, Last_name, and Hire_date columns from the EMPLOYEE table, filters only the dates from 2022, underscore (_) match exactly the number of characters in the field Hire_date. Both SQL Server and MySQL use very similar syntax for this basic SQL statement

```
SELECT
  Employee_ID,
  First_name,
  Last_name,
  Hire_date
  FROM EMPLOYEE
  WHERE Hire_date LIKE '2022_____';
```

Note: Using LIKE on a DATE column relies on implicit string conversion and is SQL Server-specific. In MySQL, use WHERE YEAR(Hire_date) = 2022 for portability.

The results of the SQL query 9.12 are shown in Figure 9.12.

	Employee_ID	First_name	Last_name	Hire_date
1	E3	Marija	Naumoska	2022-05-10
2	E7	Dimitar	Tocko	2022-02-28

Figure 9.12: Result output of the SQL query 9.12

query9.13: *Find orders and calculate a simulated tax (an 18% tax increase) for each item, only for any orders that are currently in a transition state (either 'Processing' or 'Pending').*

The SQL query selects Order_ID and Line_number, Unit_price, and calculates the taxed amount by applying an 18% tax increase to the original Unit_price from the ORDER_LINE table, but only for orders in a transition state. This filtering is done in a separate subquery using the IN operator. SQL Server and MySQL use very similar syntax for this basic SQL statement. The ROUND operator rounds the result to a specified number of decimal places; in this case, it rounds to an integer with zero decimal places. Both SQL Server and MySQL

use very similar syntax for this basic SQL statement. Exception for ORDER is a reserved keyword in MySQL, is enclosed in backticks `ORDER`, while in SQL SERVER, we use square brackets [ORDER]).

```
SELECT
  Order_ID,
  Line_number,
  Unit_price,
  ROUND(Unit_price * 1.18, 0) AS Unit_Price_with_Tax
  FROM ORDER_LINE
  WHERE Order_ID = ANY
        (SELECT Order_ID
        FROM ORDER
        WHERE Status IN ('Processing', 'Pending'));
```

The results of the SQL query 9.13 for matching records are shown in Figure 9.13.

	Order_ID	Line_number	Unit_price	Unit_Price_with_Tax
1	3	1	930	1,097
2	5	3	620	732
3	9	2	1,116	1,317
4	11	2	74,400	87,792

Figure 9.13: Result output of the SQL query 9.13

query9.14 *List Employee_ID, first name, last_name, and Salary for employees with a salary between 30000 and 50000 that are working in the Sales or Marketing department.*

To retrieve employees who meet your salary and department criteria, use the BETWEEN operator for the numeric range and the ANY operator with a subquery to filter by department. The ANY operator compares a scalar value against a single-column set of values returned by a subquery. If an employee's Department_ID matches any ID returned by the subquery (Sales or Marketing), the condition is met. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
  Employee_ID,
  First_name,
  Last_name,
  Salary
  FROM EMPLOYEE
  WHERE Salary BETWEEN 30000 AND 50000
        AND Department_ID = ANY (
              SELECT Department_ID
               FROM DEPARTMENT
               WHERE Dept_name IN ('Sales', 'Marketing'));
```

The results of the SQL query 9.14 are shown in Figure 9.14.

	Employee_ID	First_name	Last_name	Salary
1	E3	Marija	Naumoska	50,000
2	E4	Anita	Toshevska	45,000
3	E7	Dimitar	Tocko	30,000

Figure 9.14: Result output of the SQL query 9.14

9.5.4 SQL Multiset Comparison and Ordering of Query Results

SQL treats a table not as a set but rather as a multi-set, meaning duplicate records can appear more than once in the table. By default, SQL allows duplicate tuples in results, using the SELECT ALL column statement, but the DISTINCT keyword enforces set semantics by eliminating duplicates.

Company/Business Database

query9.15: *Find status tracking codes of the orders that keep purchase requests from customers.*

The SQL query to find status tracking codes for the ORDER table is a basic SELECT Status statement that lists all orders with their associated status. This is identical to writing the SELECT ALL Status statement (query 9.15 a). Exception for ORDER is a reserved keyword in MySQL, is enclosed in backticks `ORDER`, while in SQL SERVER, we use square brackets [ORDER]).

```
SELECT
  ALL Status
  FROM [ORDER];
```

The results of the SQL query 9.15 a) with ALL are shown in Figure 9.15 a).

Identifying distinct status codes with the DISTINCT keyword eliminates duplicates and returns only the **unique** values from the ORDER relation (query 9.15b). The ORDER keyword is reserved in MySQL, so it's enclosed in backticks.

```
SELECT
  DISTINCT (Status)
  FROM [ORDER];
```

The relational algebra expression defined in req8.10 maps directly to the mentioned SQL statement. The results of the SQL query 9.15 b) with DISTINCT are shown in Figure 9.15 b).

	Status
1	Shipped
2	Shipped
3	Pending
4	Shipped
5	Processing
6	Shipped
7	Canceled
8	Shipped
9	Pending
10	Shipped
11	Processing

a)

	Status
1	Shipped
2	Pending
3	Processing
4	Canceled

b)

Figure 9.15: Result output of the SQL query 9.15 a) with ALL and b) with DISTINCT

University Database System

query9.16: *Find all unique Student_ID from the ENROLLS_IN relation.*

By default, SQL allows duplicate tuples in results (making it a bag or multiset rather than a true set), but the DISTINCT keyword enforces set semantics by eliminating duplicates, which returns each student_id once; in addition to that, there are multiple student enrollments to different courses. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
  DISTINCT Student_ID
  FROM ENROLLS_IN;
```

The relational algebra expression defined in req8.9 maps directly to the mentioned SQL statement. The results of the SQL query 9.16 are shown in Figure 9.16.

	Student_ID
1	S006
2	S001
3	S002
4	S005
5	S003
6	S004

Figure 9.16: Result output of the SQL query 9.16

The **set** operations provide a robust framework for synthesizing data across multiple tables. These operations allow developers to treat entire relations as mathematical sets, enabling complex data retrieval through intuitive logical combinations. The UNION operator serves as the foundational operator for vertical integration, merging the results of two queries into a single result set containing all unique tuples. By default, SQL's UNION performs an implicit distinct operation, effectively eliminating duplicates to maintain set integrity. When the goal is to identify commonality rather than accumulation, the INTERSECT

operator is utilized. It filters the data to return only tuples that exist in both relations, acting as the logical "AND" in the relational model. Conversely, the EXCEPT operator (often implemented as MINUS in certain dialects) facilitates data exclusion. It returns tuples present in the first relation that do not appear in the second, making it indispensable for identifying gaps or "missing" records between datasets.

While the aforementioned operations require "union-compatible" schemas (identical column counts and data types), the Cartesian Product operates on a different expansion principle. By simply listing multiple tables in the FROM clause, separated by commas, without a join condition, SQL generates a cross product. This mathematical pairing links every single tuple from the first relation with every tuple from the second. While powerful for generating exhaustive combinations, it is computationally expensive, as the resulting set size is the product of the row counts of both tables. Together, these operations form the backbone of sophisticated data manipulation.

Ordering query results in SQL provides control over how retrieved data is presented, transforming unordered result sets into sorted sequences that enable analysis and reporting. The ORDER BY clause specifies sorting criteria, accepting column names or expressions with optional ASC (ascending, default) or DESC (descending) keywords.

SQL's query execution conceptually proceeds as: FROM (identify source tables) → WHERE (filter rows) → SELECT (project columns) → ORDER BY (sort results), though actual execution plans may reorder operations for optimization. Understanding this logical flow helps predict query behavior and diagnose unexpected results.

University Database System

query9.17: *Find all Faculty IDs for faculties that are either defined in the FACULTY or have an affiliated professor in the PROFESSOR tables.*

The union operator combines the two lists, from the FACULTY relation to get a list of all unique Faculty IDs and the PROFESSOR relation to get a list of all unique Faculty IDs that have an affiliated professor. The final output has eliminated duplicates. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT Faculty_ID
  FROM FACULTY
UNION
SELECT Faculty_ID
  FROM PROFESSOR;
```

The relational algebra expression defined in req8.11 maps directly to the SQL statement mentioned. The results of the SQL query 9.17 are shown in Figure 9.17.

	A-Z Faculty_ID
1	F001
2	F002
3	F003
4	F004
5	F005

Figure 9.17: Result output of the SQL query 9.17

query9.18: *List all people (students and professors) with their First_name and Last_name.*

The UNION operator must ensure that the table attributes (First_name and Last_name) are union-compatible. This means they must have the same number of attributes, and corresponding attributes must have compatible data types. The SQL query finds First_name and Last_name from the STUDENT and PROFESSOR tables. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT First_name, Last_name
  FROM STUDENT
UNION
SELECT First_name, Last_name
  FROM PROFESSOR;
```

Notice the results in Figure 9.18 contain mixed first and last names of students and professors. The relational algebra expression defined in req8.12 does not guarantee any specific ordering unless explicitly specified using a sorting operation.

	First_name	Last_name
1	Sarah	Smith
2	Bojan	Jovansen
3	Kosta	Dojcinoski
4	David	Boskovski
5	Elena	Mitan
6	Anica	Stojnasen
7	Kliment	Ohridski
8	Naum	Ohridski
9	Kiril	Prosvetitel
10	Metodij	Prosvetitel
11	Blaze	Koneski
12	Richard	Johnson

Figure 9.18: Result output of the SQL query 9.18

query9.19: *List all people (students and professors) with their Last_name and First_name.*

This multi-column sorting enables hierarchical ordering where subsequent columns break ties in preceding columns. The SQL query can be customized by using ORDER BY Last_name, First_name to sort by last name first, then by first name. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
(SELECT First_name, Last_name
  FROM STUDENT
UNION
SELECT First_name, Last_name
  FROM PROFESSOR)
  ORDER BY Last_name, First_name;
```

The results of the SQL query 9.19 are sorted as shown in Figure 9.19.

	First_name	Last_name
1	David	Boskovski
2	Kosta	Dojcinoski
3	Richard	Johnson
4	Bojan	Jovansen
5	Blaze	Koneski
6	Elena	Mitan
7	Kliment	Ohridski
8	Naum	Ohridski
9	Kiril	Prosvetitel
10	Metodij	Prosvetitel
11	Sarah	Smith
12	Anica	Stojnasen

Figure 9.19: Result output of the SQL query 9.19

Company/Business Database

query9.20: List all the cities alphabetically where there are employees or customers who live.

The relational algebra expression defined in req8.13 does not guarantee any specific ordering unless explicitly specified using a sorting operation. So to list all cities alphabetically where there are employees and customers uses UNION operator, we add the ORDER BY City SQL operation. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
(SELECT City
  FROM EMPLOYEE
UNION
SELECT City
  FROM CUSTOMER)
  ORDER BY City;
```

The results of the SQL query 9.20 are shown in Figure 9.20.

	City
1	Bitola
2	Kumanovo
3	Ohrid
4	Prilep
5	Skopje
6	Tetovo
7	Veles

Figure 9.20: Result output of the SQL query 9.20

University Database System

query9.21: *Find the ISBNs of all textbooks that have been assigned to a course.*

This query returns the intersection of all ISBNs in the TEXTBOOK and ASSIGNMENT relations. The TEXTBOOK and ASSIGNMENT relations are union-compatible on the ISBN attribute. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT ISBN
  FROM TEXTBOOK
INTERSECT
SELECT ISBN
  FROM ASSIGNMENT;
```

The relational algebra expression defined in req8.14 maps directly to the SQL statement mentioned. The results of the SQL query 9.21 are shown in Figure 9.21. Notice the textbook '978-0675207379' (Academic writing by Wilson, J, R.) is present in the TEXTBOOK table but is not in the ASSIGNMENT table. Therefore, it is excluded from the intersection result.

	A-Z ISBN
1	978-0131498642
2	978-0134080185
3	978-0134093413
4	978-0321286433
5	978-0471754471
6	978-0571754171
7	978-0834080185
8	978-2134093493
9	978-6321286413
10	978-7134082853
11	978-7821286423

Figure 9.21: Result output of the SQL query 9.21

Company/Business Database

query9.22: List all of the cities where there are both customers and employees.

This query finds all cities where both customers and employees are located, using the INTERSECT operator to return only values common to both queries.

```
SELECT City
  FROM EMPLOYEE
INTERSECT
SELECT City
  FROM CUSTOMER;
```

The relational algebra expression defined in req8.15 maps directly to the SQL statement mentioned. The results of the SQL query 9.22 are shown in Figure 9.22. Note that there are no employees from Kumanovo or Veles; only customers from these cities, so the results for the intersection operation are omitted.

	City
1	Skopje
2	Bitola
3	Ohrid
4	Prilep
5	Tetovo

Figure 9.22: Result output of the SQL query 9.22

University Database System

query9.23: *Find faculties that have no affiliated professors.*

The EXCEPT operator returns tuples present in the FACULTY table that do not appear in the PROFESSOR table, which helps to identify gaps or "missing" records between relations.

Both SQL Server and MySQL 8.0.31+ use the EXCEPT operator to perform the set difference operation.

```
SELECT Faculty_ID
  FROM FACULTY
EXCEPT
SELECT Faculty_ID
  FROM PROFESSOR;
```

Older versions of MySQL do not support the EXCEPT operator to perform the set difference operation; it uses a WHERE clause to filter out the unwanted records.

```
SELECT Faculty_ID
FROM FACULTY
WHERE Faculty_ID NOT IN (
    SELECT Faculty_ID
    FROM PROFESSOR
    WHERE Faculty_ID IS NOT NULL);
```

The relational algebra expression defined in req8.16 maps directly to the SQL statement mentioned. The results of the SQL query 9.23 are shown in Figure 9.23.

	Faculty_ID
1	F003

Figure 9.23: Result output of the SQL query 9.23

Company/Business Database

query9.24: *List all of the cities where there are customers but not employees*.

The EXCEPT operator returns a list of all cities where there are customers but not employees. Both SQL Server and MySQL 8.0.31+ use the EXCEPT operator to perform the set difference operation.

```
SELECT City
  FROM CUSTOMER
EXCEPT
SELECT City
  FROM EMPLOYEE;
```

Older versions of MySQL do not support the EXCEPT operator to perform the set difference operation; it uses a WHERE clause to filter out the unwanted records.

```
SELECT City
FROM CUSTOMER
WHERE City NOT IN (
    SELECT City
    FROM EMPLOYEE);
```

The relational algebra expression defined in req8.17 maps directly to the SQL statement mentioned. The results of the SQL query 9.24 are shown in Figure 9.24.

	A-Z City
1	Kumanovo
2	Veles

Figure 9.24: Result output of the SQL query 9.24

University Database System

query9.25: *Find all of the FACULTY Name and Location with affiliated PROFESSOR first name, last name, and hire date, using the Cartesian product operation.*

SQL generates a cross product of the FACULTY and PROFESSOR tables. Both SQL Server and MySQL use the same syntax for a Cartesian product: simply listing the tables in the FROM clause separated by a comma.

```
SELECT
        F.Name AS Faculty_Name,
        F.Location AS Faculty_Location,
        P.First_name AS Professor_First_name,
        P.Last_name AS Professor_Last_name,
        P.Hire_date
FROM FACULTY AS F, PROFESSOR AS P;
```

The relational algebra expression defined in req8.18 maps directly to the SQL statement mentioned. This results in every row from the FACULTY table being combined with every row from the PROFESSOR table, and then only the specified columns are selected, shown in Figure 9.25.

Company/Business Database

query9.26: *List all of the employees' first name, last name, city where they work, department name, and location, using the Cartesian product operation*.

SQL generates a cross product of EMPLOYEE and DEPARTMENT tables. Both SQL Server and MySQL use the same syntax for a Cartesian product.

```
SELECT
    E.First_name,
    E.Last_name,
    E.City,
    D.Dept_name,
    D.Location
FROM EMPLOYEE AS E, DEPARTMENT AS D;
```

The relational algebra expression defined in req8.19 maps directly to the SQL statement mentioned. The results of the SQL query 9.26 are shown in Figure 9.26.

	Faculty_Name	Faculty_Location	Professor_First_name	Professor_Last_name	Hire_date
1	Humanities	South Building	Kliment	Ohridski	2007-08-01
2	Bioengineering	North Building	Kliment	Ohridski	2007-08-01
3	Economy & Tourism	West Wing	Kliment	Ohridski	2007-08-01
4	Electrical Engineering	East Wing	Kliment	Ohridski	2007-08-01
5	Computer Science	Main Campus	Kliment	Ohridski	2007-08-01
6	Humanities	South Building	Naum	Ohridski	2004-01-15
7	Bioengineering	North Building	Naum	Ohridski	2004-01-15
8	Economy & Tourism	West Wing	Naum	Ohridski	2004-01-15
9	Electrical Engineering	East Wing	Naum	Ohridski	2004-01-15
10	Computer Science	Main Campus	Naum	Ohridski	2004-01-15
11	Humanities	South Building	Kiril	Prosvetitel	2001-09-25
12	Bioengineering	North Building	Kiril	Prosvetitel	2001-09-25
13	Economy & Tourism	West Wing	Kiril	Prosvetitel	2001-09-25
14	Electrical Engineering	East Wing	Kiril	Prosvetitel	2001-09-25
15	Computer Science	Main Campus	Kiril	Prosvetitel	2001-09-25
16	Humanities	South Building	Metodij	Prosvetitel	2002-09-13
17	Bioengineering	North Building	Metodij	Prosvetitel	2002-09-13
18	Economy & Tourism	West Wing	Metodij	Prosvetitel	2002-09-13
19	Electrical Engineering	East Wing	Metodij	Prosvetitel	2002-09-13
20	Computer Science	Main Campus	Metodij	Prosvetitel	2002-09-13
21	Humanities	South Building	Blaze	Koneski	2012-10-19
22	Bioengineering	North Building	Blaze	Koneski	2012-10-19
23	Economy & Tourism	West Wing	Blaze	Koneski	2012-10-19
24	Electrical Engineering	East Wing	Blaze	Koneski	2012-10-19
25	Computer Science	Main Campus	Blaze	Koneski	2012-10-19
26	Humanities	South Building	Richard	Johnson	2022-09-02
27	Bioengineering	North Building	Richard	Johnson	2022-09-02
28	Economy & Tourism	West Wing	Richard	Johnson	2022-09-02
29	Electrical Engineering	East Wing	Richard	Johnson	2022-09-02
30	Computer Science	Main Campus	Richard	Johnson	2022-09-02

Figure 9.25: Result output of the SQL query 9.25

	First_name	Last_name	City	Dept_name	Location
1	Zoran	Pejovski	Skopje	Human Resources	Ohrid
2	Zoran	Pejovski	Skopje	Marketing	Bitola
3	Zoran	Pejovski	Skopje	Sales	Skopje
4	Nikola	Delcevski	Skopje	Human Resources	Ohrid
5	Nikola	Delcevski	Skopje	Marketing	Bitola
6	Nikola	Delcevski	Skopje	Sales	Skopje
7	Marija	Naumoska	Bitola	Human Resources	Ohrid
8	Marija	Naumoska	Bitola	Marketing	Bitola
9	Marija	Naumoska	Bitola	Sales	Skopje
10	Anita	Toshevska	Bitola	Human Resources	Ohrid
11	Anita	Toshevska	Bitola	Marketing	Bitola
12	Anita	Toshevska	Bitola	Sales	Skopje
13	Naum	Stojanoski	Ohrid	Human Resources	Ohrid
14	Naum	Stojanoski	Ohrid	Marketing	Bitola
15	Naum	Stojanoski	Ohrid	Sales	Skopje
16	Hristina	Miloseska	Ohrid	Human Resources	Ohrid
17	Hristina	Miloseska	Ohrid	Marketing	Bitola
18	Hristina	Miloseska	Ohrid	Sales	Skopje
19	Dimitar	Tocko	Prilep	Human Resources	Ohrid
20	Dimitar	Tocko	Prilep	Marketing	Bitola
21	Dimitar	Tocko	Prilep	Sales	Skopje
22	Bijana	Osmani	Tetovo	Human Resources	Ohrid
23	Bijana	Osmani	Tetovo	Marketing	Bitola
24	Bijana	Osmani	Tetovo	Sales	Skopje

Figure 9.26: Result output of the SQL query 9.26

9.5.5 The EXISTS function in SQL

EXISTS is a Boolean function that returns TRUE or FALSE; hence, it can be used in a WHERE clause condition. The EXISTS function in SQL is used to check whether the result of a nested query is empty (contains no tuples) or not. The result of EXISTS is a Boolean value TRUE if the nested query result contains at least one tuple, or FALSE if the nested query result contains no tuples.

University Database System

query9.27: *List the faculties that have affiliated professors*.

To list the names of faculties that have at least one affiliated professor using the EXISTS function, you perform a query on the FACULTY table with a subquery that checks for the presence of matching records in the PROFESSOR table. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
F.Faculty_ID,
F.Name
FROM FACULTY F
WHERE EXISTS (
   SELECT *
   FROM PROFESSOR P
   WHERE P.Faculty_ID = F.Faculty_ID);
```

The results of the SQL query 9.27 are shown in Figure 9.27. Note that the faculty "Economy & Tourism" (F003) is excluded from the results because there are no professors assigned to this faculty ID in the PROFESSOR table.

	Faculty_ID	Name
1	F001	Computer Science
2	F002	Electrical Engineering
3	F004	Bioengineering
4	F005	Humanities

Figure 9.27: Result output of the SQL query 9.27

Company/Business Database

query9.28: List names of employees who live in different cities than the warehouse locations.

To retrieve the names of employees who live in cities where no warehouse is located, we use the NOT EXISTS operator. This operator checks the subquery and returns true only if the subquery returns no rows for that specific employee's city. The query9.28 evaluates each employee's city against the list of warehouse locations. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
 First_name,
 Last_name,
 City
 FROM EMPLOYEE E
 WHERE NOT EXISTS (
   SELECT *
   FROM WAREHOUSE W
   WHERE W.Location = E.City);
```

The results of the SQL query 9.28 identify the employee residing in Prilep, as there is no corresponding warehouse location in that city, as shown in Figure 9.28.

	First_name	Last_name	City
1	Dimitar	Tocko	Prilep

Figure 9.28: Result output of the SQL query 9.28

9.5.6 Joined Tables in SQL

Join operations represent the fundamental mechanism for combining data from multiple tables based on related columns, implementing the relational algebra join operation that lies at the heart of relational database querying. Joins enable queries to traverse relationships defined by foreign keys, assembling comprehensive result sets from normalized table structures that minimize data redundancy. Understanding join types and their behavior is essential for effective database querying, as different join variants produce fundamentally different result sets depending on how they handle matching and non-matching rows.

The JOIN represents the most common join type, returning only rows where matching values exist in both tables based on the specified join condition. The syntax combines rows when the join condition evaluates to true, discarding rows from either table that lack

corresponding matches. If the names of the join columns do not match between the base tables, you can rename the columns to match.

```
SELECT columns
  FROM table1
  INNER JOIN table2 ON table1.key = table2.key;
```

The INNER JOIN (or simply JOIN) returns only matching tuples, equivalent to a Cartesian product followed by selection on the join condition. An implicit EQUIJOIN condition for each pair of attributes with the same name from table1 and table2 is created. Each such pair of attributes is included only once in the resulting relation. This way, joins enforce referential integrity, producing results that contain only validated relationships between related entities, making them ideal for queries that require complete relationship information.

University Database System

query9.29: *List all of the PROFESSOR (Professor_ID, First_name, Last_name, Hire_date, and Faculty_ID) and their FACULTY (Name, Location, Budget)*.

The relational algebra expression PROFESSOR ⋈ FACULTY uses a natural join in req8.20 to identify professors and their faculties where Faculty_ID matches. SQL Server does not have a specific NATURAL JOIN keyword. Instead, we achieve the same result using an INNER JOIN with an explicit join condition on the common attribute.

```
SELECT
    P.Professor_ID,
    P.First_name,
    P.Last_name,
    P.Hire_date,
    P.Faculty_ID,
    F.Name,
    F.Location,
    F.Budget
  FROM PROFESSOR P
  INNER JOIN FACULTY F ON P.Faculty_ID = F.Faculty_ID;
```

On the other side, MySQL supports the NATURAL JOIN keyword, which automatically joins tables based on all columns that have the same name in both tables.

```
SELECT
    Professor_ID,
    First_name,
    Last_name,
    Hire_date,
    Faculty_ID,
    Name,
    Location,
    Budget
  FROM PROFESSOR
  NATURAL JOIN FACULTY;
```

The results of both the SQL Server and MySQL query 9.29 are identical and shown in Figure 9.29.

	Professor_ID	First_name	Last_name	Hire_date	Faculty_ID	Name	Location	Budget
1	P001	Kliment	Ohridski	2007-08-01	F001	Computer Science	Main Campus	1,500,000
2	P002	Naum	Ohridski	2004-01-15	F002	Electrical Engineeri	East Wing	1,200,000
3	P003	Kiril	Prosvetitel	2001-09-25	F004	Bioengineering	North Building	1,100,000
4	P004	Metodij	Prosvetitel	2002-09-13	F001	Computer Science	Main Campus	1,500,000
5	P005	Blaze	Koneski	2012-10-19	F005	Humanities	South Building	800,000
6	P006	Richard	Johnson	2022-09-02	F001	Computer Science	Main Campus	1,500,000

Figure 9.29: Result output of the SQL query 9.29

query9.30: *List the STUDENTS (first name, last name) and the courses (title) they have been enrolled*.

The relational algebra expression, in req8.22, was using the EQUIJOIN operation that joins the STUDENT, ENROLLS_IN, and COURSE relations to get all the data in one large relation. Both SQL Server and MySQL use very similar syntax, which is achieved using multiple INNER JOIN clauses with explicit join conditions matching the primary and foreign keys.

```
SELECT
  S.First_name,
  S.Last_name,
  C.Title
  FROM STUDENT AS S
  JOIN ENROLLS_IN AS E ON S.Student_ID = E.Student_ID
  JOIN COURSE AS C ON E.Course_ID = C.Course_ID;
```

The results of the SQL query 9.30 are shown in Figure 9.30.

	First_name	Last_name	Title
1	Sarah	Smith	Database Systems
2	Bojan	Jovansen	Circuit Analysis
3	Bojan	Jovansen	Advanced Circuit Analysis
4	Kosta	Dojcinoski	Linear Algebra
5	Kosta	Dojcinoski	Matrix theory and advanced vector spaces
6	David	Boskovski	Physics
7	David	Boskovski	Advanced Physics
8	Elena	Mitan	Academic writing
9	Elena	Mitan	Advanced Academic Writing
10	Anica	Stojnasen	Intro to Programming
11	Anica	Stojnasen	Advanced Programming
12	Anica	Stojnasen	Database Systems

Figure 9.30: Result output of the SQL query 9.30

query9.31: *List all professors with faculty details on professors who were hired before 01 January 2005, and whose faculty's budget is greater than 1,000,000*.

The relational algebra expression uses the Theta-Join operation in req8.24. SQL Server and MySQL queries use an INNER JOIN to connect the PROFESSOR and FACULTY tables based on their common Faculty_ID. The WHERE clause then applies conditional filters to Hire_date and Budget, satisfying the Theta-Join criteria.

```
SELECT
    P.Professor_ID,
    P.First_name,
    P.Last_name,
    P.Hire_date,
    F.Name AS Faculty_Name,
    F.Location AS Faculty_Location,
    F.Budget AS Faculty_Budget
  FROM PROFESSOR AS P
  INNER JOIN FACULTY AS F ON P.Faculty_ID = F.Faculty_ID
  WHERE P.Hire_date < '2005-01-01' AND F.Budget > 1000000.00;
```

The relational algebra expression defined in req8.24 maps directly to the SQL statement mentioned. The results of the SQL query 9.31 are shown in Figure 9.31.

	Professor_ID	First_name	Last_name	Hire_date	Faculty_Name	Faculty_Location	Faculty_Budget
1	P004	Metodij	Prosvetitel	2002-09-13	Computer Science	Main Campus	1,500,000
2	P002	Naum	Ohridski	2004-01-15	Electrical Engineering	East Wing	1,200,000
3	P003	Kiril	Prosvetitel	2001-09-25	Bioengineering	North Building	1,100,000

Figure 9.31: Result output of the SQL query 9.31

Company/Business Database

query9.32: *List all of the employees' first name, last name, city where they work, department name, and location*.

The relational algebra expression uses a natural Join between EMPLOYEE and DEPARTMENT in req8.21 to identify the employee's First_name, Last_name, City, and their department's Dept_name and Location. Considering that SQL Server does not have a specific NATURAL JOIN keyword, it uses an INNER JOIN to return only matching tuples.

```
SELECT
    First_name,
    Last_name,
    City,
    Dept_name,
    Location
  FROM EMPLOYEE
  INNER JOIN DEPARTMENT ON EMPLOYEE.Department_ID =
DEPARTMENT.Department_ID;
```

On the other hand, MySQL supports the NATURAL JOIN keyword, which automatically joins tables on all columns with the same name in both tables.

```
SELECT
    First_name,
    Last_name,
    City,
    Dept_name,
    Location
```

```
FROM EMPLOYEE
NATURAL JOIN DEPARTMENT;
```

The results of both the SQL Server and MySQL query 9.32 are identical, see Figure 9.32.

	First_name	Last_name	City	Dept_name	Location
1	Zoran	Pejovski	Skopje	Sales	Skopje
2	Nikola	Delcevski	Skopje	Sales	Skopje
3	Dimitar	Tocko	Prilep	Sales	Skopje
4	Marija	Naumoska	Bitola	Marketing	Bitola
5	Anita	Toshevska	Bitola	Marketing	Bitola
6	Bijana	Osmani	Tetovo	Marketing	Bitola
7	Naum	Stojanoski	Ohrid	Human Resources	Ohrid
8	Hristina	Miloseska	Ohrid	Human Resources	Ohrid

Figure 9.32: Result output of the SQL query 9.32

query9.33: *List the products by warehouse location and stored in quantity on hand*.

Relational algebra expression is using the EQUIJOIN operation between the PRODUCT, STORED_IN, and WAREHOUSE relations, in req8.23. Both SQL Server and MySQL use very similar syntax, where the SQL query uses two JOIN operations to link the three tables. It first joins PRODUCT and STORED_IN on Product_ID, and then joins the result with WAREHOUSE on Warehouse_ID. The SELECT statement then retrieves the specific columns. The ORDER BY clause is added to organize the results for better readability.

```
SELECT
    P.Product_name,
    W.Location,
    SI.Quantity_on_hand
FROM PRODUCT AS P
JOIN STORED_IN AS SI ON P.Product_ID = SI.Product_ID
JOIN WAREHOUSE AS W ON SI.Warehouse_ID = W.Warehouse_ID
ORDER BY W.Location, P.Product_name;
```

The results of both the SQL Server and MySQL query 9.33 are identical and shown in Figure 9.33.

	Product_name	Location	Quantity_on_hand
1	Macedonian Ruby	Bitola	23
2	Mastika	Bitola	32
3	Tavce Gravce	Bitola	21
4	Vranec Wine	Bitola	72
5	Filigree	Ohrid	6
6	Kajmak	Ohrid	12
7	Rakija	Ohrid	18
8	Ajvar	Skopje	57
9	Ohrid Pearl	Skopje	9
10	Selsko Meso	Skopje	18
11	Sirenje	Skopje	14
12	Galichnik Wedding Costum	Tetovo	7

Figure 9.33: Result output of the SQL query 9.33

query9.34*: List all of the employees with a hire date after 01 June 2021 that are working in a department with a budget of less than 400,000*.

The relational algebra expression uses the Theta-Join operation in req8.25, between the EMPLOYEE and DEPARTMENT relations. Since both SQL Server and MySQL follow standard SQL syntax for basic joins and filtering, the query is identical for both systems. Note that in SQL Server, dates are typically handled as strings in 'YYYY-MM-DD' format, just like MySQL.

```
SELECT
    E.First_name,
    E.Last_name,
    E.Hire_date,
    D.Dept_name,
    D.Budget
FROM EMPLOYEE AS E
JOIN DEPARTMENT AS D ON E.Department_ID = D.Department_ID
WHERE E.Hire_date > '2021-06-01' AND D.Budget < 400000.00;
```

The results of both the SQL Server and MySQL query 9.34 are identical and shown in Figure 9.34. Note that employee Dimitar Tocko was hired after the date; he works in the Sales department, which has a budget of 500,000 and is therefore excluded.

	First_name	Last_name	Hire_date	Dept_name	Budget
1	Marija	Naumoska	2022-05-10	Marketing	300,000
2	Anita	Toshevska	2023-07-22	Marketing	300,000
3	Bijana	Osmani	2023-04-18	Marketing	300,000
4	Hristina	Miloseska	2021-11-12	Human Resources	200,000

Figure 9.34: Result output of the SQL query 9.34

SQL also supports OUTER JOINs: LEFT OUTER JOIN includes all left table tuples plus matches from the right (using NULL for non-matches); RIGHT OUTER JOIN includes all right table tuples plus matches; and FULL OUTER JOIN includes all tuples from both tables. These variants handle situations where not all tuples have matching counterparts, which is important for analysis involving optional relationships.

University Database System

query9.35: *List all professors and faculties, including those without a faculty affiliation.*

The relational algebra expression uses the Left Outer Join operation in req8.26, between the PROFESSOR and FACULTY to return all professors, including those without a faculty affiliation (if any), along with their faculty details.

```
SELECT
    P.Professor_ID,
    P.First_name,
    P.Last_name,
    P.Academic_title,
    F.Name
```

```
FROM PROFESSOR AS P
LEFT OUTER JOIN FACULTY AS F ON P.Faculty_ID = F.Faculty_ID;
```

The results of both the SQL Server and MySQL query 9.35 are identical and shown in Figure 9.35.

	Professor_ID	First_name	Last_name	Academic_title	Name
1	P001	Kliment	Ohridski	Full Professor	Computer Science
2	P002	Naum	Ohridski	Associate Professor	Electrical Engineeri
3	P003	Kiril	Prosvetitel	Full Professor	Bioengineering
4	P004	Metodij	Prosvetitel	Full Professor	Computer Science
5	P005	Blaze	Koneski	Assistant Professor	Humanities
6	P006	Richard	Johnson	Associate Professor	Computer Science

Figure 9.35: Result output of the SQL query 9.35

query9.36: *List all professors and faculties, including faculties without affiliated professors.*

The relational algebra expression in req8.27 uses the Right Outer Join operation between the PROFESSOR and FACULTY relations to return professors, including those faculties without any professors.

```
SELECT
    P.Professor_ID,
    P.First_name,
    P.Last_name,
    P.Academic_title,
    F.Name
FROM PROFESSOR AS P
RIGHT OUTER JOIN FACULTY AS F ON P.Faculty_ID = F.Faculty_ID;
```

The results of both the SQL Server and MySQL query 9.36 are identical and shown in Figure 9.36.

	Professor_ID	First_name	Last_name	Academic_title	Name
1	P001	Kliment	Ohridski	Full Professor	Computer Science
2	P004	Metodij	Prosvetitel	Full Professor	Computer Science
3	P006	Richard	Johnson	Associate Professor	Computer Science
4	P002	Naum	Ohridski	Associate Professor	Electrical Engineeri
5	[NULL]	[NULL]	[NULL]	[NULL]	Economy & Tourism
6	P003	Kiril	Prosvetitel	Full Professor	Bioengineering
7	P005	Blaze	Koneski	Assistant Professor	Humanities

Figure 9.36: Result output of the SQL query 9.36

query9.37: *List all professors and faculties, including those with and without affiliated professors.*

The relational algebra expression in req8.28 uses the Full Outer Join operation between the PROFESSOR and FACULTY relations to return all professors, without a faculty affiliation (if any) and those faculties without any professors.

A FULL Outer Join SQL Server Query:

```
SELECT
    P.Professor_ID,
    P.First_name,
    P.Last_name,
    P.Academic_title,
    F.Name
  FROM PROFESSOR AS P
  FULL OUTER JOIN FACULTY AS F ON P.Faculty_ID = F.Faculty_ID;
```

MySQL does not natively support FULL OUTER JOIN. It can be emulated by combining LEFT JOIN and RIGHT JOIN with UNION:

```
SELECT
    P.Professor_ID,
    P.First_name,
    P.Last_name,
    P.Academic_title,
    F.Name
  FROM PROFESSOR AS P
  LEFT OUTER JOIN FACULTY AS F ON P.Faculty_ID = F.Faculty_ID
UNION
SELECT
    P.Professor_ID,
    P.First_name,
    P.Last_name,
    P.Academic_title,
    F.Name
  FROM PROFESSOR AS P
  RIGHT OUTER JOIN FACULTY AS F ON P.Faculty_ID = F.Faculty_ID;
```

The results of SQL query 9.37 are shown in Figure 9.37

	Professor_ID	First_name	Last_name	Academic_title	Name
1	P001	Kliment	Ohridski	Full Professor	Computer Science
2	P002	Naum	Ohridski	Associate Professor	Electrical Engineering
3	P003	Kiril	Prosvetitel	Full Professor	Bioengineering
4	P004	Metodij	Prosvetitel	Full Professor	Computer Science
5	P005	Blaze	Koneski	Assistant Professor	Humanities
6	P006	Richard	Johnson	Associate Professor	Computer Science
7	[NULL]	[NULL]	[NULL]	[NULL]	Economy & Tourism

Figure 9.37: Result output of the SQL query 9.37

Company/Business Database

query9.38: *List all employees who are handling the warehouses, considering employees without a warehouse*.

The relational algebra expression in req8.29 uses the Left Outer Join operation between the EMPLOYEE and WAREHOUSE relations. This SQL Server and MySQL Query uses the LEFT OUTER JOIN clause to combine the EMPLOYEE table (aliased as E) with the

WAREHOUSE table (aliased as W). It joins the two tables on the common column, Warehouse_ID. The SELECT statement then retrieves specific columns from both tables, returning all employees, including those with and those without a warehouse.

```
SELECT
    E.Employee_ID,
    E.First_name,
    E.Last_name,
    W.Warehouse_ID,
    W.Location,
    W.Capacity
  FROM EMPLOYEE AS E
  LEFT OUTER JOIN WAREHOUSE AS W
  ON E.Warehouse_ID = W.Warehouse_ID;
```

Left Outer Join (or simply Left Join) includes all rows from the left table regardless of whether matching rows exist in the right table, using NULL values for right table columns when no match occurs. This asymmetric behavior proves valuable for finding missing relationships or including all instances of one entity regardless of the existence of related data. For instance, query9.38 returns all employees, including those not assigned to warehouses (showing NULL for warehouse location), enabling identification of unassigned employees while preserving a complete employee listing.

The results of both the SQL Server and MySQL query 9.38 are identical and shown in Figure 9.38.

	Employee_ID	First_name	Last_name	Warehouse_ID	Location	Capacity
1	E1	Zoran	Pejovski	W1	Skopje	10,000
2	E2	Nikola	Delcevski	W1	Skopje	10,000
3	E3	Marija	Naumoska	W2	Bitola	7,500
4	E4	Anita	Toshevska	W2	Bitola	7,500
5	E5	Naum	Stojanoski	W3	Ohrid	5,000
6	E6	Hristina	Miloseska	W3	Ohrid	5,000
7	E7	Dimitar	Tocko	W1	Skopje	10,000
8	E8	Bijana	Osmani	W2	Bitola	7,500

Figure 9.38: Result output of the SQL query 9.38

query9.39: *List all employees who are handling the warehouses, including warehouses without employees*.

The relational algebra expression in req8.30 uses the Right Outer Join operation between the EMPLOYEE and WAREHOUSE relations. This SQL Server and MySQL Query uses the RIGHT OUTER JOIN clause to combine the EMPLOYEE table (aliased as E) with the WAREHOUSE table (aliased as W). It joins the two tables on the common column, Warehouse_ID. The SELECT statement then retrieves specific columns from both tables, providing warehouse details and the corresponding employee information where available.

Right Outer Join mirrors Left Join behavior, including all right table rows with NULLs for non-matching left table columns. While functionally equivalent to Left Join with reversed table order, Right Join can improve query readability in specific contexts.

```
SELECT
    E.Employee_ID,
    E.First_name,
    E.Last_name,
    W.Warehouse_ID,
    W.Location,
    W.Capacity
  FROM EMPLOYEE AS E
  RIGHT OUTER JOIN WAREHOUSE AS W ON E.Warehouse_ID = W.Warehouse_ID;
```

The results of both the SQL Server and MySQL query 9.39 are identical and shown in Figure 9.39.

	Employee_ID	First_name	Last_name	Warehouse_ID	Location	Capacity
1	E1	Zoran	Pejovski	W1	Skopje	10,000
2	E2	Nikola	Delcevski	W1	Skopje	10,000
3	E7	Dimitar	Tocko	W1	Skopje	10,000
4	E3	Marija	Naumoska	W2	Bitola	7,500
5	E4	Anita	Toshevska	W2	Bitola	7,500
6	E8	Bijana	Osmani	W2	Bitola	7,500
7	E5	Naum	Stojanoski	W3	Ohrid	5,000
8	E6	Hristina	Miloseska	W3	Ohrid	5,000
9	[NULL]	[NULL]	[NULL]	W4	Tetovo	1,000

Figure 9.39: Result output of the SQL query 9.39

query9.40: *List all employees who are handling the warehouses, consider employees without a warehouse, and warehouses without employees*.

The relational algebra expression in req8.31 uses the Full Outer Join operation between the EMPLOYEE and WAREHOUSE relations. This SQL query uses the FULL OUTER JOIN clause to combine the EMPLOYEE table (aliased as E) with the WAREHOUSE table (aliased as W). The join condition is on the common column, Warehouse_ID. The SELECT statement then retrieves employee and warehouse details. This operation ensures that every employee and every warehouse is represented in the final result, regardless of whether a match exists. Full Outer Join combines both behaviors, including all rows from both tables with NULLs filling non-matching sides, useful for comprehensive relationship analysis, including orphaned records from either table. For example, query9.40 reveals employees without warehouses and warehouses without assigned employees simultaneously.

A FULL Outer Join SQL Server Query:

```
SELECT
    E.Employee_ID,
    E.First_name,
    E.Last_name,
    W.Warehouse_ID,
    W.Location,
    W.Capacity
  FROM EMPLOYEE AS E
  FULL OUTER JOIN WAREHOUSE AS W ON E.Warehouse_ID = W.Warehouse_ID;
```

MySQL does not natively support FULL OUTER JOIN. It can be emulated by combining LEFT JOIN and RIGHT JOIN with UNION:

```
SELECT
    E.Employee_ID,
    E.First_name,
    E.Last_name,
    W.Warehouse_ID,
    W.Location,
    W.Capacity
  FROM EMPLOYEE AS E
  LEFT JOIN WAREHOUSE AS W ON E.Warehouse_ID = W.Warehouse_ID
UNION
SELECT
    E.Employee_ID,
    E.First_name,
    E.Last_name,
    W.Warehouse_ID,
    W.Location,
    W.Capacity
  FROM EMPLOYEE AS E
  RIGHT OUTER JOIN WAREHOUSE AS W ON E.Warehouse_ID = W.Warehouse_ID;
```

The results of both the SQL Server and MySQL query 9.40 are identical and shown in Figure 9.40.

	Employee_ID	First_name	Last_name	Warehouse_ID	Location	Capacity
1	E1	Zoran	Pejovski	W1	Skopje	10,000
2	E2	Nikola	Delcevski	W1	Skopje	10,000
3	E3	Marija	Naumoska	W2	Bitola	7,500
4	E4	Anita	Toshevska	W2	Bitola	7,500
5	E5	Naum	Stojanoski	W3	Ohrid	5,000
6	E6	Hristina	Miloseska	W3	Ohrid	5,000
7	E7	Dimitar	Tocko	W1	Skopje	10,000
8	E8	Bijana	Osmani	W2	Bitola	7,500
9	[NULL]	[NULL]	[NULL]	W4	Tetovo	1,000

Figure 9.40: Result output of the SQL query 9.40

9.5.7 Aggregate and Grouping Functions in SQL

Aggregate functions perform calculations across multiple rows to produce single summary values, essential for statistical analysis and data summarization. These functions operate on sets of values rather than individual rows, collapsing multiple tuples into computed results that reveal patterns and metrics.

The COUNT function returns the number of values in an attribute or rows in a result set. COUNT() counts all rows including NULLs, while COUNT(column_name) counts only non-NULL values.

University Database System

query9.41: *Count students enrolled in courses in the ENROLLS_IN table*.

The SQL query to count students enrolled in courses in the ENROLLS_IN table uses a basic COUNT() statement that returns the total number of enrollment records, counting the same student once for each course they are enrolled in, query9.41 a).

```
SELECT
  COUNT(*) AS Total_Enrollments
  FROM ENROLLS_IN;
```

The relational algebra expression defined in req8.35a maps directly to the mentioned SQL statement. The results of the SQL query9.41 a) are shown in Figure 9.41 a).

while COUNT(DISTINCT Student_ID) counts **unique** students enrolled in courses from the ENROLLS_IN table, query9.41 b).

```
SELECT
  COUNT(DISTINCT Student_ID) AS Unique_Enrolled_Students
  FROM ENROLLS_IN;
```

The relational algebra expression defined in req8.35b maps directly to the mentioned SQL statement. The results of SQL query 9.41b) are shown in Figure 9.41 b). In both cases, SQL Server and MySQL use very similar syntax for these basic SQL statements.

a) b)

Figure 9.41: Result output of the SQL query 9.41 a) and b)

Company/Business Database

query9.42: *Count employees and find average, minimum, and maximum salary in the EMPLOYEE table*.

The SQL query to count employees (Employee_ID) and calculate the total salary (function SUM), average salary (function AVG), minimum salary (function MIN), and maximum salary (function MAX) from the EMPLOYEE table. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
  COUNT(Employee_ID) AS Number_of_employees,
  SUM(Salary) AS Total_Salary,
  AVG(Salary) AS Average_Salary,
  MIN(Salary) AS Minimal_Salary,
  MAX(Salary) AS Maximal_Salary
  FROM EMPLOYEE;
```

The relational algebra expression defined in req8.36 maps directly to the mentioned SQL statement. The results of SQL query 9.42 are shown in Figure 9.42.

	123 Number_of_employees	123 Total_Salary	123 Average_Salary	123 Minimal_Salary	123 Maximal_Salary
1	8	340,000	42,500	25,000	60,000

Figure 9.42: Result output of the SQL query 9.42

The SUM function calculates totals of numeric values. SELECT SUM(Salary) FROM EMPLOYEE computes the total payroll, while SELECT SUM(Credits) FROM COURSE totals course credits. SUM ignores NULL values.

The AVG function computes the arithmetic mean. SELECT AVG(Age) FROM STUDENT calculates the average student age, providing central tendency measures. AVG excludes NULL values.

The MIN and MAX functions identify the smallest and largest values, respectively. SELECT MIN(Budget), MAX(Budget) FROM FACULTY reveals the budget range. These work with numeric, date, and string types. Combined with GROUP BY, aggregates produce category-specific summaries and can calculate average salary and employee count per department, transforming data into insights.

University Database System

query9.43: *Count students enrolled in courses and then find the average, minimum, and maximum grade in ENROLLS_IN relation and group them by Student_ID*.

The SQL query uses a grouping operation to count the number of courses per student and then find the average, minimum, and maximum grades for each student. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
  Student_ID,
  COUNT(Course_ID) AS Courses_Enrolled,
  AVG(Grade) AS Average_Grade,
  MIN(Grade) AS Minimum_Grade,
  MAX(Grade) AS Maximum_Grade
  FROM ENROLLS_IN
  GROUP BY Student_ID;
```

The relational algebra expression defined in req8.37 maps directly to the mentioned SQL statement. The results of SQL query 9.43 are shown in Figure 9.43.

	A-Z Student_ID	123 Courses_Enrolled	123 Average_Grade	A-Z Minimum_Grade	A-Z Maximum_Grade
1	S001	1	9	9	9
2	S002	2	6	5	7
3	S003	2	6.5	5	8
4	S004	2	7	6	8
5	S005	2	8.5	10	7
6	S006	3	8	10	8

Figure 9.43: Result output of the SQL query 9.43

Company/Business Database

query9.44: *Find the number of employees working in each department and sum their salaries*.

The SQL query to find the number of employees and the sum of their salaries for each Department_ID using only the EMPLOYEE table. Both SQL Server and MySQL use very similar syntax for this basic SQL statement.

```
SELECT
  Department_ID,
  COUNT(Employee_ID) AS NumEmployees,
  SUM(Salary) AS TotalSalary
  FROM EMPLOYEE
  GROUP BY Department_ID;
```

The relational algebra expression defined in req8.38 maps directly to the mentioned SQL statement. The results of SQL query 9.44 are shown in Figure 9.44.

	A-Z Department_ID	123 as NumEmployees	123 as TotalSalary
1	D1	3	145,000
2	D2	3	120,000
3	D3	2	75,000

Figure 9.44: Result output of the SQL query 9.44

9.5.8 Views (Virtual Tables) in SQL

Views represent virtual tables defined by stored queries that dynamically generate result sets when referenced, providing abstraction layers over underlying base tables without physically storing data. Unlike conventional tables, views execute their defining SELECT statements each time queried, presenting current data reflecting modifications to source tables.

```
CREATE VIEW view_name
AS SELECT …;
```

University Database System

query9.45: *List the student's ID, first name, last name, and faculty ID for all resident students*.

Creating views uses the CREATE VIEW view_name AS SELECT statement syntax. In this case, to list all students' ID, first name, last name, and faculty ID for all resident students.

```
CREATE VIEW Resident_Students
AS SELECT
  Student_ID,
  First_name,
  Last_name,
  Faculty_ID
  FROM STUDENT
  WHERE Country='Macedonia';
```

defines a view showing current resident students. Applications query views identically to tables: SELECT * FROM Resident_Students, with the database executing the underlying SELECT logic. The results of the SQL query are shown in Figure 9.45.

	Student_ID	First_name	Last_name	Faculty_ID
1	S001	Sarah	Smith	F001
2	S003	Kosta	Dojcinoski	F004
3	S004	David	Boskovski	F002
4	S005	Elena	Mitan	F005

Figure 9.45: Result output of the SQL query 9.45

Views simplify complex queries by encapsulating joins, aggregations, and filters, providing pre-aggregated departmental statistics.

Company/Business Database

query9.46: *List the department summary with department ID, count the employees, and their average salary*.

Creating a view uses a Select statement to list the number of employees, average salaries per department ID, as a department summary.

```
CREATE VIEW Department_Summary
AS SELECT
  Department_ID,
  COUNT(*) AS Employee_Count,
  AVG(Salary) AS Avg_Salary
  FROM EMPLOYEE
  GROUP BY Department_ID;
```

The results of the SQL query SELECT * FROM Department_Summary are shown in Figure 9.46.

	Department_ID	Employee_Count	Avg_Salary
1	D1	3	48,333.333333
2	D2	3	40,000
3	D3	2	37,500

Figure 9.46: Result output of the SQL query 9.46

query9.47: *List the employees' public information: employee ID, first name, last name, and department ID*.

Views serve multiple purposes, including security, simplification, and consistency. Security benefits arise from exposing only authorized columns while hiding sensitive data, such as salaries:

```
CREATE VIEW Public_Employee_Info
AS SELECT
  Employee_ID,
  First_name,
  Last_name,
```

```
Department_ID
FROM EMPLOYEE;
```

The results of the SQL query SELECT * FROM Public_Employee_Info are shown in Figure 9.47.

	Employee_ID	First_name	Last_name	Department_ID
1	E1	Zoran	Pejovski	D1
2	E2	Nikola	Delcevski	D1
3	E3	Marija	Naumoska	D2
4	E4	Anita	Toshevska	D2
5	E5	Naum	Stojanoski	D3
6	E6	Hristina	Miloseska	D3
7	E7	Dimitar	Tocko	D1
8	E8	Bijana	Osmani	D2

Figure 9.47: Result output of the SQL query 9.47

9.5.9 Advanced SQL Queries

Advanced SQL queries go beyond simple selection and projection to encompass complex multi-table operations, subqueries, aggregate functions, and set operations that mirror the sequential, optimization-driven logic of relational algebra. Complex relational algebra queries often involve multiple sequential operations, making the execution order crucial for overall performance. Proper ordering and strategies for query operations are essential to ensure efficient query execution (Elmasri & Navathe, 2016). A multi-step query chain performs relational operations such as selection, projection, join, and set operations to progressively transform data until the final result is achieved (Date, 2003). For example, retrieving specific customer orders that meet certain criteria may require selecting relevant tuples, joining them with customer relations, projecting necessary attributes, and applying additional filtering or aggregation operations (Garcia-Molina et al., 2008). The sequence of these operations significantly impacts query performance, as different orders can produce varying intermediate result sizes and computational costs (Silberschatz et al., 2019). Therefore, query optimization aims to reorder and restructure relational algebra expressions to minimize execution time and resource use while maintaining logical equivalence and producing the same final results (Elmasri & Navathe, 2016). In SQL, this translates directly into how nested subqueries, join types, and aggregate functions are structured and layered within a single statement.

Understanding commutativity and associativity in SQL terms means recognizing that the order in which joins are written can affect which execution plan the optimizer selects, even though the logical result remains the same (Date, 2003). Most modern query optimizers explore multiple join orderings automatically, but providing well-filtered subqueries and accurate table statistics helps the optimizer make better decisions. Writing queries that push filters as deep as possible, project only necessary columns, and use CTEs to modularize complex logic aligns SQL practice directly with the algebraic optimization principles that underpin efficient relational query execution (Elmasri & Navathe, 2016).

University Database System

query9.48: *List students enrolled in courses taught by professors hired before 2005*.

The SQL query to find students enrolled in courses taught by professors hired before 2005 requires linking students to courses, then linking courses to professors, and finally filtering by the professor's hire date.

```
SELECT DISTINCT
    S.Student_ID,
    S.First_name,
    S.Last_name
  FROM STUDENT AS S
  INNER JOIN ENROLLS_IN AS E ON S.Student_ID = E.Student_ID
  INNER JOIN COURSE AS C ON E.Course_ID = C.Course_ID
  INNER JOIN PROFESSOR AS P ON C.Professor_ID = P.Professor_ID
  WHERE P.Hire_date < '2005-01-01';
```

The relational algebra expression defined in req8.42 maps directly to the mentioned SQL statement. The results of SQL query 9.48 are shown in Figure 9.48.

	Student_ID	First_name	Last_name
1	S002	Bojan	Jovansen
2	S003	Kosta	Dojcinoski
3	S004	David	Boskovski

Figure 9.48: Result output of the SQL query 9.48

query9.49: *List all people (students and professors) with their names and email addresses*.

The SQL query uses the UNION operator to find students and professors names and email addresses.

```
(SELECT s.First_name, s.Last_name, e.Email
  FROM STUDENT AS s
  INNER JOIN EMAIL AS e ON s.Student_ID=e.Student_ID)
UNION
(SELECT First_name, Last_name, Email
  FROM PROFESSOR)
  ORDER BY Last_name, First_name;
```

The relational algebra expression defined in req8.43 maps directly to the mentioned SQL statement. The results of SQL query 9.49 have been ordered *alphabetically* by last and first name, and are shown in Figure 9.49.

	First_name	Last_name	Email
1	David	Boskovski	david.boskovski@univeristy.edu
2	Kosta	Dojcinoski	kosta.dojcinoski@univeristy.edu
3	Richard	Johnson	richard.johnson@univeristy.edu
4	Bojan	Jovansen	bojan.j@gmail.com
5	Bojan	Jovansen	bojan.jovansen@univeristy.edu
6	Blaze	Koneski	blaze.koneski@univeristy.edu
7	Elena	Mitan	elena.mitan@gmail.com
8	Elena	Mitan	elena.mitan@univeristy.edu
9	Kliment	Ohridski	kliment.ohridski@univeristy.edu
10	Naum	Ohridski	naum.ohridski@univeristy.2edu
11	Kiril	Prosvetitel	kiril.prosvetitel@univeristy.edu
12	Metodij	Prosvetitel	metodij.prosvetitel@univeristy.edu
13	Sarah	Smith	sarah.smith@gmail.com
14	Sarah	Smith	sarah.smith@univeristy.edu
15	Anica	Stojnasen	anica.s@gmail.com
16	Anica	Stojnasen	anica.stojansen@univeristy.edu

Figure 9.49: Result output of the SQL query 9.49

Company/Business Database

query9.50: *List all employees who are managing the department names*.

The SQL query joins the EMPLOYEE and DEPARTMENT tables, based on Manages_Dept_ID attribute in the EMPLOYEE table that matches the Department_ID in the DEPARTMENT table. This operation will list the employees who are department managers and the departments they manage.

```
SELECT
    E.First_name,
    E.Last_name,
    D.Dept_name
FROM EMPLOYEE AS E
INNER JOIN DEPARTMENT AS D
ON E.Manages_Dept_ID = D.Department_ID;
```

The relational algebra expression defined in req8.44 maps directly to the mentioned SQL statement. The results of SQL query 9.50 are shown in Figure 9.50.

	First_name	Last_name	Dept_name
1	Zoran	Pejovski	Sales
2	Marija	Naumoska	Marketing
3	Naum	Stojanoski	Human Resources

Figure 9.50: Result output of the SQL query 9.50

query9.51: *Find all of the employees' first name, last name, and city who work in the same city as the department location*.

The SQL query to find employees working in the same city as their department's location requires joining the EMPLOYEE and DEPARTMENT tables and filtering the results where the City in the EMPLOYEE table matches the Location in the DEPARTMENT table.

```
SELECT
  E.First_name,
  E.Last_name,
  E.City
FROM EMPLOYEE AS E
JOIN DEPARTMENT AS D
ON E.Department_ID = D.Department_ID
WHERE E.City = D.Location;
```

The relational algebra expression defined in req8.45 maps directly to the mentioned SQL statement. The results of SQL query 9.51 are shown in Figure 9.51.

	First_name	Last_name	City
1	Zoran	Pejovski	Skopje
2	Nikola	Delcevski	Skopje
3	Marija	Naumoska	Bitola
4	Anita	Toshevska	Bitola
5	Naum	Stojanoski	Ohrid
6	Hristina	Miloseska	Ohrid

Figure 9.51: Result output of the SQL query 9.51

query9.52: *Count employees who have a salary greater than 40000*.

The SQL query to count the number of employees with a salary greater than 40,000, you need to apply a selection operation on the EMPLOYEE relation based on the salary condition and then count the resulting tuples.

```
SELECT
  COUNT(*) AS NumberOfEmployees
FROM EMPLOYEE
WHERE Salary > 40000;
```

The relational algebra expression defined in req8.46 maps directly to the mentioned SQL statement. The results of SQL query 9.52 are shown in Figure 9.52.

	NumberOfEmployees
1	4

Figure 9.52: Result output of the SQL query 9.52

query9.53: *List the ID, first name, and last name of all employees who work in a warehouse located in 'Skopje'*.

```
SELECT
    E.Employee_ID,
    E.First_name,
    E.Last_name,
```

```
    W.Location
FROM (
        SELECT Warehouse_ID, Location
        FROM WAREHOUSE
        WHERE Location = 'Skopje') AS W
INNER JOIN EMPLOYEE AS E
ON E.Warehouse_ID = W.Warehouse_ID;
```

The relational algebra expression defined in req8.47 maps directly to the mentioned SQL statement. The results of SQL query 9.53 are shown in Figure 9.53.

	Employee_ID	First_name	Last_name	Location
1	E1	Zoran	Pejovski	Skopje
2	E2	Nikola	Delcevski	Skopje
3	E7	Dimitar	Tocko	Skopje

Figure 9.53: Result output of the SQL query 9.53

Alternative Approaches - SQL Implementation

Multi-step queries often enable **alternative** approaches to solving the same problem, and understanding these alternatives is central to writing SQL that is both logically correct and computationally efficient. This decomposition facilitates inspection of intermediate results during development, enables query reuse through subquery composition, and allows database optimizers to compare equivalent algebraic expressions, selecting the most efficient execution strategy based on data characteristics and available resources while maintaining logical correctness. The sequence of SQL operations significantly impacts query performance, as different approaches can produce varying intermediate result sizes and computational costs (Silberschatz et al., 2019).

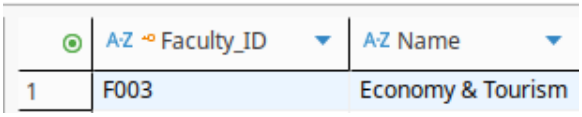
University Database System

query9.54 (**query9.23**): *Find faculties that have no affiliated professors*.

The query9.23 was implemented using the EXCEPT syntax previously. The alternative SQL query9.54 to find the faculties that have no affiliated professors using a Left Outer Join, on the FACULTY and PROFESSOR relations, and then filter for records where the professor's side of the join is NULL.

```
SELECT
    F.Faculty_ID,
    F.Name
FROM FACULTY AS F
LEFT OUTER JOIN PROFESSOR AS P ON F.Faculty_ID = P.Faculty_ID
WHERE P.Professor_ID IS NULL;
```

The results of both the SQL Server and MySQL query 9.54 are identical and shown in Figure 9.54.



	Faculty_ID	Name
1	F003	Economy & Tourism

Figure 9.54: Result output of the SQL query 9.54

Company/Business Database

query9.55 (**query9.24**): *List all of the cities where there are customers but not employees*.

The query9.24 was implemented using the EXCEPT syntax previously. The alternative SQL query9.55 to find all cities where a customer resides, but ensure there is no corresponding employee living in that same city, uses a Left Outer Join and then selects only the rows where the Employee_ID is NULL.

```
SELECT
    DISTINCT C.City
FROM CUSTOMER AS C
LEFT JOIN EMPLOYEE AS E ON C.City = E.City
WHERE E.Employee_ID IS NULL;
```

The results of both the SQL Server and MySQL query 9.55 are identical and shown in Figure 9.55, use the LEFT JOIN relational algebra expression defined in req8.48.

	City
1	Kumanovo
2	Veles

Figure 9.55: Result output of the SQL query 9.55

query9.56(**req8.49a**): *Based on the orders, calculate the total cost for each customer per product*.

The SQL query is using ORDER and ORDER_LINE relations to perform a join to link customers to their purchased products and then apply aggregation and sorting. Both SQL Server and MySQL use very similar syntax for this basic SQL statement. Exception for ORDER is a reserved keyword in MySQL, is enclosed in backticks `ORDER`, while in SQL SERVER, we use square brackets [ORDER]).

```
SELECT
   O.Customer_ID,
   OL.Product_ID,
   SUM(OL.Unit_price * OL.Quantity) AS TotalCost
FROM [ORDER] AS O
INNER JOIN ORDER_LINE AS OL
ON O.Order_ID = OL.Order_ID
GROUP BY O.Customer_ID, OL.Product_ID
ORDER BY TotalCost DESC;
```

The relational algebra expression defined in req8.49a maps directly to the mentioned SQL statement. The results of SQL query 9.56 are shown in Figure 9.56.

	Customer_ID	Product_ID	TotalCost
1	C4	10	148,800
2	C6	7	55,800
3	C7	8	31,000
4	C1	9	23,250
5	C3	4	2,790
6	C2	11	2,232
7	C4	5	1,860
8	C3	12	1,736
9	C1	1	744
10	C2	3	682
11	C5	6	620

Figure 9.56: Result output of the SQL query 9.56

query9.57(**req8.49b**): *Based on the orders, calculate the total cost for each product per customer*.

The SQL query calculates the total cost using the ORDER and ORDER_LINE relations to perform a join to link customers to their purchased products, and then applies aggregation and sorting. Both SQL Server and MySQL use very similar syntax for this basic SQL statement. Exception for ORDER in MySQL, uses backticks ``, SQL SERVER, we uses [].

```
SELECT
   Product_ID,
  Customer_ID,
   SUM(Unit_price * Quantity) AS TotalCost
FROM [ORDER] AS O
INNER JOIN ORDER_LINE AS OL
ON O.Order_ID = OL.Order_ID
GROUP BY Customer_ID, Product_ID
ORDER BY TotalCost ASC;
```

The relational algebra expression defined in req8.49b maps directly to the mentioned SQL statement. The results of SQL query 9.57 are shown in Figure 9.57

	Product_ID	Customer_ID	TotalCost
1	6	C5	620
2	3	C2	682
3	1	C1	744
4	12	C3	1,736
5	5	C4	1,860
6	11	C2	2,232
7	4	C3	2,790
8	9	C1	23,250
9	8	C7	31,000
10	7	C6	55,800
11	10	C4	148,800

Figure 9.57: Result output of the SQL query 9.57

query9.58: *List faculties with no current course enrollments*.

The SQL query to identify faculties that currently have no students enrolled in any of their offered courses, we compare the set of all faculties against the set of faculties that have at least one enrollment record.

SQL Server uses the EXCEPT operator to perform the set difference operation, following the relational algebra expression defined in req8.50, which maps directly to the mentioned SQL statement.

```
SELECT Faculty_ID, Name
 FROM FACULTY
 WHERE Faculty_ID IN (
    SELECT Faculty_ID
     FROM FACULTY
    EXCEPT
    SELECT C.Faculty_ID
     FROM COURSE AS C
    JOIN ENROLLS_IN AS E
    ON C.Course_ID = E.Course_ID);
```

MySQL does not support the EXCEPT operator to perform the set difference operation; it uses a WHERE clause to filter out the unwanted records.

```
SELECT
    F.Faculty_ID,
    F.Name
 FROM FACULTY AS F
 LEFT JOIN COURSE AS C ON F.Faculty_ID = C.Faculty_ID
 LEFT JOIN ENROLLS_IN AS E ON C.Course_ID = E.Course_ID
 WHERE E.Student_ID IS NULL;
```

The results of SQL query 9.58 are shown in Figure 9.58.

	Faculty_ID	Name
1	F003	Economy & Tourism

Figure 9.58: Result output of the SQL query 9.58

query9.59: *Find the most popular course (highest enrollment).*

The SQL query to find the most popular course identifies which Course_ID appears most frequently in the ENROLLS_IN table.

SQL Server Implementation:

```
SELECT
    TOP 1 WITH TIES
    C.Course_ID,
     C.Title,
```

```
    COUNT(E.Student_ID) AS Enrollment_Count
FROM COURSE AS C
INNER JOIN ENROLLS_IN AS E ON C.Course_ID = E.Course_ID
GROUP BY C.Course_ID, C.Title
ORDER BY COUNT(e.Student_ID)  DESC;
```

MySQL Implementation:

```
SELECT
    C.Course_ID,
    C.Title,
    COUNT(E.Student_ID) AS Enrollment_Count
FROM COURSE AS C
INNER JOIN ENROLLS_IN AS E ON C.Course_ID = E.Course_ID
GROUP BY C.Course_ID, C.Title
ORDER BY Enrollment_Count DESC
LIMIT 1;
```

The relational algebra expression defined in req8.51 corresponds to the mentioned SQL statement. The results of SQL query 9.59 are shown in Figure 9.59.

	Course_ID	Title	Enrollment_Count
1	CS303	Database Systems	2

Figure 9.59: Result output of the SQL query 9.59

University Database System

query9.60: *Find TEXTBOOK Titles that are given for ASSIGNMENT*.

The SQL query to find titles common to both the set of all textbook titles and the set of titles associated with assignments follows the relational algebra expression using INTERSECT defined in req8.52, and maps directly to the following SQL statement. Both SQL Server and MySQL 8.0.31+ use very similar syntax for this basic SQL statement:

```
SELECT
    Title
FROM TEXTBOOK
INTERSECT
SELECT Title
FROM ASSIGNMENT AS A
JOIN TEXTBOOK AS T ON A.ISBN = T.ISBN;
```

Alternative approach to INTERSECT operation for older MySQL versions would be:

```
SELECT
DISTINCT Title
FROM TEXTBOOK AS T
INNER JOIN ASSIGNMENT AS A ON T.ISBN = A.ISBN;
```

The results of SQL query 9.60 are shown in Figure 9.60.

	Title
1	Electric Circuits
2	Physics for Scientists and Engineers
3	Introduction to Programming by Examples
4	Academic concepts
5	Linear Algebra Done Right
6	Matrix theory and advanced vector spaces
7	Fundamentals of Database Systems
8	Object-oriented programming with examples
9	Complex circuit analysis and design
10	Quantum mechanics and advanced thermodynamics
11	Research methodology and scientific writing

Figure 9.60: Result output of the SQL query 9.60

University Database System

query9.61: *Find the titles of textbooks that are not assigned.*

The SQL query to find the titles of textbooks that are not assigned, we can perform an EXCEPT operation.

SQL Server uses the EXCEPT operator to perform the set difference operation, following the relational algebra expression defined in req8.53, which maps directly to the mentioned SQL statement.

```
SELECT Title
  FROM TEXTBOOK AS T
EXCEPT
SELECT T.Title
  FROM ASSIGNMENT AS A
INNER JOIN TEXTBOOK AS T
ON A.ISBN = T.ISBN;
```

MySQL does not support the EXCEPT operator to perform the set difference operation; it uses a WHERE clause to filter out the unwanted records.

```
SELECT T.Title
  FROM TEXTBOOK AS T
LEFT JOIN ASSIGNMENT AS A USING (ISBN)
WHERE A.ISBN IS NULL;
```

The results of SQL query 9.61 are shown in Figure 9.61.

	Title
1	Academic writing

Figure 9.61: Result output of the SQL query 9.61

9.5.10 Query Optimization and Execution

Modern database systems apply advanced query optimizers that transform SQL queries into efficient execution plans by applying Relational Algebra equivalences plus cost-based optimization strategies. When a query is submitted, the DBMS performs several phases: parsing and validation (checking syntax and semantics), optimization (generating and evaluating execution plans), and execution (performing the operations). The optimizer's goal is to minimize query execution time, which depends on factors such as disk I/O (typically the slowest operation), CPU usage, memory consumption, and network transfer (in distributed systems).

The decomposition into discrete subquery layers facilitates inspection of intermediate results during development, enables query reuse through subquery composition, and allows database optimizers to compare equivalent algebraic expressions, selecting the most efficient execution strategy based on data characteristics and available resources while maintaining logical correctness. Understanding algebraic properties such as commutativity and associativity enables query optimizers to explore alternative execution plans and select strategies that minimize disk I/O, memory usage, and computational overhead while ensuring correct and efficient query execution (Date, 2003; Elmasri & Navathe, 2016). The underlying goal remains consistent: to align the structure of the SQL statement with the logical and physical optimization principles of relational algebra, producing queries that are transparent in intent, efficient in execution, and maintainable as data volumes and schema complexity grow (Elmasri & Navathe, 2016; Garcia-Molina et al., 2008). This subsection demonstrates how SQL query optimizers apply algebraic transformations to improve performance. Understanding these concepts helps developers write queries that the optimizer can efficiently execute.

University Database System

query9.62: *Find students enrolled in the course*.

The SQL query to find students enrolled in courses demonstrates the associativity of joins — the same result can be achieved by grouping the joins differently, and SQL's query optimizer can exploit these algebraic equivalences to select the most efficient execution plan. The relational algebra expression in req8.54 maps directly to the following SQL statements. When joining STUDENT, ENROLLS_IN, and COURSE, the algebraic equivalences allow query transformation without changing semantics.

- **Left grouped join: (STUDENT⋈ENROLLS_IN)⋈ COURSE**

```
SELECT
    SE.First_name,
    SE.Last_name,
    C.Title
  FROM (SELECT
            S.First_name,
            S.Last_name,
            E.Course_ID
        FROM STUDENT AS S
```

```
        INNER JOIN ENROLLS_IN AS E ON S.Student_ID = E.Student_ID) AS SE
        INNER JOIN COURSE AS C ON SE.Course_ID = C.Course_ID;
```

- **Right grouped join: ENROLLS_IN⋈ (STUDENT⋈COURSE)**

```
SELECT
    S.First_name,
    S.Last_name,
    EC.Title
  FROM STUDENT AS S
  INNER JOIN (
        SELECT
            E.Student_ID,
            C.Title,
            C.Course_ID
        FROM ENROLLS_IN AS E
        INNER JOIN COURSE AS C ON E.Course_ID = C.Course_ID) AS EC
  ON S.Student_ID = EC.Student_ID;
```

The standard SQL statement lets the optimizer decide on the execution as long as results filtering is not required. Both SQL Server and MySQL 8.0.31+ use very similar syntax:

```
SELECT
    S.First_name,
    S.Last_name,
    C.Title
    FROM STUDENT AS S
  INNER JOIN ENROLLS_IN AS E ON S.Student_ID = E.Student_ID
  INNER JOIN COURSE AS C ON E.Course_ID = C.Course_ID;
```

The results of SQL query 9.62 are shown in Figure 9.62.

	First_name	Last_name	Title
1	Sarah	Smith	Database Systems
2	Bojan	Jovansen	Circuit Analysis
3	Bojan	Jovansen	Advanced Circuit Analysis
4	Kosta	Dojcinoski	Linear Algebra
5	Kosta	Dojcinoski	Matrix theory and advanced vector spaces
6	David	Boskovski	Physics
7	David	Boskovski	Advanced Physics
8	Elena	Mitan	Academic writing
9	Elena	Mitan	Advanced Academic Writing
10	Anica	Stojnasen	Intro to Programming
11	Anica	Stojnasen	Advanced Programming
12	Anica	Stojnasen	Database Systems

Figure 9.62: Result output of the SQL query 9.62

Company/Business Database

query9.63: req8.55 *Find all orders and product quantities*

The SQL query to list the products contained in each order joins ORDER (R), ORDER_LINE (S), and PRODUCT (T), with the algebraic equivalences allowing query transformation without changing semantics.

- **Left grouped join: (ORDER ⋈ ORDER_LINE) ⋈ PRODUCT**

```
SELECT
   OOL.Order_ID,
   P.Product_ID,
   P.Product_name,
   OOL.Quantity
  FROM (
  SELECT
        OL.Order_ID,
        OL.Product_ID,
        OL.Quantity
  FROM [ORDER] AS O
  INNER JOIN ORDER_LINE AS OL ON O.Order_ID = OL.Order_ID) as OOL
  INNER JOIN PRODUCT AS P ON OOL.Product_ID = P.Product_ID;
```

- **Right grouped join: ORDER ⋈ (ORDER_LINE ⋈ PRODUCT)**

```
SELECT
   O.Order_ID,
   P.Product_ID,
   P.Product_name,
   OL.Quantity
  FROM [ORDER] AS O
  INNER JOIN (ORDER_LINE AS OL
  INNER JOIN PRODUCT AS P ON OL.Product_ID = P.Product_ID) ON O.Order_ID =
OL.Order_ID;
```

The standard SQL statement lets the optimizer decide on the execution. Both SQL Server and MySQL 8.0.31+ use very similar syntax:

```
SELECT
   O.Order_ID,
   P.Product_ID,
   P.Product_name,
   OL.Quantity
  FROM ([ORDER] AS O
  INNER JOIN ORDER_LINE AS OL ON O.Order_ID = OL.Order_ID)
  INNER JOIN PRODUCT AS P ON OL.Product_ID = P.Product_ID;
```

Note: In MySQL use `ORDER` (backticks) instead of [ORDER].
The results of SQL query 9.63 are shown in Figure 9.63.

	Order_ID	Product_ID	Product_name	Quantity
1	1	1	Ajvar	2
2	9	11	Vranec Wine	2
3	2	3	Rakija	1
4	10	12	Selsko Meso	2
5	3	4	Mastika	3
6	11	10	Galichnik Wedding Costum	2
7	4	5	Sirenje	4
8	5	6	Kajmak	1
9	6	7	Ohrid Pearl	6
10	7	8	Macedonian Ruby	1
11	8	9	Filigree	5

Figure 9.63: Result output of the SQL query 9.63

The Commutativity of Selection principle in relational algebra states that when multiple filter conditions are applied to a relation, their order is interchangeable without affecting the final result. In SQL, this means chaining WHERE clause conditions in any sequence yields the same output. Query optimizers exploit this property to reorder predicates strategically — typically applying the most selective (restrictive) filters first — reducing intermediate result sizes and improving overall query execution performance.

The significance of join ordering in SQL stems from the fact that each JOIN operation produces an intermediate result set that serves as input to subsequent operations. Poor join ordering can result in large intermediate tables, leading to excessive memory usage, increased disk I/O, and higher CPU costs, whereas an optimal ordering can dramatically reduce intermediate result sizes through early filtering and selective joins (Date, 2003). Additionally, the order in which tables are joined affects the query optimizer's ability to effectively exploit available indexes, influencing access paths, selectivity estimates, and physical data access patterns (Elmasri & Navathe, 2016). As data volumes and query complexity increase, understanding the impact of JOIN ordering becomes essential for database designers and developers aiming to ensure scalable performance and efficient query execution in relational database management systems (Silberschatz et al., 2019).

University Database System

query9.64: *Find the names of students enrolled in "Intro to Programming" (CS101).*

The SQL query to find students enrolled in "Intro to Programming", from the university database, joins three relations: **STUDENT**, **ENROLLS_IN**, and **COURSE**, following the relational algebra expression in req8.56 maps directly to the following SQL statements:

- **Inefficient Expression.** First Join, then Select**,** this creates a large intermediate result before filtering:

```
SELECT
    S.Student_ID,
    S.First_name,
    S.Last_name,
```

```
    C.Title
FROM STUDENT AS S
INNER JOIN ENROLLS_IN AS E ON S.Student_ID = E.Student_ID
INNER JOIN COURSE AS C ON E.Course_ID = C.Course_ID
WHERE C.Title = 'Intro to Programming';
```

Retrieving students enrolled in "Intro to Programming" demonstrates that changing the order of operations can significantly affect query speed.

- **Optimized Expression.** First Select, then Join, optimized version with nested subqueries

```
SELECT
    S.Student_ID,
    S.First_name,
    S.Last_name,
    TC.Title
FROM (
SELECT C.Course_ID, C.Title
FROM COURSE AS C
WHERE C.Title = 'Intro to Programming') AS TC
INNER JOIN ENROLLS_IN AS E ON E.Course_ID = TC.Course_ID
INNER JOIN STUDENT AS S ON S.Student_ID = E.Student_ID;
```

The results of SQL query 9.64 are shown in Figure 9.64.

	Student_ID	First_name	Last_name	Title
1	S006	Anica	Stojnasen	Intro to Programming

Figure 9.64: Result output of the SQL query 9.64

Company/Business Database

query9.65: req8.57 *Find the names and quantities of all products ordered by the customer 'Ivanovska'.*

The SQL query to find all products, by names and quantities, from a specific customer requires multiple relations from the Company system relational model to be combined, including CUSTOMER, ORDER, ORDER_LINE, and PRODUCT.

- **Inefficient Expression (first Join, then Select):**

In the **inefficient approach**, the database system first joins all four relations—CUSTOMER, ORDER, ORDER_LINE, and PRODUCT—to construct an intermediate relation that contains every customer, all their orders, the associated order lines, and the corresponding product information.

```
SELECT
    P.Product_name,
    OL.Quantity
FROM CUSTOMER AS C
JOIN [ORDER] AS O ON C.Customer_ID = O.Customer_ID
```

```
JOIN ORDER_LINE AS OL ON O.Order_ID = OL.Order_ID
JOIN PRODUCT AS P ON OL.Product_ID = P.Product_ID
WHERE C.Last_name = 'Ivanovska';
```

- **Optimized Expression (first Select, then Join):**

In contrast, the **optimized approach** applies the selection operation as early as possible:

```
SELECT
    P.Product_name,
    OL.Quantity
FROM (
SELECT *
FROM CUSTOMER
WHERE Last_name = 'Ivanovska') AS C
JOIN [ORDER] AS O ON C.Customer_ID = O.Customer_ID
JOIN ORDER_LINE AS OL ON O.Order_ID = OL.Order_ID
JOIN PRODUCT AS P ON OL.Product_ID = P.Product_ID;
```

Note: In MySQL use `ORDER` (backticks) instead of [ORDER].

By filtering the **CUSTOMER** relation first, the system reduces it to a single tuple corresponding to the customer Ivanovska.

The results of SQL query 9.65 are shown in Figure 9.65.

	A-Z Product_name	123 Quantity
1	Ajvar	2
2	Filigree	5

Figure 9.65: Result output of the SQL query 9.65

Performance Optimization Principles

Selection and projection pushdowns are one of the most fundamental optimization techniques in SQL query processing. Its core objective is to reduce the volume of data flowing through a query execution plan by filtering rows and eliminating unnecessary columns as early as possible in the execution sequence (Silberschatz et al., 2019). Rather than allowing large intermediate result sets to propagate through costly operations such as JOIN, aggregations, and sorting before applying restrictions, modern query optimizers push WHERE and column selection operations closer to the data source, converting a data-intensive execution strategy into an efficient pipeline that processes only relevant data from the outset (Date, 2003).

The benefits of pushdown optimization are cumulative: reducing intermediate result sizes lowers memory usage, minimizes disk I/O, and decreases CPU overhead in subsequent operations (Garcia-Molina et al., 2008). For example, applying a highly selective WHERE predicate to a large table before a JOIN can reduce the number of processed rows by orders of magnitude (Elmasri & Navathe, 2016). Early column pruning further contributes by narrowing row width, improving cache utilization, and data transfer efficiency (Date, 2019). Pushdown strategies also enable the optimizer to better leverage indexes and data

partitions, often replacing full-table scans with efficient indexed access paths (Silberschatz et al., 2019).

In practice, selection pushdown positions filter predicates as close as possible to the initial data access point. When a WHERE condition is pushed to the storage layer, the optimizer can substitute costly full-table scans with index seeks, fundamentally reducing what enters the processing pipeline and directly impacting overall query performance (Connolly & Begg, 2015).

University Database System

query9.66: *Find all professors from the Computer Science faculty, hired before 2010.*

The SQL query to find professors' faculty information and hire date must combine information from the **PROFESSOR** and **FACULTY** relations. Although the logical conditions of the query are simple, the efficiency of its execution depends heavily on where the selection predicates are applied within the query plan. The relational algebra expression defined in req8.58 maps the *inefficient* and *optimized* plans. Both SQL Server and MySQL use very similar syntax for this SQL query statement.

- **Inefficient Plan (Filter after Join):**

In the **inefficient execution plan**, the database system first performs a join between **PROFESSOR** and **FACULTY**, producing an intermediate relation that includes all professors along with their associated faculty information.

```
SELECT
    P.Professor_ID,
    P.First_name,
    P.Last_name,
    P.Hire_date,
    F.Name AS Faculty_Name
FROM PROFESSOR AS P
JOIN FACULTY AS F ON P.Faculty_ID = F.Faculty_ID
WHERE P.Faculty_ID = 'F001' AND P.Hire_date < '2010-01-01';
```

- **Optimized Plan (Selection Pushdown):**

In the **optimized execution plan**, the selection predicates are pushed down and applied directly to the **PROFESSOR** relation before any join is performed.

```
SELECT
    P.Professor_ID,
    P.First_name,
    P.Last_name,
    P.Hire_date,
    F.Name AS Faculty_Name
FROM (
SELECT Professor_ID, First_name, Last_name, Hire_date, Faculty_ID
FROM PROFESSOR
WHERE Faculty_ID = 'F001' AND Hire_date < '2010-01-01') AS P
```

```
INNER JOIN FACULTY AS F ON P.Faculty_ID = F.Faculty_ID;
```

The results of SQL query 9.66 are shown in Figure 9.66.

	Professor_ID	First_name	Last_name	Hire_date	Faculty_Name
1	P001	Kliment	Ohridski	2007-08-01	Computer Science
2	P004	Metodij	Prosvetitel	2002-09-13	Computer Science

Figure 9.66: Result output of the SQL query 9.66

Company/Business Database

query9.67: *Find employees who are handling 'Pending' orders*

The SQL query to find pending orders by employees combines information from the **EMPLOYEE** and **ORDER** relations. The **EMPLOYEE** relation stores employee details, while the **ORDER** relation records customer orders and their current processing status. The relational algebra expression defined in req8.59 maps the *inefficient* and *optimized* plans. Both SQL Server and MySQL use very similar syntax for this SQL query statement.

- **Inefficient Plan (Filter after Join):**

In the **inefficient execution plan**, the database system first performs a join between **EMPLOYEE** and **ORDER**, producing an intermediate relation that includes every employee paired with their associated orders, regardless of order status. Only after this potentially large intermediate result is generated is the selection condition applied to retain orders with *Status = 'Pending'*:

```
SELECT
  E.Employee_ID,
  E.First_name,
  E.Last_name,
  O.Order_ID,
  O.Status
FROM EMPLOYEE AS E
JOIN [ORDER] AS O
ON E.Employee_ID = O.Employee_ID
WHERE O.Status = 'Pending';
```

- **Optimized Plan (Selection Pushdown):**

In contrast, the **optimized execution plan** applies the selection operation directly to the **ORDER** relation before performing the join:

```
SELECT
  E.Employee_ID,
  E.First_name,
  E.Last_name,
  PO.Order_ID,
  PO.Status
FROM EMPLOYEE AS E
JOIN (
```

```
SELECT Order_ID, Status, Employee_ID
FROM [ORDER]
WHERE Status = 'Pending') AS PO
ON E.Employee_ID = PO.Employee_ID;
```

Note: In MySQL use `ORDER` (backticks) instead of [ORDER].

The results of SQL query 9.67 are shown in Figure 9.67.

	Employee_ID	First_name	Last_name	Order_ID	Status
1	E3	Marija	Naumoska	3	Pending
2	E1	Zoran	Pejovski	9	Pending

Figure 9.67: Result output of the SQL query 9.67

Projection pushdown extends SQL query optimization beyond row filtering to column elimination, removing unnecessary columns as early as possible in the execution plan. By applying column pruning before resource-intensive operations such as JOIN and sorting, the database reduces the width of rows flowing through the execution pipeline (Silberschatz et al., 2019). Narrower rows consume less memory in temporary buffers, reduce data volume in distributed architectures, and accelerate operations sensitive to record width — particularly sorting and hash-based join algorithms (Garcia-Molina et al., 2008). This technique proves especially valuable for analytical queries accessing wide tables, where selecting only relevant columns significantly reduces I/O and memory overhead (Date, 2003). Projection pushdown the required columns (and the necessary join keys, like IDs) onto the individual tables *before* the join, thus narrowing the relations that are being joined.

University Database System

query9.68: *Find the names and grades of all students in the course CS303.*

The SQL query to find student names and grades in a specific course must combine information from the **STUDENT** and **ENROLLS_IN** relations. The relational algebra expression defined in req8.60 maps the *inefficient* and *optimized* plans. Both SQL Server and MySQL use very similar syntax for this SQL query statement.

- **Inefficient Plan (Projection late):**

In the **inefficient execution plan**, the database system first performs a join between **STUDENT** and **ENROLLS_IN**, producing a large intermediate relation that includes all student attributes along with enrollment data.

```
SELECT
    S.First_name,
    S.Last_name,
    E.Grade
FROM STUDENT AS S
INNER JOIN ENROLLS_IN AS E ON S.Student_ID = E.Student_ID
WHERE E.Course_ID = 'CS303';
```

This approach is inefficient because large and irrelevant columns—such as *Street*, *City*, and *Birth_date*—are unnecessarily propagated through the join, increasing memory usage, disk I/O, and processing time.

- **Optimized Plan (Projection Pushdown):**

In the **optimized execution plan**, **projection pushdown** is applied to remove unneeded attributes as early as possible. Instead of joining the full tables and then projecting, we project the specific columns needed from each table *before* the join occurs.

```
SELECT
    S.First_name,
    S.Last_name,
    E.Grade
FROM STUDENT AS S
INNER JOIN (
        SELECT Student_ID, Grade
        FROM ENROLLS_IN
        WHERE Course_ID = 'CS303') AS E
ON S.Student_ID = E.Student_ID;
```

The results of SQL query 9.68 are shown in Figure 9.68.

	First_name	Last_name	Grade
1	Sarah	Smith	9
2	Anica	Stojnasen	6

Figure 9.68: Result output of the SQL query 9.68

Company/Business Database

query9.69: *List the products by warehouse only in case the Quantity_on_hand is bigger than three times the Reorder_level in the STORED_IN relation.*

The SQL query to find products in stock that significantly exceed their reorder levels uses Projection Pushdown to reduce the size of the data stream before joining the product descriptions with the inventory data. The relational algebra expression defined in req8.61 maps the *inefficient* and *optimized* plans. Both SQL Server and MySQL use very similar syntax for this SQL query statement.

- **Inefficient Plan (Projection late):**

In this plan, the database engine joins the entire STORED_IN table with the entire PRODUCT table (carrying over columns like Price, Weight, Description, and Reorder_level) before finally filtering the rows and selecting the relevant columns.

```
SELECT
    P.Product_name,
    S.Warehouse_ID,
    S.Quantity_on_hand,
    S.Reorder_level
FROM STORED_IN AS S
```

```
INNER JOIN PRODUCT AS P
ON S.Product_ID = P.Product_ID
WHERE S.Quantity_on_hand > (3 * S.Reorder_level);
```

- **Optimized Plan (Projection Pushdown):**

This plan "pushes" the selection and projection operations down to the individual tables. It filters inventory first and keeps only the necessary columns (Product_ID and Product_name from one side; Product_ID, Warehouse_ID, Quantity_on_hand, and Reorder_level from the other) before the join occurs.

```
SELECT
   P.Product_name,
   S.Warehouse_ID,
   S.Quantity_on_hand,
   S.Reorder_level
FROM (
        SELECT
        Product_ID, Warehouse_ID, Quantity_on_hand, Reorder_level
        FROM STORED_IN
        WHERE Quantity_on_hand > (3 * Reorder_level)) AS S
INNER JOIN (
        SELECT
        Product_ID, Product_name
        FROM PRODUCT) AS P
ON S.Product_ID = P.Product_ID;
```

Performance Note: By narrowing the relations before the join, the system avoids loading the Description, Category, Price, and Weight columns from the PRODUCT table into memory for every single comparison, significantly reducing I/O and memory overhead. The results of SQL query 9.69 are shown in Figure 9.69.

	Product_name	Warehouse_ID	Quantity_on_hand	Reorder_level
1	Ajvar	W1	57	10
2	Tavce Gravce	W2	21	6
3	Rakija	W3	18	4
4	Filigree	W3	6	1
5	Vranec Wine	W2	72	10

Figure 9.69: Result output of the SQL query 9.69

Operation Ordering and Impact on Performance

Understanding how relational algebra operations translate to SQL and how query optimizers apply transformation rules is essential for writing efficient database queries. The examples in this section demonstrate that while SQL provides multiple ways to express the same logical query, the choice of implementation significantly impacts performance. By applying selection and projection pushdown principles and choosing appropriate join strategies, developers can significantly reduce execution time and resource consumption.

Key Optimization Techniques:

1. Selection Pushdown

- Apply WHERE filters as early as possible to minimize intermediate result sizes.
- Use CTEs or derived tables to materialize filtered results before joins.
- Leverage indexes on filter columns for efficient data access.
- Consider EXISTS over IN for large datasets (enables semi-join optimization).

2. Projection Pushdown

- Select only necessary columns to reduce data volume in memory and I/O.
- Project columns before joins to narrow tuple width.
- Particularly important for tables with many wide columns or BLOBs.
- Reduces network traffic in distributed or client-server architectures.

3. Join Ordering

- Let the optimizer choose join order in most cases.
- Start with the smallest or most filtered table when forcing order.
- Use FORCE ORDER (SQL Server) or STRAIGHT_JOIN (MySQL) only when the optimizer consistently chooses a suboptimal plan and manual intervention is justified by profiling evidence.

Cost-based optimization examines multiple execution plans using statistical information about tables (cardinality, value distributions, clustering) and available access paths (indexes, sort orders). The optimizer estimates the cost of each plan based on I/O operations, CPU cycles, and memory requirements, and selects the plan with the minimum estimated cost. Access path selection determines whether to use a full table scan (reading all rows sequentially) or an index scan (using a B-tree or hash index to locate specific rows). Indexes dramatically improve query performance for selective queries but impose overhead for insertions and updates. Join method selection chooses among nested loop joins (iterating outer table and probing inner table for each row, efficient when inner table has an index), hash joins (building a hash table on one input and probing with the other, efficient for large inputs with no useful indexes), and merge joins (sorting both inputs and merging, efficient when inputs are already sorted or when sort is inexpensive). Learning these optimized strategies helps database developers write efficient queries and design appropriate indexes.

9.6 Data Security: DCL Commands

Data Control Language (DCL) comprises SQL commands that manage access permissions and security in database systems, acting as an essential element of database administration and governance. DCL ensures that users can perform only operations for which they have been explicitly authorized, implementing the principle of least privilege, which requires users to have only the permissions necessary for their roles. The two primary DCL commands are GRANT, which provides permissions, and REVOKE, which removes them. Permissions operate at multiple granularities: database level (connect to a database,

create objects), schema level (access all objects in a schema), object level (table, view, procedure), and column level (access specific columns within tables). The GRANT syntax follows the pattern

```
GRANT {privilege_list | ALL PRIVILEGES}
ON Object_Name
TO {principal | PUBLIC}
```

where privilege_list includes SELECT, INSERT, UPDATE, DELETE for data operations, and CREATE, ALTER, DROP for structural changes, object specifies the database object (table, view, procedure), and principal identifies the user or role receiving permissions. For example,

```
GRANT SELECT, INSERT
ON STUDENT
TO UniversityStaff
```

allows the UniversityStaff role to query and add student records. The WITH GRANT OPTION clause enables grantees to grant permissions to others further, creating permission hierarchies but requiring attentive control to prevent uncontrolled privilege escalation. Role-based access control (RBAC) simplifies permission management by grouping related permissions into roles, then assigning roles to users:

```
CREATE ROLE DataAnalyst;

SQL Server:
GRANT SELECT ON SCHEMA::dbo TO DataAnalyst;
ALTER ROLE DataAnalyst ADD MEMBER JaneSmith;

MySQL:
GRANT SELECT ON database_name.* TO DataAnalyst;
GRANT DataAnalyst TO JaneSmith;
```

Both approaches assign Jane all analyst permissions through role membership, though the syntax differs between SQL Server and MySQL. REVOKE removes permissions using REVOKE permission_list ON object FROM principal, with an optional CASCADE clause to remove permissions from all users who received them transitively through WITH GRANT OPTION.

```
REVOKE [GRANT OPTION FOR] {privilege_list | ALL PRIVILEGES}
ON Object_Name
FROM {principal | PUBLIC} [RESTRICT|CASCADE]
```

> **Definition: Data Control Language (DCL)**
>
> DCL is a subset of SQL commands used to manage user permissions and access rights to database objects, primarily including GRANT and REVOKE statements for security control.

SQL Server extends DCL with additional security features, including schemas as security boundaries, transparent data encryption for data at rest, Always Encrypted for end-to-end

encryption, dynamic data masking to obfuscate sensitive data, and row-level security implementing fine-grained access control through security predicates. Column-level permissions restrict access to sensitive attributes:

```
GRANT SELECT
ON EMPLOYEE (Employee_ID, First_name, Last_name)
TO HRAssistant;

REVOKE SELECT
ON EMPLOYEE(Salary)
FROM HRAssistant;
```

This ensures HR assistants see employee names but not salaries.

Views serve as security mechanisms by exposing filtered or aggregated data:

```
CREATE VIEW PublicStudentInfo
AS SELECT
Student_ID, First_name, Last_name, City
FROM STUDENT;

GRANT SELECT
ON PublicStudentInfo
TO Public;
```

This provides access to non-sensitive student data without exposing addresses or personal details. Auditing complements DCL by logging access attempts, capturing who accessed what data when, and enabling forensic analysis of security breaches and regulatory compliance, such as GDPR, HIPAA, or SOX. Modern databases support audit specifications capturing login attempts, permission changes, data access, and DDL modifications, with audit logs stored securely outside the database to prevent tampering.

9.7 Differences between SQL Server and MySQL

Although SQL standards establish a common foundation for relational database management, the two most widely used database systems, Microsoft SQL Server and MySQL, differ substantially in architecture, feature set, licensing model, and SQL dialect. Understanding these differences is essential for developers writing portable code, database administrators migrating or supporting heterogeneous environments, and organizations selecting the right technology for their requirements. This section examines each system individually and then provides a structured side-by-side comparison of the most significant implementation differences.

9.7.1 SQL Server vs. MySQL: General Differences

Microsoft SQL Server is a comprehensive enterprise database platform that has evolved over three decades from its origins as a port of Sybase SQL Server into one of the most widely deployed commercial database systems. SQL Server closely follows ANSI SQL standards as it adds extensive Microsoft-specific features through Transact-SQL (T-SQL), a procedural extension that includes control-of-flow constructs (IF...ELSE, WHILE loops,

TRY...CATCH error handling), variable declarations and assignments, stored procedures, triggers, and user-defined functions. SQL Server performs well in Windows-centric environments, offering tight integration with Microsoft's ecosystem including the .NET Framework for stored procedure development in C# or VB.NET, Azure cloud services for hybrid and cloud-native deployments, Active Directory for authentication and authorization, and business intelligence tools like SQL Server Reporting Services (SSRS), SQL Server Integration Services (SSIS), and SQL Server Analysis Services (SSAS).

SQL Server's enterprise feature set covers AlwaysOn Availability Groups for high availability, Transparent Data Encryption (TDE) for data at rest, Always Encrypted for end-to-end column-level encryption, dynamic data masking, row-level security, and five transaction isolation levels (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE, SNAPSHOT). Comprehensive auditing capabilities support regulatory frameworks such as HIPAA, SOX, and GDPR. Licensing is primarily commercial on a per-core or per-user basis; free editions include SQL Server Express (10 GB limit) for small applications and the Developer edition (full Enterprise features, non-production use only). Cloud deployment is available through Azure SQL Database (PaaS) and Azure SQL Managed Instance, both supporting pay-as-you-go pricing.

MySQL, originally developed by MySQL AB (a Swedish company founded by Michael 'Monty' Widenius, David Axmark, and Allan Larsson) and now owned by Oracle Corporation, represents the world's most popular open-source relational Database. MySQL achieved dominance through its role in the LAMP stack (Linux, Apache, MySQL, PHP/Python/Perl), becoming the default database for web applications, content management systems, and Internet-scale services.

MySQL's architecture emphasizes simplicity, performance, and reliability for read-intensive workloads. A defining characteristic is its pluggable storage engine architecture: InnoDB (default since MySQL 5.5) provides full ACID compliance, foreign key enforcement, crash recovery, and MVCC-based concurrency; MyISAM offers fast reads but lacks transactions and foreign keys (deprecated for general use); Memory (HEAP) stores data entirely in RAM for fast temporary access; and Archive provides compressed storage for historical, rarely-queried data. MySQL 8.0 significantly advanced SQL standards compliance by adding CTEs, window functions, atomic DDL operations, persistent auto-increment counters, and full Unicode support (utf8mb4 as default). The Community Edition is freely available under the GNU GPL, with an active ecosystem of forks (MariaDB, Percona Server), cloud-managed services (Amazon RDS, Google Cloud SQL), and third-party tooling. Oracle’s MySQL Enterprise Edition offers commercial extensions including advanced backup, monitoring, and security features with formal support contracts.

9.7.2 SQL Server vs. MySQL: Implementation Differences

Although SQL Server and MySQL share the same relational foundations and ANSI SQL core, their implementations diverge in numerous practical ways that affect day-to-day development and database administration. The most significant areas of divergence are summarized below. Data types differ substantially: SQL Server uses VARCHAR(MAX) for large text where MySQL uses TEXT; SQL Server provides DATETIME2 with nanosecond precision compared to MySQL’s DATETIME; SQL Server uses BIT for Boolean values while

MySQL uses TINYINT(1) or the BOOLEAN alias; and SQL Server provides a dedicated uniqueidentifier type for GUIDs while MySQL typically stores them as VARCHAR(36) or BINARY(16). String concatenation is performed with the + operator in SQL Server (e.g., First_name + ' ' + Last_name), whereas MySQL uses the CONCAT() function. Date and time functions differ: SQL Server uses GETDATE(), DATEDIFF(), and DATEADD(), while MySQL provides NOW(), TIMESTAMPDIFF(), and DATE_ADD() with INTERVAL syntax. Auto-increment columns are defined with IDENTITY(seed, increment) in SQL Server and with AUTO_INCREMENT in MySQL. Row limiting uses SELECT TOP n or OFFSET...FETCH in SQL Server, versus LIMIT and OFFSET in MySQL. Case sensitivity is controlled by collation in both systems, but SQL Server defaults to case-insensitive comparisons while MySQL behavior depends on the character set and collation selected. Transaction isolation differs: SQL Server defaults to READ COMMITTED with lock-based concurrency, while MySQL InnoDB defaults to REPEATABLE READ using MVCC. JSON support uses SQL Server's ISJSON(), JSON_VALUE(), JSON_QUERY(), and FOR JSON clause versus MySQL's native JSON data type, JSON_EXTRACT(), JSON_SET(), and the -> operator. Stored procedures use different syntactic conventions: SQL Server requires BEGIN...END blocks with DECLARE for variable declarations, while MySQL requires setting a custom DELIMITER before procedure definitions. Trigger capabilities also differ: SQL Server supports INSTEAD OF triggers and uses inserted and deleted pseudo-tables, while MySQL supports only BEFORE and AFTER triggers with OLD and NEW row references.

9.7.3 SQL Server vs. MySQL: CREATE DATABASE Syntax Comparison

In SQL Server, this command can specify file locations, initial sizes, growth parameters, and collation settings that determine character sorting and comparison rules. MySQL's CREATE DATABASE command is simpler, specifying only the database name and, optionally, the character set and collation.

SQL Server Example:

```
CREATE DATABASE University
ON PRIMARY (
 NAME = University_Data,
 FILENAME = 'D:\\Data\\UniversityDB.mdf',
 SIZE = 10MB,
 MAXSIZE = 100MB,
 FILEGROWTH = 5MB)
LOG ON (
 NAME = UniversityDB_Log,
 FILENAME = 'C:\\Data\\UniversityDB_log.ldf',
 SIZE = 2MB,
 MAXSIZE = 5MB,
 FILEGROWTH = 1MB);
```

MySQL Example:

```
CREATE DATABASE UniversityDB
CHARACTER SET utf8
COLLATE utf8_general_ci;
```

Both SQL Server and MySQL use the CREATE DATABASE command to create new databases, but they differ in syntax and the options available. SQL Server offers more detailed control over database files and growth, while MySQL emphasizes character set and collation settings.

Once the database exists, we use the USE database_name; command, which has identical syntax in both SQL Server and MySQL, to set it as the active database for subsequent commands.

Syntax Comparison Examples:

```
-- String Concatenation
-- SQL Server: SELECT First_name + ' ' + Last_name AS FullName FROM EMPLOYEE;
-- MySQL: SELECT CONCAT(First_name, ' ', Last_name) AS FullName FROM EMPLOYEE;
 -- Date Functions
-- SQL Server: SELECT GETDATE() AS CurrentDate, DATEDIFF(YEAR, Hire_date, GETDATE()) AS YearsOfService FROM EMPLOYEE;
-- MySQL: SELECT NOW() AS CurrentDate, TIMESTAMPDIFF(YEAR, Hire_date, NOW()) AS YearsOfService FROM EMPLOYEE;
-- Top N Records
-- SQL Server: SELECT TOP 10 * FROM PRODUCT ORDER BY Price DESC;
-- MySQL: SELECT * FROM PRODUCT ORDER BY Price DESC LIMIT 10;
-- Auto Increment
-- SQL Server: CREATE TABLE Orders ( Order_ID INT IDENTITY(1,1) PRIMARY KEY, Order_Date DATE );
-- MySQL: CREATE TABLE Orders ( Order_ID INT AUTO_INCREMENT PRIMARY KEY, Order_Date DATE );
```

9.8 Summary

SQL remains the core technology for relational database management, presenting a standardized yet flexible interface for defining structures, manipulating data, and controlling access across several database platforms. This chapter has demonstrated how SQL serves as the implementation bridge between conceptual database design and operational systems, tracing the journey from ER diagrams through the relational model to physical DDL implementation. We've explored SQL's rich query capabilities rooted in relational algebra, examined its transaction and security features, and highlighted practical differences across

major implementations such as SQL Server and MySQL. The extensive examples using University and Company databases illustrate SQL's application to actual scenarios, from simple selections to complex multi-table joins and aggregate analyses. As database technology continues to evolve with NoSQL alternatives, distributed systems, and cloud-native architectures, SQL's declarative paradigm and mature ecosystem ensure its sustained relevance. Mastering SQL requires understanding not just syntax but also the underlying relational model, query optimization principles, and implementation nuances across platforms, skills that remain essential for data professionals across all domains of present-day computing.

Review Questions

9.1. SQL: Definition, Characteristics, and Purpose: Define SQL and describe its four major characteristics: declarative nature, standardization, comprehensive functionality, and set-based processing. How does each characteristic contribute to SQL's widespread adoption across different database platforms and application domains?

9.2. Data Definition Language (DDL): Explain the purpose and scope of DDL in SQL. Describe the CREATE TABLE, ALTER TABLE, DROP TABLE, and TRUNCATE TABLE commands, including their syntax, use cases, and any irreversible effects. What additional objects can be defined using DDL beyond tables?

9.3. Data Manipulation Language (DML): Describe the four core DML commands: SELECT, INSERT, UPDATE, and DELETE. For each command, explain its purpose, basic syntax, and a practical use case from the University or Company databases presented in this chapter. What distinguishes DML from DDL in terms of the impact on the database?

9.4. Keys and Referential Integrity Constraints: Explain the purpose of PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, and CHECK constraints. How does referential integrity protect data consistency across related tables? Provide SQL syntax examples for each constraint type and describe what database violation occurs when a constraint is violated.

9.5. Creating a Database Table Schema and DML Operations: Describe the process of designing and creating a database table schema using DDL. Then explain and demonstrate the SQL syntax for INSERT, UPDATE, and DELETE operations, including how WHERE clauses control which rows are affected. What precautions should be taken before executing DELETE or UPDATE without a WHERE clause?

9.6. The SELECT Statement and Its Clauses: Explain the function of each clause in the SQL SELECT statement: SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, and LIMIT/TOP. In what order are these clauses logically evaluated by the database engine, and what restrictions apply to each clause? Illustrate with an example query using GROUP BY and HAVING together.

9.7. GROUP BY, WHERE, and HAVING: Explain how the GROUP BY clause partitions query results into groups for aggregate computation. Clearly distinguish between the WHERE clause (filters individual rows before grouping) and the HAVING clause (filters

groups after aggregation). Provide a SQL query example using both WHERE and HAVING in the same statement, and explain why they cannot generally be interchanged.

9.8. Subqueries vs. Joins and Multi-Step Queries: Compare and contrast subqueries and joins as mechanisms for combining data from multiple tables. Describe three types of subqueries and when each is appropriate. When is it preferable to use a multi-step query over a single complex join? Provide an example illustrating the trade-offs.

9.9. Database Views: Define a database view and explain how it differs from a base table. Describe the SQL syntax to CREATE and DROP a view. What are the advantages of views for security, abstraction, and query simplification? Under what conditions can a view be updated?

9.10. SQL Query Optimization and Execution: Describe the role of the query optimizer in SQL execution and explain four fundamental optimization techniques: selection pushdown, projection pushdown, join order optimization, and index utilization. How do execution plans help database administrators identify and resolve performance bottlenecks?

9.11. Data Control Language (DCL) and Database Security: Explain how DCL controls database security through the GRANT and REVOKE commands. Describe privilege levels (object-level and schema-level), role-based access control, and how views can be used as a security layer to restrict data exposure. Provide SQL syntax examples for granting SELECT and INSERT privileges, and revoking them.

9.12. SQL Server vs. MySQL: Key Differences: Identify and explain at least six significant differences between Microsoft SQL Server and MySQL, covering data types, string functions, date/time functions, auto-increment syntax, row-limiting clauses, transaction isolation defaults, and stored procedure syntax. For each difference, provide a concrete SQL example from both systems. Under what organizational or technical circumstances would you recommend one system over the other?

Exercises

9.1. Library Management System – Database Creation: Using the relational model of the Library Management System from Exercise 8.1: a) Write SQL DDL statements to CREATE DATABASE Library and create tables for Book, Member, and Staff relations, specifying appropriate data types, PRIMARY KEY constraints, and NOT NULL constraints for each attribute. b) Identify all primary keys and justify your choices. c) Write a basic SQL SELECT query to search for all books published after 2010, ordered alphabetically by title.

9.2. Library Management System – Relationships and Queries: Using the relational model from Exercise 8.2 (Library borrowing relationship): a) Write the SQL DDL to create the Borrowing table as a participating entity with appropriate FOREIGN KEY constraints referencing Book, Member, and Staff. b) Enforce referential integrity by specifying CASCADE or RESTRICT behavior on delete. c) Write a SQL SELECT query that lists all current borrowings, including the member's first and last name and the book title, using a JOIN operation.

9.3. Hospital Management System – Weak Entity and Cascading Deletes: Using the Hospital Management System from Exercise 8.3: a) Write SQL DDL statements to create the

Patient and Visit tables, correctly modeling Visit as a weak entity with a composite primary key (Patient_ID, Visit_No). b) Identify and specify the correct data types and constraints for all key attributes. c) Write a SQL query demonstrating what happens to Visit records when a Patient record is deleted, using ON DELETE CASCADE, and explain the behavior with a concrete example.

9.4. Cardinality and Participation – SQL Constraints: For each of the following business scenarios, write the SQL DDL with appropriate PRIMARY KEY, FOREIGN KEY, and cardinality constraints: a) "Each employee works for exactly one department, and each department has multiple employees." b) "Students can enroll in multiple courses, and courses can have multiple students enrolled." c) "Every order must be placed by exactly one customer, but customers may not have placed any orders yet." d) "An employee may manage at most one department, but not every employee is a manager."

9.5. Online Bookstore System – Database Creation and Queries: Using the relational model for the Online Bookstore System from Exercise 8.5: a) Write SQL DDL to create the complete Online Bookstore database, including tables for Customer, Book, Publisher, Order, and Order_Item with all PRIMARY KEY and FOREIGN KEY constraints. b) Identify all primary and foreign keys and explain each relationship. c) Write a SQL SELECT query to retrieve all books with a stock quantity less than 5, sorted by title, to support restocking decisions.

9.6. Restaurant Management System – Database Creation and Queries: Using the relational model for the Restaurant Management System from Exercise 8.6: a) Write SQL DDL to create the Restaurant Management database with tables for Customer, Reservation, Table, Menu_Item, and Order. b) Identify all primary and foreign keys. c) Write a SQL SELECT query to retrieve all reservations sorted by reservation date in descending order, including the customer name and party size.

9.7. University Course Scheduling System – Database and Enrollment Queries: Using the relational model for the University Course Scheduling System from Exercise 8.7: a) Write SQL DDL to create the University Course Scheduling database including tables for Course, Section, Professor, Classroom, TimeSlot, and Enrollment with all constraints. b) Identify all primary and foreign keys, including the composite keys for Section and Enrollment. c) Write a SQL SELECT query to list all students enrolled in each course section, showing the student name, course title, section number, classroom, and scheduled time, sorted by course title and then student last name.

9.8. Hospital Management System – Extended Schema and Patient Queries: Using the relational model for the Hospital Management System from Exercise 8.8: a) Write SQL DDL to create the extended hospital database, including tables for Doctor, Nurse, Patient, Department, Admission, Treatment, and Equipment with appropriate constraints, including cascading behavior for Admission as a weak entity. b) Identify all primary and foreign keys. c) Write a SQL SELECT query to retrieve all patients admitted since the beginning of the current year, sorted by admission date, including the patient name, diagnosis, and the name of the treating department.

9.9. E-Commerce Platform – Database and Shipment Reports: Using the relational model for the E-Commerce Platform from Exercise 8.9: a) Write SQL DDL to create the E-Commerce Platform database, including tables for Customer, Seller, Product, Category,

Order, OrderItem, Warehouse, and Shipment with all constraints. b) Identify all primary and foreign keys. c) Write a SQL SELECT query report for Shipments that lists each shipment together with its shipment items, showing the shipment ID, seller name, warehouse location, product name, quantity shipped, and shipping date.

9.10. Airline Reservation System – Database and Frequent Flyer Reports: Using the relational model for the Airline Reservation System from Exercise 8.10: a) Write SQL DDL to create the Airline Reservation System database, including tables for Airport, Aircraft, Flight, FlightInstance, Passenger, Reservation, Seat, Ticket, CrewMember, and FrequentFlyer with all constraints. b) Identify all primary and foreign keys, including composite keys for seat assignments and crew assignments. c) Write a SQL SELECT query to generate a frequent flyer report listing each enrolled passenger with their membership number, tier level (Silver, Gold, or Platinum), total mile balance, and the number of flights taken, sorted by mile balance in descending order.

References and Recommended Readings

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of Database Systems* (7th ed.). Pearson.

Connolly, T. M., & Begg, C. E. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson.

Date, C. J. (2003). *An Introduction to Database Systems* (8th ed.). Addison-Wesley.

Teorey, T. J., Lightstone, S. S., Nadeau, T., & Jagadish, H. V. (2011). *Database Modeling and Design: Logical Design* (5th ed.). Morgan Kaufmann.

Hoffer, J. A., Ramesh, V., & Topi, H. (2016). *Modern Database Management* (12th ed.). Pearson.

Coronel, C., & Morris, S. (2017). *Database Systems: Design, Implementation, & Management* (12th ed.). Cengage Learning.

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database System Concepts* (7th ed.). McGraw-Hill.

Ramakrishnan, R., & Gehrke, J. (2003). *Database Management Systems* (3rd ed.). McGraw-Hill.

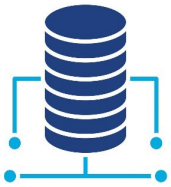
Hristoski, I., & Dimovski, T. (2022). *Dizajn na relaciski bazi na podatoci*, Tom 1 [Design of Relational Databases, Volume 1]. University "St. Kliment Ohridski", Bitola. (In Macedonian).

Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database Systems: The Complete Book* (2nd ed.). Pearson.

Part 4

Advanced Topics

Normalization



Learning Objectives:

By completing Part 4, you will be able to:

- Identify structural problems in relational schemas, including data redundancy and update, insertion, and deletion anomalies
- Define and apply functional dependencies to analyze relationships among attributes within a relation
- Transform relations through the normal forms — 1NF, 2NF, 3NF, and BCNF — using systematic decomposition
- Verify that a relational schema satisfies the requirements of each normal form
- Recognize when denormalization is an appropriate and intentional design trade-off
- Explain why databases designed through proper ER/EER modeling naturally tend toward normalized schemas
- Locate and interpret the complete relational schema and sample data for both case studies as a unified reference for all chapters in this book

***Chapter Overview*:**

Chapter 10: Normalization completes the logical design process by introducing formal tools for verifying and refining relational schemas. Building directly on the University and Company schemas developed in Part 3, this chapter explains why poorly structured tables cause anomalies, defines functional dependencies as the theoretical foundation of normalization, and guides the reader step by step through 1NF, 2NF, 3NF, and BCNF. The chapter closes by connecting normalization theory back to the ER-based design methodology used throughout this book, demonstrating that good design practice and formal correctness are two sides of the same coin.

Chapter 11: Integrated Case Studies consolidates the complete relational schemas and sample data for both case studies into a single authoritative reference. Rather than tracing partial schemas across earlier chapters, the reader will find here the full definitions of all relations — with primary keys, foreign keys, and instance data — for both the University Management System and the Company/Business Database. This chapter serves as the definitive reference point for students and instructors throughout the course.

Chapter 10

10. Normalization

10.1 Why Normalization Matters

Throughout the preceding chapters, we built our University and Company databases by following a clear path: we gathered requirements, created ER and EER diagrams, mapped those diagrams to a relational schema, and implemented the result in SQL. This methodology naturally produces well-structured tables. But not every database begins with careful design. Legacy systems, quick prototypes, spreadsheets converted into databases, and designs that grew organically over time often end up with tables that store too much information in a single place. **Normalization** is the formal process that allows us to detect, understand, and correct these structural problems.

Even for databases built through proper ER modeling, including ours, normalization serves an important purpose: it provides a rigorous theoretical framework to *verify* that our designs are correct. After working through this chapter, you will be able to prove, not just assume, that the University and Company schemas developed in Chapter 7 are sound.

10.1.1 The Problem with Storing Everything Together

To understand what normalization solves, consider what happens when we abandon table discipline and store related information together in a single, flat table. Suppose that early in the development of the university system, before any design methodology had been applied, an administrator created a single spreadsheet to track student enrollments. Instead of separate tables for students, courses, faculties, and professors, everything was combined into one table, which we will call STUDENT_ENROLLMENT_FLAT:

Table 10.1: STUDENT_ENROLLMENT_FLAT (unnormalized)

Student_ID	First_name	Last_name	Course_ID	Course_title	Credits	Faculty_name	Professor_name
S002	Bojan	Jovansen	EE201	Circuit Analysis	6	Electrical Engineering	Naum Ohridski
S002	Bojan	Jovansen	EE301	Advanced Circuit Analysis	7	Electrical Engineering	Naum Ohridski
S003	Kosta	Dojcinoski	MA301	Linear Algebra	6	Bioengineering	Kiril Prosvetitel

S003	Kosta	Dojcinoski	MA401	Matrix Theory and Advanced Vector Spaces	7	Bioengineering	Kiril Prosvetitel
S004	David	Boskovski	PH101	Physics	8	Electrical Engineering	Metodij Prosvetitel
S004	David	Boskovski	PH201	Advanced Physics	9	Electrical Engineering	Metodij Prosvetitel
S006	Anica	Stojnasen	CS101	Intro to Programming	5	Computer Science	Kliment Ohridski
S006	Anica	Stojnasen	CS201	Advanced Programming	6	Computer Science	Kliment Ohridski
S006	Anica	Stojnasen	CS303	Database Systems	6	Computer Science	Richard Johnson

At first glance, this table appears to contain all the information we need. A quick scan shows us which students are enrolled in which courses, who teaches each course, and which faculty the course belongs to. The problems, however, become apparent the moment we attempt to update or maintain it.

10.1.2 Update Anomalies

Suppose the Faculty of Electrical Engineering is renamed to the Faculty of Electrical Engineering and Technology following a university restructuring. In Table 10.1, the name "Electrical Engineering" appears in four separate rows, for all courses taken by students S002 and S004. A correct update requires modifying every one of those rows. If a single row is missed — perhaps because the update query was run with an incorrect filter, or because new rows were inserted after the update — the database ends up with two different names for the same faculty. Part of the table now says *"Electrical Engineering"* while the rest says *"Electrical Engineering and Technology."* The data has become **inconsistent**.

This is an ***update anomaly***: a single real-world change requires multiple row modifications, and the risk of inconsistency grows with every duplicate.

The same problem applies to professor data. Professor Naum Ohridski's name appears twice in Table 10.1 — once for each course Bojan Jovansen is enrolled in. Any attribute we might want to store about Professor Ohridski (his office, hire date, or academic title) would have to be repeated in every enrollment row for every student taking his courses. In our actual university database, Professor Ohridski teaches EE201 and EE301, which means his information would repeat once for every enrolled student — a potentially large number of redundant values, all of which must be kept perfectly synchronized.

10.1.3 Insertion Anomalies

Now, suppose the Faculty of Computer Science has approved a new course — CS401, Software Engineering, worth 6 credits, to be taught by Professor Richard Johnson. The course is ready for enrollment, but no students have yet signed up.

In STUDENT_ENROLLMENT_FLAT, there is no way to record this course without attaching it to a student. Every row requires a Student_ID, a First_name, and a Last_name. We would have to either leave those fields empty (storing **NULL** in fields that have no natural NULL value) or invent a placeholder student — both of which are unacceptable in a properly managed database.

This is an ***insertion anomaly***: the table's structure makes it impossible to record legitimate, independent facts — in this case, the existence of a course — without fabricating or omitting other data.

The same problem arises in the Company system when imagined as a flat structure. If a new supplier has been approved but has not yet supplied any products, there is no way to insert that supplier into a flat ORDER_PRODUCT_SUPPLIER table without associating it with a product and an order that do not yet exist.

10.1.4 Deletion Anomalies

Consider what happens at the end of an academic year when we archive old enrollment records. Student S003, Kosta Dojcinoski, is enrolled in two courses: MA301 Linear Algebra and MA401 Matrix Theory and Advanced Vector Spaces, both taught by Professor Kiril Prosvetitel and both belonging to the Faculty of Bioengineering. In Table 10.1, these are the *only* two rows that contain any information about the Faculty of Bioengineering, about these two courses, and about Professor Prosvetitel.

When S003's enrollment records are deleted, we lose not just the student's data. We lose the only records in the table telling us that MA301 and MA401 exist, that they carry 6 and 7 credits respectively, that they belong to the Faculty of Bioengineering, and that Professor Kiril Prosvetitel teaches them. All of this information disappears as an unintended side effect of a routine administrative operation.

This is a ***deletion anomaly***: removing one piece of information causes the unintended loss of entirely different, independently valuable information. In a properly structured database, deleting a student's enrollment records would affect only the ENROLLS_IN table. The COURSE, FACULTY, and PROFESSOR tables would remain intact, preserving all course, faculty, and professor data regardless of enrollment activity.

10.1.5 The Root Cause: Mixing Independent Facts

What all three anomalies have in common is that Table 10.1 mixes facts about fundamentally different things into a single structure. It conflates information about students, courses, faculties, and professors, all of which exist *independently* of one another in the real world. A course exists whether or not any student is enrolled in it. A professor exists whether or not they are currently teaching. A faculty exists whether or not it currently has active enrollments.

When independent facts are stored together, every row must carry **redundant** copies of information that belongs elsewhere. The name "Electrical Engineering" appears four times in Table 10.1 because four enrollment records involve courses from that faculty, not because there are four faculties. This **redundancy** is not just wasteful; it is the structural source of all three anomaly types described above.

Normalization is the systematic process of decomposing such tables into smaller, *well-structured* relations in which every piece of information is stored exactly once, in exactly the right place. The **normal forms** — *1NF*, *2NF*, *3NF*, and *BCNF* — are a progressive set of rules that guide this decomposition, with each level addressing a different category of structural problem.

As we will see in the sections that follow, the University and Company schemas developed through our ER-based design process in Chapter 7 already satisfy these rules. Working through the normal forms with our own data will both reinforce the theory and give you a concrete framework for evaluating any relational design you encounter in professional practice.

10.2 Functional Dependencies

A fundamental prerequisite for understanding normalization is the concept of **functional dependency**. Functional dependencies describe the relationships between the *attributes* of a relation and provide the theoretical foundation upon which all normal forms are built. Before examining individual normal forms, therefore, we must develop a precise understanding of what functional dependencies are, how they are discovered, and how they interact with the concept of keys.

A functional dependency is a property of the **meaning** of attributes, not merely of the data currently stored in a relation. The fact that a particular set of data happens to exhibit a pattern is insufficient evidence of a true functional dependency; the dependency must hold for every possible valid state of the relation, now and in the future. This distinction is critical in practice because developers who derive dependencies solely from inspecting sample data often miss constraints that are violated only in specific cases or in upcoming data.

For the discussion that follows, we assume a relational schema with a set of attributes, each bearing a unique name. This assumption mirrors the relational model itself, in which every column in a given relation must be uniquely named, even when the same real-world concept appears in multiple tables.

10.2.1 Definition and Notation

> **Functional Dependency (FD)**
>
> Given a relation R with attributes X and Y (each of which may consist of one or more attributes), Y is functionally dependent on X — written X → Y — if and only if each value of X is associated with exactly one value of Y in every valid state of R. X is called the determinant; Y is called the dependent. (Codd, 1970; Elmasri & Navathe, 2016)

The arrow notation X → Y is read "X functionally determines Y" or "Y is functionally dependent on X." The direction of the arrow is significant: it indicates that knowledge of X is

sufficient to determine Y, but knowledge of Y alone is not necessarily sufficient to determine X. For a given value of Y, there may be many different values of X, which is why functional dependency is an asymmetric relationship.

When two tuples in a relation agree on the value of X, they must also agree on the value of Y. This constraint must be observed not just for the data currently in the database, but for all data that could legitimately be inserted in the future. The designer therefore derives functional dependencies from the domain's semantics—from business rules and real-world constraints—rather than from inspecting table contents.

Determinant: The attribute or set of attributes on the left-hand side of the arrow in a functional dependency. The determinant is the source of the unique identification.

University Database System— Functional Dependencies

Consider the STUDENT relation from Chapter 7:

STUDENT (Student_ID, First_name, Last_name, Birth_date, Age, Street, City, Country, Zip_code, Faculty_ID*)

The following functional dependencies conform to this relation:

- Student_ID → First_name, Last_name, Birth_date, Age, Street, City, Country, Zip_code, Faculty_ID

Because Student_ID is the primary key, knowing the Student_ID uniquely determines every other attribute. Two different students will never share the same Student_ID, so if two tuples agree on Student_ID, they must agree on all remaining attributes.

- Zip_code → City, Country

This dependency arises from real-world geography: a given postal code belongs to exactly one city in one country. Knowing the Zip_code is sufficient to determine the City and Country.

Consider also the COURSE relation:

COURSE (Course_ID, Title, Credits, Description, Faculty_ID*, Professor_ID*)

- Course_ID → Title, Credits, Description, Faculty_ID, Professor_ID

Every course has a unique Course_ID, so the course identifier determines all other attributes, including which faculty offers it and which professor teaches it.

And in the PROFESSOR relation:

PROFESSOR (Professor_ID, First_name, Last_name, Office, Email, Hire_date, Academic_title, Faculty_ID*, Heads_Faculty_ID*)

- Professor_ID → First_name, Last_name, Office, Email, Hire_date, Academic_title, Faculty_ID, Heads_Faculty_ID
- Email → Professor_ID, First_name, Last_name, ...

Email is a candidate key for PROFESSOR because no two professors share the same email address. Both Professor_ID and Email are therefore determinants for the full set of professor attributes.

Company Database System — Functional Dependencies

Consider the EMPLOYEE relation from Chapter 7:

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Country, Zip_code, Department_ID*, Warehouse_ID*, Manages_Dept_ID*)

The following functional dependencies hold:

- Employee_ID → First_name, Last_name, Email, Salary, Hire_date, Street, City, Zip_code, Country, Department_ID, Warehouse_ID, Manages_Dept_ID

Employee_ID is the primary key and uniquely identifies all attributes of an employee record.

- Email → Employee_ID, First_name, Last_name, ...

Because every employee has a unique Email address, the email is also a candidate key, a second determinant for the full employee record.

- Zip_code → City, Country

As with the University system, postal codes determine the city and country, a real-world geographic dependency that holds across both case studies.

In the DEPARTMENT relation:

DEPARTMENT (Department_ID, Dept_name, Location, Budget)

- Department_ID → Dept_name, Location, Budget
- Dept_name → Department_ID, Location, Budget

Both Department_ID and Dept_name are candidate keys, since no two departments share the same name. Each is therefore a valid determinant for all other department attributes.

10.2.2 Full and Partial Functional Dependency

Full and partial functional dependencies become relevant only when the determinant X is a composite attribute — that is, when X consists of two or more attributes. If X is a single attribute, the dependency is trivially full (there is no proper subset to remove). The distinction between full and partial dependency is therefore central to Second Normal Form, which prohibits partial dependencies on composite primary keys.

> **Full Functional Dependency**
>
> A functional dependency X → Y is a full functional dependency if removing any attribute from X causes the dependency to no longer hold. That is, Y depends on the whole of X and not on any proper subset of X. (Elmasri & Navathe, 2016)

> **Partial Functional Dependency**
>
> A functional dependency X → Y is a partial functional dependency if there exists a proper subset X' ⊂ X such that X' → Y still holds. Y depends on only part of X, not the whole of it. (Elmasri & Navathe, 2016)

University Database System— Full vs. Partial FD

The junction relation ENROLLS_IN has a composite primary key:

ENROLLS_IN (Student_ID*, Course_ID*, Enrollment_date, Grade)

Consider the functional dependencies involving this composite key:

{Student_ID, Course_ID} → Enrollment_date, Grade

This is a full functional dependency. The Grade and Enrollment_date are properties of a specific enrollment, with the combination of a particular student in a particular course. Removing either Student_ID or Course_ID from the left-hand side breaks the dependency: knowing only Course_ID does not tell you a student's grade, and knowing only Student_ID does not tell you which course's grade is meant.

Now imagine a poorly designed table that also stores course attributes:

ENROLLS_IN_ALT (Student_ID*, Course_ID*, Enrollment_date, Grade, Course_Title, Credits)

Course_ID → Course_Title, Credits

This is a partial functional dependency. Course_Title and Credits depend only on Course_ID — a proper subset of the composite key {Student_ID, Course_ID}. The student's identity contributes nothing to determining what a course is called or how many credits it carries. This partial dependency indicates a design flaw: course properties should live in the COURSE relation, not in ENROLLS_IN.

Company Database System — Full vs. Partial FD

The ORDER_LINE relation also has a composite primary key:

ORDER_LINE (Order_ID*, Line_number, Unit_price, Quantity, Discount, Product_ID*)

{Order_ID, Line_number} → Unit_price, Quantity, Discount, Product_ID

This is a full functional dependency. The unit price, quantity, and discount on a given order line are determined jointly by the order and the line number; neither alone is sufficient.

Consider a flawed version of the STORED_IN relation that embeds product attributes alongside inventory data:

STORED_IN_ALT (Product_ID*, Warehouse_ID*, Quantity_on_hand, Reorder_level, Product_name)

Product_ID → Product_name

This is a partial functional dependency. Product_name depends only on Product_ID — a proper subset of the composite primary key {Product_ID, Warehouse_ID}. The warehouse plays no role in determining a product's name; that name is a fixed property of the product itself, regardless of where it is stored. This partial dependency indicates a design flaw: product attributes such as Product_name belong in the PRODUCT relation, not in STORED_IN. Storing them here creates redundancy — the same product name would repeat once for every warehouse that stocks that product — and introduces update anomalies whenever a product name changes.

10.2.3 Transitive Functional Dependency

A transitive dependency arises when a non-key attribute depends not directly on the primary key but on another non-key attribute that itself depends on the primary key. The result is that changes to the intermediate attribute Y can cause inconsistencies: if Y's value changes for one tuple, Z's value must also change, yet nothing in the relational model enforces this automatically unless Y and Z are stored together in a separate table.

Transitive dependencies are the structural problem that Third Normal Form is designed to eliminate. Identifying them requires tracing the chain of dependencies from the primary key through all non-key attributes, looking for cases where the dependency path takes an intermediate step through a non-key column.

> **Transitive Functional Dependency**
>
> Given a relation R and attributes X, Y, Z: if $X \rightarrow Y$ and $Y \rightarrow Z$, and Y does not functionally determine X ($Y \not\rightarrow X$), then Z is transitively dependent on X through Y. Z depends on X indirectly — through an intermediate non-key attribute Y. (Elmasri & Navathe, 2016)

University Database System— Transitive FD

Suppose we defined a denormalized version of the STUDENT relation that included faculty attributes directly:

STUDENT_ALTERNATIVE (Student_ID, First_name, Last_name, Birth_date, Age, Street, City, Country, Zip_code, Faculty_ID, Faculty_name, Faculty_location)

The following chain of dependencies exists:

- Student_ID → Faculty_ID
- Faculty_ID → Faculty_name, Faculty_location

Therefore: Student_ID → Faculty_ID → Faculty_name, Faculty_location, Faculty_name and Faculty_location are transitively dependent on Student_ID through Faculty_ID. They do not describe the student directly — they describe the faculty that the student belongs to. If the Faculty of Electrical Engineering moves to a new building, every student row affiliated with that faculty must be updated. This is an update anomaly arising directly from the transitive dependency.

The correct design — already shown in Chapter 7 — is to store faculty attributes in a separate FACULTY relation and reference it via the Faculty_ID foreign key.

FACULTY (Faculty_ID, Name, Budget, Location)

STUDENT (Student_ID, First_name, Last_name, Birth_date, ..., Faculty_ID*)

Company Database System — Transitive FD

Consider a denormalized EMPLOYEE table that embeds department attributes:

EMPLOYEE_ALTERNATIVE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Department_ID, Dept_name, Dept_location, Dept_budget)

The chain of dependencies:

- Employee_ID → Department_ID

- Department_ID → Dept_name, Dept_location, Dept_budget

Therefore: Employee_ID → Department_ID → Dept_name, Dept_location, Dept_budget Dept_name, Dept_location and Dept_budget are transitively dependent on Employee_ID through Department_ID. They describe the department, not the individual employee. If the Sales department relocates from Skopje to Bitola, every employee row in the Sales department must be updated — an update anomaly caused by this transitive dependency.

The Chapter 7 schema correctly separates these:

DEPARTMENT (Department_ID, Dept_name, Location, Budget)

EMPLOYEE (Employee_ID, ..., Department_ID*)

10.2.4 Keys Revisited: Superkeys, Candidate Keys, and Primary Keys

Chapter 7 introduced several types of keys in the context of the relational model. Those definitions are now revisited through the lens of functional dependency (FD) theory, which provides a more formal and precise basis for understanding why keys are essential to normalization, as shown in Table 10.2.

Table 10.2: Mapping Table database keys and Relationship to functional dependency

Key Type	Definition	Relationship to FD
Superkey	Any set of attributes that uniquely identifies every tuple in a relation	A superkey K functionally determines every attribute in the relation: K → all attributes
Candidate Key	A minimal superkey: no proper subset of it is also a superkey	A candidate key is a full functional determinant for all other attributes
Primary Key	The candidate key chosen by the designer to serve as the principal identifier	Establishes the primary FD from which 2NF and 3NF are evaluated
Alternate Key	Any candidate key not chosen as the primary key	Still a valid determinant; relevant for BCNF analysis
Composite Key	A key consisting of two or more attributes	Enables partial FDs if non-key attributes depend on only part of the key
Foreign Key	An attribute referencing the primary key of another relation	Enforces referential integrity; itself a determinant within the referenced relation

The concept of *prime attribute* is also important for normalization: a prime attribute is any attribute that participates in at least one candidate key. A *non-prime attribute* is one that does not participate in any candidate key. The normal forms 2NF and 3NF are specifically concerned with how non-prime attributes depend on the primary key.

University Database System— Keys and functional dependency

In the PROFESSOR relation, the candidate keys are Professor_ID and Email:

- Professor_ID → {all attributes} (primary key)
- Email → {all attributes} (alternate key)

Both are minimal: neither attribute set has a proper subset that would still uniquely identify professors. Both are fully functional determinants for every non-key attribute in the relation.

In the ENROLLS_IN relation, the composite primary key is {Student_ID, Course_ID}:

- {Student_ID, Course_ID} → Enrollment_date, Grade

Student_ID and Course_ID are each prime attributes. Enrollment_date and Grade are non-prime attributes. A partial dependency would exist if either Enrollment_date or Grade depended on only Student_ID or only Course_ID, but neither does, so ENROLLS_IN is in 2NF.

Company Database System — Keys and functional dependency

In the EMPLOYEE relation, the candidate keys are Employee_ID and Email:

- Employee_ID → {all attributes} (primary key)
- Email → {all attributes} (alternate key)

Both are minimal: neither attribute set has a proper subset that would still uniquely identify an employee. Both are fully functional determinants for every non-key attribute in the relation.

In the ORDER_LINE relation, the composite primary key is {Order_ID, Line_number}:

- {Order_ID, Line_number} → Unit_price, Quantity, Discount, Product_ID

Order_ID and Line_number are each prime attributes. Unit_price, Quantity, Discount, and Product_ID are non-prime attributes. A partial dependency would exist if any of these attributes depended on only Order_ID or only Line_number — but none do, so ORDER_LINE is in 2NF.

10.3 First Normal Form (1NF)

The First Normal Form is the foundational requirement of the relational model itself. A relation that fails to satisfy 1NF is not truly a relation in the mathematical sense; it violates the basic atomicity principle that underlies the entire relational paradigm. Every higher normal form assumes that 1NF has already been achieved; normalization, therefore, always begins here.

10.3.1 Definition and Requirements

> **First Normal Form (1NF)**
>
> A relation R is in First Normal Form if and only if every attribute in every tuple contains exactly one atomic (indivisible) value drawn from the attribute's domain. The relation must have no repeating groups, no multi-valued attributes in a single column, and no nested or composite values within a single cell. (Codd, 1970)

The four *requirements* for 1NF can be stated as follows:

- **Atomic values**: Every cell in the relation must contain a single, indivisible value. A cell may not contain a list, set, array, or any other composite structure.

- **Unique column names**: Every attribute in a relation must have a distinct name. No two columns may represent the same concept under different labels.
- **Consistent domain per column**: All values in a given column must be drawn from the same domain. A column defined as holding integers may not occasionally hold strings.
- **No repeating groups**: The relation must not contain groups of related columns that repeat a pattern (e.g., Phone1, Phone2, Phone3). Such patterns must be expressed as separate tuples in a related table.

The connection between 1NF and the ER modeling process covered in Chapters 3 through 5 is direct: when designers identify **multi-valued** attributes (attributes that can hold more than one value for a single entity instance) in an ER diagram, they are implicitly detecting future 1NF violations. The standard treatment of multi-valued attributes during ER-to-relational mapping is creating a separate relation to hold the multiple values and is precisely the instrument for achieving 1NF.

10.3.2 Recognizing 1NF Violations

The most common 1NF violations encountered in practice fall into two categories: *attributes* that hold multiple values within a single column (multi-valued attributes), and *repeating-group* column patterns where related information is expressed as a fixed number of separate columns rather than as separate rows.

Multi-valued attributes embedded in a single column are often identifiable by separator characters, comma-separated lists, semicolon-delimited strings, or JSON arrays stored in a text field. While modern database engines can store such values, the relational model does not treat them as atomic, and querying or updating individual elements becomes cumbersome and error-prone.

Repeating-group patterns are common in spreadsheet-to-database migrations, where additional values were historically added as new columns rather than as new rows. Columns named Phone1, Phone2, and Phone3, or Address_Line_1 through Address_Line_5, are telltale signs of this anti-pattern.

10.3.3 Achieving 1NF: University and Company Examples

The University and Company schemas developed in Chapter 7 already satisfy 1NF because multi-valued attributes were correctly handled during the ER-to-relational mapping process (Step 6 of the mapping algorithm). The following examples demonstrate what the violation would look like and how the Chapter 7 schema resolves it.

University Database System— 1NF

The STUDENT entity in the ER diagram (Figure 4.41) includes Email as a multi-valued attribute, indicating that a single student may have multiple email addresses (a personal address and a university address, for instance).

A 1NF-violating representation would store multiple emails in a single column:

STUDENT_ALTERNATIVE (Student_ID, First_name, Last_name, ..., Email)

Where Email might contain: "sarah@gmail.com; sarah.smith@university.edu"

This violates atomicity. The Chapter 7 solution (Step 6: Mapping Multi-valued Attributes) creates a separate EMAIL relation:

STUDENT (Student_ID, First_name, Last_name, Birth_date, Age, Street, City, Country, Zip_code, Faculty_ID*)

EMAIL (Student_ID*, Email)

The EMAIL relation satisfies 1NF: every cell contains exactly one email address. A student with two addresses is represented by two rows in EMAIL, not by a comma-separated list in a single cell. The composite primary key {Student_ID, Email} ensures no duplicate email is recorded for the same student.

🏢 *Company Database System — 1NF*

In the Company ER diagram, EMPLOYEE, CUSTOMER, and SUPPLIER each have Phone as a multi-valued attribute, reflecting the reality that organizations maintain multiple contact numbers for a single person or business.

A 1NF violation would occur if phone numbers were stored as a repeating-group pattern:

EMPLOYEE_ALTERNATIVE (Employee_ID, First_name, Last_name, ..., Phone1, Phone2, Phone3)

This pattern violates 1NF in two ways: it creates repeating groups (Phone1, Phone2, Phone3 all represent the same concept), and it imposes an artificial limit of three phone numbers.

The Chapter 7 schema resolves this correctly with dedicated phone tables:

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, ...)

EMPLOYEE_PHONE (Employee_ID*, Phone_number)

CUSTOMER (Customer_ID, First_name, Last_name, Email, ...)

CUSTOMER_PHONE (Customer_ID*, Phone_number)

SUPPLIER (Supplier_ID, Supplier_name, ...)

SUPPLIER_PHONE (Supplier_ID*, Phone_number)

Each phone number now occupies its own row with an atomic value. The number of phone entries per entity is unlimited, and querying for a specific phone number requires only a simple predicate rather than searching across multiple columns.

10.4 Second Normal Form (2NF)

The Second Normal Form addresses a specific class of redundancy that arises only in tables with composite primary keys. It builds directly on the concept of full functional dependency introduced in Section 10.2.2, and requires that every non-prime attribute depend fully on the entire primary key, rather than on just a portion of it.

10.4.1 Definition and Requirements

Second Normal Form (2NF)

A relation R is in Second Normal Form if and only if: (1) R is in 1NF, and (2) every non-prime attribute in R is fully functionally dependent on the entire primary key of R. That is, no non-prime attribute is partially dependent on any proper subset of the primary key. (Codd, 1971)

The implication for tables with single-attribute primary keys is immediate: if the primary key consists of a single attribute, there are no proper subsets to which a partial dependency could exist. A relation with a single-column primary key that is already in 1NF is automatically in 2NF. Second Normal Form is therefore only a meaningful constraint to check for relations with composite primary keys.

When a *partial* dependency exists, the consequence is *redundancy*: the dependent attribute is repeated in every row that shares the same subset-key value. This redundancy creates the same update, insertion, and deletion anomalies described in Section 10.1. The remedy is decomposition: separating the partially dependent attribute into a new relation along with the subset of the key that determines it.

10.4.2 Identifying Partial Dependencies

The systematic process for identifying **partial** dependencies in a relation with a **composite** primary key {X, Y} proceeds as follows:

1. List all non-prime attributes of the relation.
2. For each non-prime attribute N, test whether X → N holds (i.e., whether X alone determines N).
3. For each non-prime attribute N, test whether Y → N holds (i.e., whether Y alone determines N).
4. If either test succeeds, a partial dependency exists: N is only partially dependent on {X, Y}.
5. Non-prime attributes that pass neither test — requiring both X and Y together to be determined — are **fully** functionally dependent on the composite key.

Partial dependencies are detected by applying *domain* knowledge, not by statistical analysis of data. The question to ask is: "Does knowing only the value of X (regardless of Y) give you a unique value for N in every possible valid state of the database?" If the answer is yes, a partial dependency is *present*.

10.4.3 Decomposition to 2NF: University Example

University Database System

Consider a poorly designed version of the enrollment table that embeds course properties alongside enrollment-specific data:

ENROLLS_IN_ALT (Student_ID*, Course_ID*, Enrollment_date, Grade, Course_Title, Credits, Professor_Last_name)

The *composite* primary key is {Student_ID, Course_ID}.

Let us identify functional dependencies:

- {Student_ID, Course_ID} → Enrollment_date, Grade

Full dependency — both attributes of the key are required.

- Course_ID → Course_Title, Credits, Professor_Last_name

Partial dependency — Course_Title, Credits, and Professor_Last_name are determined by Course_ID alone, independently of which student is enrolled.

Decomposition to 2NF:

Separate the partial dependencies into their own relation:

ENROLLS_IN (Student_ID*, Course_ID*, Enrollment_date, Grade)

COURSE (Course_ID, Title, Credits, Professor_ID*)

ENROLLS_IN now contains only attributes that require both Student_ID and Course_ID for their determination. COURSE contains the attributes that depend solely on Course_ID. Both relations are in 2NF, all anomalies are eliminated, and each fact is stored exactly once.

Professor_Last_name is replaced by Professor_ID* in the decomposed COURSE relation, referencing the PROFESSOR relation rather than duplicating a name attribute — consistent with the design principle that entity data belongs in the entity's own relation.

10.4.4 Decomposition to 2NF: Company Example

Company Database System — 2NF Decomposition

Consider a denormalized version of the ORDER_LINE table that identifies each line by the order and the product ordered, and embeds product properties alongside transaction data:

ORDER_LINE_ALT (Order_ID*, Product_ID*, Unit_price, Quantity, Discount, Product_name, Category, List_price, Weight)

The composite primary key for this design is {Order_ID, Product_ID}.

Functional dependency analysis reveals:

- {Order_ID, Product_ID} → Unit_price, Quantity, Discount

Full dependency — both components of the key are needed to determine the transaction-specific details of a line.

- Product_ID → Product_name, Category, List_price, Weight

Partial dependency — Product_name, Category, List_price, and Weight are determined by Product_ID alone. A product's name and price are properties of the product, not of a particular order line.

Decomposition to 2NF:

ORDER_LINE (Order_ID*, Product_ID*, Unit_price, Quantity, Discount)

PRODUCT (Product_ID, Product_name, Description, Category, Price, Weight)

Note: The Chapter 7 schema uses a Line_number attribute as part of the ORDER_LINE primary key rather than Product_ID directly, which allows a single order to contain multiple lines for the same product. The decomposition result for PRODUCT is identical under both designs; what changes is only the key structure of ORDER_LINE. The normalized outcome — permanently separating product catalogue data from transaction-specific line data — is the same. The Unit_price stored on each ORDER_LINE captures the price at the time of sale, which may legitimately differ from the current Price in PRODUCT — an important business distinction that the normalized design makes possible.

10.5 Third Normal Form (3NF)

Third Normal Form eliminates a more subtle form of redundancy than 2NF: the transitive dependency. Even after a relation is in 2NF — meaning every non-prime attribute depends fully on the primary key — redundancy can persist if non-prime attributes depend on each other rather than directly on the primary key. Third Normal Form requires that these transitive paths be broken and the intermediate dependencies housed in separate relations.

10.5.1 Definition and Requirements

Define 3NF: the table must be in 2NF, and no non-prime attribute must transitively depend on the primary key through another non-primary attribute.

> **Third Normal Form (3NF)**
>
> A relation R is in Third Normal Form if and only if: (1) R is in 2NF, and (2) no non-prime attribute in R is transitively dependent on the primary key through another non-prime attribute. Equivalently, for every functional dependency X → Y within R where Y is not a subset of X, either X is a superkey of R, or every attribute in Y is a prime attribute. (Codd, 1971)

The second formulation of 3NF, given by Codd (1971), is sometimes expressed informally as: "Every non-key attribute must depend on the key, the whole key, and nothing but the key." The first clause ("the key") rules out transitive dependencies through non-key attributes; the second ("the whole key") is 2NF; the third ("nothing but the key") addresses certain edge cases covered by BCNF, discussed in Section 10.6.

The practical test for transitive dependency is straightforward: given the primary key PK and a non-prime attribute A, ask whether there exists another non-prime attribute B such that PK → B and B → A. If such a B exists, and B does not determine PK (i.e., B is not a candidate key), then A is transitively dependent on PK through B, this is a 3NF violation.

10.5.2 Identifying Transitive Dependencies

Transitive dependencies often arise when a relation conflates information about two different real-world entities. A student table that also stores the student's faculty name is conflating information about students and faculty. An employee table that stores the

department's budget is conflating information about employees and departments. The intermediate attribute, Faculty_ID in the first case, Department_ID in the second, is the bridge across which the transitive dependency travels.

The detection method is to scan each non-prime attribute and ask: "Does this attribute describe the entity identified by the primary key, or does it describe something else that the primary key merely references?" If the attribute describes something other than the primary entity, it is likely to be involved in a transitive dependency.

10.5.3 Decomposition to 3NF: University Example

University Database System— 3NF Decomposition

The following example demonstrates the decomposition of a denormalized STUDENT table — one that embeds faculty attributes directly — into separate, normalized STUDENT and FACULTY relations.

Suppose a combined student table were constructed before proper normalization:

STUDENT_ALT (Student_ID, First_name, Last_name, Birth_date, Age, Street, City, Country, Zip_code, Faculty_ID, Faculty_name, Faculty_budget, Faculty_location)

This relation is in 2NF (no composite key, so no partial dependencies), but it contains a transitive dependency:

- Student_ID → Faculty_ID
- Faculty_ID → Faculty_name, Faculty_budget, Faculty_location

Therefore: Student_ID → Faculty_ID → Faculty_name, Faculty_budget, Faculty_location, Faculty_name, Faculty_budget, and Faculty_location are transitively dependent on Student_ID through Faculty_ID. They describe the faculty, not the student.

Decomposition to 3NF:

FACULTY (Faculty_ID, Name, Budget, Location)

STUDENT (Student_ID, First_name, Last_name, Birth_date, Age, Street, City, Country, Zip_code, Faculty_ID*)

FACULTY captures all information about faculties. STUDENT references FACULTY via the Faculty_ID foreign key. Each fact is stored once; updating a faculty name requires changing a single row in FACULTY, and the change is automatically reflected for all associated students.

The same pattern applies to the COURSE relation. A denormalized COURSE table that included Professor_last_name and Professor_office would introduce:

- Course_ID → Professor_ID → Professor_last_name, Professor_office

The Chapter 7 schema avoids this by storing professor attributes exclusively in PROFESSOR and referencing them via Professor_ID* in COURSE.

10.5.4 Decomposition to 3NF: Company Example

Company Database System — 3NF Decomposition

Consider a denormalized EMPLOYEE table that embeds department details:

EMPLOYEE_ALT (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Country, Zip_code, Department_ID, Dept_name, Dept_location, Dept_budget, Warehouse_ID*, Manages_Dept_ID*)

This relation is in 2NF but violates 3NF through a transitive dependency:

- Employee_ID → Department_ID
- Department_ID → Dept_name, Dept_location, Dept_budget

Therefore, Employee_ID → Department_ID → Dept_name, Dept_location, Dept_budget, Dept_name, Dept_location, and Dept_budget describe the department, not the employee. If the Marketing department relocates from Bitola to Skopje, every row for every Marketing employee must be updated.

Decomposition to 3NF:

DEPARTMENT (Department_ID, Dept_name, Location, Budget)

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Country, Zip_code, Department_ID*, Warehouse_ID*, Manages_Dept_ID*)

Note that Department_ID changes from a plain attribute in EMPLOYEE_ALT to Department_ID* in the decomposed result — it becomes a foreign key referencing the new DEPARTMENT relation. Warehouse_ID* and Manages_Dept_ID* carry through unchanged, since they were already proper foreign key references and play no role in the transitive dependency being resolved.

A similar analysis applies to the WAREHOUSE relation. If EMPLOYEE stored warehouse location and capacity directly, rather than referencing them via Warehouse_ID*, the warehouse attributes would be transitively dependent on Employee_ID through Warehouse_ID. The Chapter 7 schema correctly isolates these:

WAREHOUSE (Warehouse_ID, Location, Capacity, Department_ID*)

EMPLOYEE (..., Warehouse_ID*, ...)

With this design, updating a warehouse's location requires a single change to the WAREHOUSE table, automatically effective for all employees who work at that warehouse.

10.6 Boyce-Codd Normal Form (BCNF)

Boyce-Codd Normal Form (BCNF), introduced by Raymond Boyce and Edgar Codd in 1974 (Codd, 1974), is a stricter reformulation of Third Normal Form. While 3NF permits a narrow class of functional dependencies that involve prime attributes, BCNF eliminates even those. In practice, the vast majority of relations that are in 3NF are also in BCNF; violations are rare and typically involve tables with multiple overlapping candidate keys. Nevertheless, BCNF is theoretically important and occasionally surfaces in real-world designs.

10.6.1 Definition and Relationship to 3NF

The contrast with 3NF is subtle but important. The 3NF definition contains an exception: a dependency X → Y is permitted even if X is not a superkey, provided that Y consists

entirely of prime attributes. BCNF removes this exception entirely. Under BCNF, the only functional dependencies allowed are those in which the left-hand side is a superkey, regardless of whether the right-hand side consists of prime or non-prime attributes.

The consequence is that BCNF is strictly stronger than 3NF: every relation in BCNF is in 3NF, but not every relation in 3NF is in BCNF. The gap between the two normal forms is closed only in relations where the two conditions coincide, which is the case for the overwhelming majority of practical database tables, particularly those with a single candidate key or with candidate keys that do not overlap.

> **Boyce-Codd Normal Form (BCNF)**
>
> A relation R is in Boyce-Codd Normal Form if and only if, for every non-trivial functional dependency $X \rightarrow Y$ in R, X is a superkey of R. A functional dependency $X \rightarrow Y$ is trivial if $Y \subseteq X$. (Codd, 1974)

10.6.2 When 3NF Is Not Enough: Identifying BCNF Violations

A BCNF violation can only occur in a relation that has two or more candidate keys, and those candidate keys often overlap, that is, they may share at least one attribute. If a relation has only one candidate key, 3NF and BCNF are equivalent, and no BCNF violation is possible beyond what 3NF would already catch.

The classic scenario involves a relation where a non-key attribute functionally determines part of the primary key. In such a case, the dependency from the non-key attribute to the prime attribute is not caught by 3NF (because 3NF's exception covers prime attributes on the right-hand side), but it is caught by BCNF (because the left-hand side is not a superkey).

University Database System— BCNF Scenario

Consider a scenario in which CLASS_SESSION is extended to record instructor assignments, and the following business rule holds: a professor teaches at most one session in each office per time slot, and each session has exactly one professor in one office at one time. Suppose a relation:

SESSION_INSTRUCTOR (Session_ID, Professor_ID, Office, Time_slot, Course_ID)

With two candidate keys: {Session_ID, Professor_ID} and {Office, Time_slot}. Suppose the dependency:

Professor_ID → Office

holds, meaning each professor always uses the same office. Professor_ID is not a superkey of the relation, yet it determines Office, as a prime attribute (part of the candidate key {Office, Time_slot}). This is a BCNF violation that 3NF does not catch — 3NF permits the dependency because Office is a prime attribute, but BCNF rejects it because Professor_ID is not a superkey.

Decomposition:

PROFESSOR_OFFICE (Professor_ID*, Office)

SESSION_SCHEDULE (Session_ID, Professor_ID*, Time_slot, Course_ID)

Note: BCNF decomposition does not always preserve all functional dependencies. In this case, the constraint that each Office+Time_slot maps to one session must be enforced by an application-level check rather than a foreign key constraint, which is a known trade-off of BCNF decomposition.

Company Database System — BCNF Scenario

Consider a simplified SUPPLIES relation extended to record warehouse assignments:

SUPPLY_ASSIGNMENT (Supplier_ID, Product_ID, Warehouse_ID, Assigned_manager_ID)

Suppose the business rule: each warehouse has exactly one manager, and each manager manages exactly one warehouse. This creates candidate keys {Supplier_ID, Product_ID, Warehouse_ID} and {Supplier_ID, Product_ID, Assigned_manager_ID}. The dependency:

Assigned_manager_ID → Warehouse_ID

holds because a manager uniquely identifies their warehouse. Assigned_manager_ID is not a superkey, yet it determines Warehouse_ID — a prime attribute. This violates BCNF.

Decomposition:

WAREHOUSE_MANAGER (Assigned_manager_ID*, Warehouse_ID*)

SUPPLY_ASSIGNMENT (Supplier_ID*, Product_ID*, Assigned_manager_ID*)

The Chapter 7 schema avoids this scenario by keeping manager assignment (Manages_Dept_ID) and warehouse assignment (Warehouse_ID) separate in the EMPLOYEE relation, which naturally prevents the overlapping candidate key that gives rise to BCNF violations.

10.6.3 Decomposition to BCNF

The general BCNF decomposition algorithm is as follows: given a relation R and a functional dependency X → Y that violates BCNF (because X is not a superkey of R), replace R with two relations: R1 containing X ∪ Y (the attributes of the violating dependency), and R2 containing X ∪ (all attributes of R not in Y) (the original relation minus the right-hand side of the violating dependency, retaining X as the join attribute). The process is applied recursively until all resulting relations are in BCNF.

An important limitation of BCNF decomposition is that it does not always produce a dependency-preserving decomposition. A **decomposition** is dependency-preserving if every original functional dependency can be checked within a single decomposed relation, without needing to join multiple relations. When BCNF decomposition violates this property, enforcing some constraints requires either application logic or expensive join-based checks. In such cases, designers sometimes accept 3NF (which always admits a lossless, dependency-preserving decomposition) as the target normal form rather than BCNF.

10.7 Beyond BCNF: A Brief Overview

The normal forms discussed in Sections 10.3 through 10.6 — 1NF, 2NF, 3NF, and BCNF — all address redundancy caused by functional dependencies: relationships where knowing the value of one attribute (or set of attributes) uniquely determines the value of another. They all address redundancy caused by functional dependencies: relationships where knowing the value of one attribute (or set of attributes) uniquely determines the value of another. However, two additional normal forms address more complex types of dependencies that functional dependency theory does not capture: multi-valued dependencies (addressed by Fourth Normal Form) and join dependencies (addressed by Fifth Normal Form).

Fourth Normal Form (4NF), introduced by Fagin in 1979 (Fagin, 1979), addresses multi-valued dependencies. A multi-valued dependency $X \twoheadrightarrow Y$ exists in a relation R when, for every pair of tuples that agree on X, their values for Y can be independently combined with any values in the remaining attributes of R. A relation in BCNF may still suffer from redundancy due to multi-valued dependencies when it contains two or more independent multi-valued facts about the same entity. 4NF requires that, for every non-trivial multi-valued dependency $X \twoheadrightarrow Y$, X must be a superkey.

Fifth Normal Form (5NF), also known as Project-Join Normal Form (PJNF) and introduced by Fagin in 1979, addresses join dependencies. A join dependency occurs when a relation can be decomposed into three or more relations, all of which must be joined to reconstruct the original, and where no two-way join is sufficient. 5NF is the highest practically defined normal form and handles cases that 4NF cannot.

In the context of the University and Company databases developed throughout this textbook, 4NF and 5NF violations are not present: the entities and relationships were carefully identified during ER modeling (Chapters 3–5), and multi-valued facts about the same entity were correctly separated into junction tables during the ER-to-relational mapping process (Chapter 7). For the majority of transactional database designs, BCNF represents a sufficient normalization target. 4NF and 5NF become relevant in specialized scenarios involving complex multi-valued relationships, particularly in scientific, logistics, or data warehousing contexts.

10.8 Normalization and ER-Based Design

A fundamental insight that emerges from working through the normal forms is the deep connection between formal normalization theory and the entity-relationship design methodology followed throughout this textbook. Databases built by carefully identifying entities, attributes, and relationships in a conceptual model, as described in Chapters 3 through 5, and translated into a relational schema in Chapter 7, naturally tend to be normalized. This section makes that connection explicit, explains why it holds, and uses it as a framework for verifying the correctness of the University and Company schemas.

10.8.1 Why Good ER Design Produces Normalized Schemas

The principles of good ER modeling and the requirements of the normal forms address the same underlying problem from different directions. Normalization theory diagnoses

structural defects in finished tables; ER modeling prevents those defects from being introduced in the first place.

Consider the correspondence between ER modeling decisions and normalization outcomes given in Table 10.3. The correspondence is not coincidental. Normalization theory was developed to characterize the structural properties that a good relational design should have. ER modeling methodology was developed to help designers produce such good designs. Both originate from Codd's foundational work on the relational model (Codd, 1970, 1971) and embody the same core principle: that each fact should be stored in exactly one place, in the table whose primary key directly identifies the entity that the fact describes.

Table 10.3: Correspondence between ER modeling decisions and normalization outcomes

ER Modeling Practice	Normalization Result
Each real-world entity gets its own table (STUDENT, COURSE, FACULTY are separate)	Prevents transitive dependencies that arise from mixing entity types — the primary mechanism for 3NF violations
Multi-valued attributes are mapped to separate relations (EMAIL, EMPLOYEE_PHONE)	Directly achieves 1NF by eliminating non-atomic values
Relationship attributes are stored in junction tables (ENROLLS_IN, STORED_IN)	Prevents partial dependencies by ensuring attributes of a relationship depend on the full composite key — the 2NF requirement
Primary keys are identified for every entity	Establishes the determinant structure from which all normal forms are evaluated
Weak entities include the owner's primary key as part of their composite key	Correctly captures the full key required for 2NF in dependent tables
Foreign keys reference primary keys of other tables	Avoids redundant storage of facts that belong to referenced entities, preventing transitive dependencies

10.8.2 Verifying Our University and Company Schemas

We now apply the normal form tests systematically to the University and Company schemas from Chapter 7, confirming that the ER-based design process has produced relations that satisfy BCNF. Rather than repeating the full analysis for every relation, the schemas are presented below; their compliance with BCNF follows from the normalization analysis conducted throughout Sections 10.3 through 10.6.

University Database System— Schema Verification

STUDENT (Student_ID, First_name, Last_name, Birth_date, Age, Street, City, Country, Zip_code, Faculty_ID*)

ENROLLS_IN (Student_ID*, Course_ID*, Enrollment_date, Grade)

PROFESSOR (Professor_ID, First_name, Last_name, Office, Email, Hire_date, Academic_title, Faculty_ID*, Heads_Faculty_ID*)

COURSE (Course_ID, Title, Credits, Description, Faculty_ID*, Professor_ID*)

Company Database System — Schema Verification

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Country, Zip_code, Department_ID*, Warehouse_ID*, Manages_Dept_ID*)

ORDER_LINE (Order_ID*, Line_number, Unit_price, Quantity, Discount, Product_ID*)

STORED_IN (Product_ID*, Warehouse_ID*, Quantity_on_hand, Reorder_level)

SUPPLIES (Product_ID*, Supplier_ID*, Supply_date, Unit_cost)

The conclusion from this verification is that the Chapter 7 schemas for both the University and Company systems satisfy **BCNF**. This outcome is not surprising: it is the expected result of a careful, entity-centered design process. The value of working through the verification is not to discover violations but to build confidence in the methodology and to develop the analytical skills needed to evaluate schemas encountered in professional practice, with schemas that may not have been developed with such care.

10.9 When to Denormalize

Normalization is not an end in itself. It is a tool for eliminating **redundancy** and the **anomalies** that redundancy causes, and its application is appropriate in the context of transactional databases where data is frequently inserted, updated, and deleted. However, normalization is not universally the optimal design strategy for every database workload.

In read-intensive analytical applications, such as data warehouses, reporting databases, and business intelligence systems, the join operations required to reconstruct information spread across multiple normalized tables can become performance bottlenecks. A query that combines STUDENT, FACULTY, ENROLLS_IN, COURSE, and PROFESSOR to produce an enrollment report must traverse five relations and perform four joins. In an online transaction processing (OLTP) system, this cost is acceptable because updates are frequent and correctness is paramount. In a data warehouse where millions of rows are aggregated in a single query, the same cost may be expensive.

Controlled **denormalization** is the intentional introduction of redundancy for performance purposes, which is a legitimate and sometimes necessary design decision in such contexts. Common denormalization techniques include:

- **Storing derived attributes**: Pre-computing and storing values that could be calculated from other data (e.g., storing Total_amount on ORDER rather than recomputing it by summing ORDER_LINE rows on every query).
- **Combining frequently joined tables**: Merging two tables that are almost always accessed together into a single wider table to eliminate the join cost.
- **Adding redundant foreign key copies**: Storing an attribute in two tables to allow common queries to avoid a join (e.g., adding Faculty_name to STUDENT to avoid joining FACULTY for display purposes).

The critical principle is that denormalization must be *intentional* and *documented*. An application that knowingly stores redundant data must implement compensating mechanisms, such as triggers, application logic, or scheduled reconciliation processes, to

maintain consistency. Undocumented denormalization that originated as a design shortcut is not intentional denormalization; it is a design flaw.

For the University and Company databases in this textbook, full normalization to BCNF is appropriate. These are transactional systems where correctness, consistency, and the absence of anomalies are the primary concerns. Any denormalization decisions for reporting or analytical purposes should be implemented as separate read-optimized views or materialized views, leaving the core transactional schema fully normalized.

10.10 Summary

This chapter completes the logical design process by introducing normalization, with the formal methodology for detecting and correcting structural defects in relational schemas. The chapter develops the theory systematically, from the foundational concept of functional dependency through the four major normal forms, and concludes by connecting normalization theory to the ER-based design practice used throughout this textbook.

Section 10.1 established the need for normalization through the three categories of anomaly that arise from mixing independent facts in a single relation: update anomalies, insertion anomalies, and deletion anomalies. The root cause of all three anomaly types is redundancy — storing the same fact in multiple places. Section 10.2 introduced functional dependency as the theoretical foundation of normalization. A functional dependency $X \rightarrow Y$ holds when each value of X is associated with exactly one value of Y across all valid states of a relation. Three categories of dependency were distinguished: full functional dependencies, partial functional dependencies, and transitive functional dependencies. The section also revisited the key types from Chapter 7, such as superkeys, candidate keys, primary keys, and alternate keys, through the lens of functional dependency theory.

The First Normal Form requires every attribute value to be atomic and prohibits multi-valued attributes and repeating groups within a single column. The University system's EMAIL relation and the Company system's EMPLOYEE_PHONE, CUSTOMER_PHONE, and SUPPLIER_PHONE relations were presented as the correct treatment of multi-valued attributes, already implemented correctly in Chapter 7 through Step 6 of the ER-to-relational mapping algorithm.

The Second Normal Form, which extends 1NF by requiring that every non-prime attribute depend fully on the entire primary key, is relevant only for tables with composite keys. A denormalized ENROLLS_IN table (University) and a denormalized ORDER_LINE table (Company) were used to illustrate partial dependencies and their decomposition into separate, correctly structured relations.

The Third Normal Form extends 2NF by prohibiting transitive dependencies — chains where a non-prime attribute depends on the primary key through another non-prime attribute. The classic example of embedding faculty attributes in the STUDENT table (University) and department attributes in the EMPLOYEE table (Company) demonstrated how transitive dependencies produce the same anomalies as partial dependencies, and how decomposition into separate FACULTY and DEPARTMENT tables resolves them.

The Boyce-Codd Normal Form is a stricter reformulation of 3NF that requires every determinant in every functional dependency to be a superkey. BCNF violations are rare in practice but arise in tables with multiple overlapping candidate keys. The section illustrated BCNF scenarios for both the University and Company systems and noted the important caveat that BCNF decomposition does not always preserve all functional dependencies, sometimes requiring application-level constraint enforcement.

Section 10.7 briefly introduced Fourth Normal Form (addressing multi-valued dependencies) and Fifth Normal Form (addressing join dependencies), establishing that these higher normal forms are not required for the University and Company schemas, but are relevant in specialized scenarios.

Section 10.8 demonstrated that databases designed through proper ER modeling naturally satisfy the normal forms. A review of representative relations from the University and Company schemas confirmed BCNF compliance, reinforcing the insight that normalization serves as verification of good design rather than as a substitute for it.

Finally, controlled denormalization — the intentional introduction of redundancy for analytical performance purposes — was presented as a legitimate design trade-off in read-intensive workloads, provided it is documented and accompanied by mechanisms to maintain consistency.

Review Questions

10.1. What is a functional dependency? Explain the difference between a determinant and a dependent attribute.

10.2. Define full functional dependency and partial functional dependency. In which type of relation can a partial dependency arise? Explain why.

10.3. What is a transitive functional dependency? Provide an example from the University Management System or the Company Database System in which a transitive dependency would exist if the schema were denormalized.

10.4. State the requirements for First Normal Form. Identify two real-world scenarios from the University or Company system in which 1NF would be violated if multi-valued attributes were stored in a single column, and describe the correct resolution.

10.5. Define Second Normal Form. Why can a relation with a single-column primary key never violate 2NF? What specific structural condition in a relation signals that a 2NF check is necessary?

10.6. Explain how a partial dependency in a composite-key relation causes each of the three anomaly types (update, insertion, deletion). Use the ENROLLS_IN or ORDER_LINE relation as the basis for your explanation.

10.7. Define Third Normal Form. What is the difference between 3NF and 2NF in terms of the type of dependency each normal form addresses? How does the test for a 3NF violation differ from the test for a 2NF violation?

10.8. Explain the difference between 3NF and Boyce-Codd Normal Form. Under what conditions is a relation guaranteed to be in BCNF if it is already in 3NF? What structural feature of a relation can cause it to be in 3NF but not in BCNF?

10.9. Why do databases designed through a systematic ER modeling process tend to satisfy the higher normal forms without explicit normalization steps? Describe at least three specific ER modeling practices and the normal form requirement each practice naturally satisfies.

10.10. What is denormalization? Under what application workload characteristics might controlled denormalization be justified? What obligations does a designer take on when intentionally introducing redundancy into a normalized schema?

Exercises

10.1. Identifying Functional Dependencies

Consider the following relation for a Library System: LOAN (Loan_ID, Member_ID, Member_name, Book_ISBN, Book_title, Author, Loan_date, Due_date, Branch_ID, Branch_name, Branch_location).

a) Identify all functional dependencies that hold in this relation, assuming standard library business rules.
b) Identify all candidate keys.
c) Identify all prime and non-prime attributes.
d) For each non-prime attribute, state whether its dependency on the primary key is full, partial, or transitive.

10.2. Functional Dependencies in the University System

Refer to the CLASS_SESSION relation from the University schema: CLASS_SESSION (Course_ID*, Session_number, Year, Room, Semester, Time, Day, Duration).

a) State the primary key and write all functional dependencies that hold in CLASS_SESSION.
b) Explain why Duration depends on the full composite key rather than on any single component.
c) If Semester were additionally stored in the COURSE table, would a transitive dependency exist in CLASS_SESSION? Justify your answer.
d) Verify that CLASS_SESSION satisfies 3NF.

10.3. Full vs. Partial Functional Dependency

A university administrator proposes storing enrollment and course catalogue data in a single table: COURSE_ENROLLMENT(Student_ID, Course_ID, First_name, Last_name, Course_title, Credits, Faculty_name, Enrollment_date, Grade).

a) Identify the composite primary key for this table.
b) List all partial functional dependencies present.
c) List all full functional dependencies on the composite key.
d) Decompose this table into 2NF, showing all resulting relations with their primary keys.

10.4. Transitive Dependencies in the Company System

Consider the following denormalized table for the Company system: EMPLOYEE_DEPT(Employee_ID, First_name, Last_name, Email, Salary, Department_ID, Dept_name, Dept_location, Dept_budget, Warehouse_ID, Warehouse_location, Warehouse_capacity).

a) Identify all transitive functional dependencies in this table.
b) For each transitive dependency, write the full dependency chain (PK → intermediate → dependent attribute).
c) Decompose this table into 3NF, showing all resulting relations with their primary and foreign keys.
d) Verify that each relation in your decomposition satisfies 2NF.

10.5. Applying 1NF to the Company System

A legacy version of the Company database stored the SUPPLIER entity with the following schema: SUPPLIER_LEGACY(Supplier_ID, Supplier_name, Contact_person, Phone_numbers, Product_IDs, Email_addresses). The Phone_numbers column contains comma-separated phone numbers; Product_IDs contains a list of supplied product identifiers; Email_addresses contains multiple email addresses separated by semicolons.

a) Identify all 1NF violations in SUPPLIER_LEGACY.
b) Apply Step 6 of the ER-to-relational mapping algorithm to resolve each violation.
c) Write the complete set of 1NF-compliant relations with appropriate primary keys.
d) Compare your result with the Chapter 7 Company schema. What additional relations would be needed?

10.6. 2NF Decomposition — University ATTENDS Relation

Consider an extended version of the ATTENDS relation: ATTENDS_ALT(Student_ID*, Course_ID*, Session_number*, Year*, Attendance, Student_first_name, Student_last_name, Course_title, Session_room, Session_day).

a) Identify the composite primary key.
b) For each non-prime attribute, determine whether its dependency on the primary key is full, partial, or neither.
c) List all partial dependencies present.
d) Decompose ATTENDS_ALT into 2NF, writing each resulting relation with its primary key and foreign keys.
e) Verify that each resulting relation is in 2NF.

10.7. 3NF Decomposition — Combined Order Table

A poorly designed order management table contains: ORDER_FULL(Order_ID, Order_date, Status, Customer_ID, Customer_first_name, Customer_email, Customer_city, Employee_ID, Employee_last_name, Employee_dept_name, Required_date, Shipped_date).

a) Identify the primary key and all candidate keys.
b) Identify all transitive dependencies. Write the full chain for each.
c) Identify any partial dependencies.
d) Decompose ORDER_FULL into 3NF.

e) Confirm that your decomposition matches the Chapter 7 Company schema structure.

10.8. BCNF Analysis

Consider a scheduling relation for a tutoring center: TUTORING(Tutor_ID, Student_ID, Subject, Office, Time_slot). Business rules: each tutor teaches exactly one subject; each office is dedicated to one subject; a tutor may see multiple students but only one at a time; a student may have only one tutor per subject.

a) Identify all candidate keys.
b) Write all functional dependencies that hold.
c) Determine whether this relation is in 3NF. Justify your answer.
d) Determine whether this relation is in BCNF. If not, identify the BCNF violation.
e) If a BCNF violation exists, decompose the relation into BCNF and note whether dependency preservation is maintained.

10.9. Normalization of a Product Inventory Table

A company maintains the following inventory table: INVENTORY(Product_ID, Product_name, Category, Supplier_ID, Supplier_name, Supplier_city, Warehouse_ID, Warehouse_location, Warehouse_capacity, Quantity_on_hand, Reorder_level, Unit_price).

a) Identify all functional dependencies.
b) Identify all primary and candidate keys.
c) Identify all 1NF, 2NF, and 3NF violations.
d) Decompose INVENTORY progressively: first to 1NF (if needed), then to 2NF, then to 3NF.
e) Show the final set of normalized relations and compare them to the relevant Chapter 7 Company tables.

10.10. Normalization of a Course Registration System

A university registration system consolidates data into: REGISTRATION (Reg_ID, Student_ID, Student_name, Student_email, Faculty_ID, Faculty_name, Course_ID, Course_title, Credits, Professor_ID, Professor_name, Professor_office, Enrollment_date, Grade, Semester, Year).

a) Identify the primary key and all functional dependencies.
b) Classify each dependency as full, partial, or transitive.
c) Decompose REGISTRATION progressively to 3NF.
d) Write the complete set of resulting relations with all keys.
e) Verify that each resulting relation satisfies BCNF.

10.11. Anomaly Analysis

Use the STUDENT_ENROLLMENT_FLAT table from Section 10.1 (reproduced below for reference) to answer the following questions. STUDENT_ENROLLMENT_FLAT(Student_ID, First_name, Last_name, Course_ID, Course_title, Credits, Faculty_name, Professor_name).

a) Describe a specific update anomaly scenario using the actual data values in Table 10.1.
b) Describe a specific insertion anomaly: what legitimate fact cannot be recorded in this table, and why?

c) Describe a specific deletion anomaly: what information is unintentionally lost when a specific student's records are deleted?
d) For each anomaly, explain which functional dependency — partial or transitive — is its root cause.
e) Show the normalized set of relations that eliminates all three anomaly types.

10.12. Dependency Preservation in BCNF

Consider a relation EXAM_SCHEDULE(Student_ID, Course_ID, Exam_office, Proctor_ID) where: each exam office hosts exactly one course at a time; each proctor supervises exactly one office; a student takes each exam in a specific office with a specific proctor.

a) Identify all candidate keys for EXAM_SCHEDULE.
b) List all functional dependencies.
c) Is EXAM_SCHEDULE in 3NF? In BCNF? Justify your answers.
d) If a BCNF decomposition is required, show it and identify which functional dependency, if any, is no longer directly enforceable.
e) Propose an alternative design that avoids the BCNF violation without decomposition.

10.13. Normalization in the Context of ER Modeling

Recall from Chapters 3–5 how the PROFESSOR entity was identified during the conceptual design of the University system. The PROFESSOR relation in Chapter 7 is: PROFESSOR (Professor_ID, First_name, Last_name, Office, Email, Hire_date, Academic_title, Faculty_ID*, Heads_Faculty_ID*).

a) Verify that PROFESSOR satisfies 1NF, 2NF, 3NF, and BCNF. For each normal form, state the test and the result.
b) Identify the candidate keys of PROFESSOR.
c) Explain how the ER modeling decision to make Faculty a separate entity (rather than an attribute of PROFESSOR) prevented a specific 3NF violation.
d) If PROFESSOR had an additional attribute Research_area_budget that depended on Faculty_ID → Research_area_budget, what normal form would be violated, and how would you resolve it?

10.14. Progressive Normalization — Healthcare Scenario

A healthcare database contains the following unnormalized table: PATIENT_RECORD(Patient_ID, Patient_name, Patient_DOB, Doctor_ID, Doctor_name, Doctor_specialty, Department_ID, Dept_name, Dept_floor, Diagnosis_code, Diagnosis_description, Visit_date, Prescription, Insurance_ID, Insurance_provider, Insurance_coverage_limit).

a) Identify all functional dependencies in PATIENT_RECORD.
b) Identify all 1NF violations (if any).
c) Decompose to 2NF, showing all resulting relations.
d) Decompose your 2NF result further to 3NF, showing all resulting relations with primary keys and foreign keys.
e) Verify BCNF compliance of each resulting relation.
f) Draw the dependency diagram showing the final normalized schema.

10.15. Verifying the ASSIGNMENT Relation

The University schema includes: ASSIGNMENT(Professor_ID*, Course_ID*, ISBN*). This is an N-ary relationship table (Step 7 of the mapping algorithm) linking professors, courses, and textbooks.

a) State the primary key of ASSIGNMENT.
b) Identify all functional dependencies within ASSIGNMENT.
c) Verify that ASSIGNMENT satisfies 2NF.
d) Verify that ASSIGNMENT satisfies 3NF.
e) Verify BCNF. Is there any functional dependency that could cause a BCNF violation in a future extension of this table?

10.16. Denormalization Decision Analysis

The university's business intelligence team requires a monthly report that lists, for each course: course title, credits, faculty name, professor full name, number of enrolled students, and average grade. Currently, producing this report requires joining COURSE, FACULTY, PROFESSOR, and ENROLLS_IN.

a) Write a description of the join operations required to produce the report from the normalized schema.
b) Propose a denormalized structure that would allow the report to be produced from a single table scan.
c) Identify all redundancies introduced by the denormalization.
d) Describe the anomalies that the denormalized structure is susceptible to.
e) Propose mechanisms (triggers, views, or scheduled processes) to maintain consistency in the denormalized structure.
f) Explain why this denormalization should be implemented as a separate reporting table rather than by modifying the transactional schema.

10.17. Normalization of an E-Commerce Order System

An e-commerce platform uses the following flat table: ORDER_SUMMARY(Order_ID, Order_date, Customer_ID, Customer_name, Customer_email, Customer_city, Product_ID, Product_name, Category, Unit_price, Quantity, Supplier_ID, Supplier_name, Warehouse_ID, Warehouse_city, Warehouse_capacity, Discount, Subtotal).

a) Identify all functional dependencies.
b) Identify the primary key and any candidate keys.
c) Classify all non-prime attributes by their dependency type (full, partial, transitive).
d) Decompose ORDER_SUMMARY completely to 3NF.
e) Compare your result to the Company schema in Chapter 7 and identify any differences.
f) Verify that each relation in your decomposition satisfies BCNF.

10.18. Comprehensive Normalization — Academic Conference System

Design and normalize a schema for an academic conference management system with the following requirements: "Conferences have unique IDs, names, venues, and dates. Authors submit papers. Each paper has a title, abstract, and track. Papers are reviewed by program committee members. Each review has a score, recommendation, and review date. A

committee member can review multiple papers; a paper receives multiple reviews. Authors have affiliations (university or company name). Presentations are scheduled in specific offices at specific times."

a) Identify all entities and propose an initial flat schema capturing all attributes.
b) Identify all functional dependencies in the flat schema.
c) Decompose to 1NF, then 2NF, then 3NF, showing the result after each step.
d) Verify BCNF compliance for each final relation.
e) Write the complete normalized relational schema with all primary keys, foreign keys, and candidate keys.

For each relation, state the business rule that is captured by its primary key constraint.

References and Recommended Readings

Primary References

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM, 13*(6), 377-387.

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of Database Systems* (7th ed.). Pearson.

Hristoski, I., & Dimovski, T. (2022). *Dizajn na relaciski bazi na podatoci, Tom 1* [Design of Relational Databases, Volume 1]. University "St. Kliment Ohridski", Bitola. (In Macedonian)

Connolly, T. M., & Begg, C. E. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson.

Codd, E. F. (1971). Further normalization of the database relational model. In R. Rustin (Ed.), *Database Systems: Courant Computer Science Symposia Series 6* (pp. 33–64). Prentice-Hall.

Historical and Theoretical Foundations

Fagin, R. (1977). Multivalued dependencies and a new normal form for relational databases. *ACM Transactions on Database Systems, 2*(3), 262–278.

Fagin, R. (1979). Normal forms and relational database operators. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 153–160.

Codd, E. F. (1974). Recent investigations into relational database systems. *IBM Research Report RJ1385*

Chapter 11

11. Integrated Case Studies

11.1 Introduction

This chapter consolidates the complete relational schemas and sample data for the two case studies used throughout this book: the University Management System and the Company/Business Database. Rather than referring to partial schemas scattered across earlier chapters, this chapter serves as a single authoritative reference for all relations, attributes, primary and foreign keys, and instance data. Students and instructors are encouraged to return to this chapter whenever a specific relation definition or a concrete example data is needed.

The schemas presented here reflect the basic Entity-Relationship model as mapped to the relational model in Chapter 7. The notation follows the conventions established in that chapter: relation names appear in ALL_CAPS, primary key attributes are underlined, and foreign key attributes are marked with an asterisk (*). Attributes that serve as both primary and foreign keys are both underlined and asterisked. The design rationale for each schema — entity definitions, relationship types, cardinality constraints, and mapping decisions — is discussed in Chapters 3 through 7.

Note: The schemas in this chapter are derived from the basic ER conceptual model. The Extended Entity-Relationship (EER) specializations developed in Chapter 5 — including the PERSON hierarchy for the University system and the EMPLOYEE dual specialization for the Company system — introduce additional subclass relations that are not included here. The EER-extended schemas and their corresponding datasets are planned for a future edition of this book.

11.2 University Management System

The University Management System models the academic operations of a university: faculties, professors, courses, students, enrolments, class sessions, textbooks, and their interrelationships. The system supports course scheduling, student grade tracking, class attendance monitoring, and textbook assignment management.

11.2.1 Relational Schema

The complete relational schema for the University Management System consists of eleven relations. Primary key attributes are underlined; foreign key attributes carry an

asterisk (*); attributes serving as both primary and foreign keys are underlined and asterisked.

FACULTY (Faculty_ID, Name, Budget, Location**)**

STUDENT (Student_ID, First_name, Last_name, Birth_date, Age, Street, City, Country, Zip_code, Faculty_ID***)**

EMAIL (Student_ID*, Email**)**

PROFESSOR (Professor_ID, First_name, Last_name, Office, Email, Hire_date, Academic_title, Faculty_ID*, Heads_Faculty_ID***)**

COURSE (Course_ID, Title, Credits, Description, Faculty_ID*, Professor_ID***)**

PREREQUISITE (Course_ID*, Prerequisite_Course_ID***)**

ENROLLS_IN (Student_ID*, Course_ID*, Enrollment_date, Grade**)**

CLASS_SESSION (Course_ID*, Session_number, Year, Room, Semester, Time, Day, Duration**)**

ATTENDS (Student_ID*, Course_ID*, Session_number*, Year*, Attendance**)**

TEXTBOOK (ISBN, Title, Author, Edition, Publisher, Available_copies**)**

ASSIGNMENT (Professor_ID*, Course_ID*, ISBN***)**

11.2.2 Sample Data

The following tables present the sample data instances for each relation in the University Management System. These instances are used as running examples in Chapters 4 through 10.

Table 11.1: FACULTY — Five academic faculties of the university.

Faculty_ID	*Name*	*Budget*	*Location*
F001	Computer Science	1,500,000	Main Campus
F002	Electrical Engineering	1,200,000	East Wing
F003	Economy & Tourism	900,000	West Wing
F004	Bioengineering	1,100,000	North Building
F005	Humanities	800,000	South Building

Table 11.2: STUDENT — Six enrolled students with personal and faculty affiliation data.

Student_ ID	*First_ name*	*Last_name*	*Birth_date*	*Age*	*Street*	*City*	*Country*	*Zip_co de*	*Faculty _ID*
S001	Sarah	Smith	25.08.1995	30	Jane Sandanski 32	Ohrid	Macedo nia	6000	F001
S002	Bojan	Jovansen	20.11.2002	23	Finkgatan 45	Malmö	Sweden	214 59	F002
S003	Kosta	Dojcinoski	10.03.2004	21	Asamati 78	Resen	Macedo nia	7310	F004
S004	David	Boskovski	25.01.2001	23	Ilindeska 65	Skopje	Macedo nia	1000	F002

Student_ID	First_name	Last_name	Birth_date	Age	Street	City	Country	Zip_code	Faculty_ID
S005	Elena	Mitan	01.07.2003	22	Turisticka 22	Ohrid	Macedonia	6000	F005
S006	Anica	Stojnasen	15.05.2003	22	Balladgatan 51	Malmö	Sweden	215 71	F001

Table 11.3: EMAIL — Multivalued email addresses for students (one student may have multiple entries).

Student_ID	Email
S001	sarah.smith@university.edu
S001	sarah.smith@gmail.com
S002	bojan.jovansen@university.edu
S002	bojan.j@gmail.com
S003	kosta.dojcinoski@university.edu
S004	david.boskovski@university.edu
S005	elena.mitan@gmail.com
S005	elena.mitan@university.edu
S006	anica.stojansen@university.edu
S006	anica.s@gmail.com

Table 11.4: PROFESSOR — Six professors with academic titles, office locations, and faculty affiliations.

Professor_ID	First_name	Last_name	Office	Email	Hire_date	Academic_title	Faculty_ID	Heads_Faculty_ID
P001	Kliment	Ohridski	A201	kliment.ohridski@university.edu	01.08.2007	Full Professor	F001	NULL
P002	Naum	Ohridski	B305	naum.ohridski@university.edu	15.01.2004	Associate Professor	F002	F002
P003	Kiril	Prosvetitel	C102	kiril.prosvetitel@university.edu	25.09.2001	Full Professor	F004	NULL
P004	Metodij	Prosvetitel	D404	metodij.prosvetitel@university.edu	13.09.2002	Full Professor	F001	F001
P005	Blaze	Koneski	E203	blaze.koneski@university.edu	19.10.2012	Assistant Professor	F005	F005
P006	Richard	Johnson	E206	richard.johnson@university.edu	02.09.2022	Associate Professor	F001	NULL

Table 11.5: COURSE — Eleven courses offered across five faculties.

Course_ID	Title	Credits	Description	Faculty_ID	Professor_ID
CS101	Intro to Programming	5	Fundamentals of coding	F001	P001

Course_ID	*Title*	*Credits*	*Description*	*Faculty_ID*	*Professor_ID*
CS201	Advanced Programming	6	Object-oriented programming and data structures	F001	P001
CS303	Database Systems	6	Database fundamentals, design, SQL, and practical applications	F001	P006
EE201	Circuit Analysis	6	Electrical circuit theory	F002	P002
EE301	Advanced Circuit Analysis	7	Complex circuit analysis and design	F002	P002
MA301	Linear Algebra	6	Vector spaces and matrices	F004	P003
MA401	Matrix Theory and Advanced Vector Spaces	7	Matrix theory and advanced vector spaces	F004	P003
PH101	Physics	8	Mechanics and thermodynamics	F002	P004
PH201	Advanced Physics	9	Quantum mechanics and advanced thermodynamics	F002	P004
HU201	Academic Writing	5	Academic writing — scientific approach	F005	P005
HU301	Advanced Academic Writing	6	Research methodology and scientific writing	F005	P005

Table 11.6: PREREQUISITE — Prerequisite relationships between courses.

Course_ID	*Prerequisite_Course_ID*
CS201	CS101
CS303	CS101
EE301	EE201
MA401	MA301
PH201	PH101
HU301	HU201

Table 11.7: ENROLLS_IN — Student enrolments in courses.

Student_ID	*Course_ID*	*Enrollment_date*	*Grade*
S001	CS303	01.10.2025	9
S002	EE201	01.10.2025	7
S002	EE301	01.10.2025	5
S003	MA301	01.10.2025	8
S003	MA401	01.10.2025	5
S004	PH101	01.10.2025	6
S004	PH201	01.10.2025	8
S005	HU201	01.10.2025	10

Student_ID	*Course_ID*	*Enrollment_date*	*Grade*
S005	HU301	01.10.2025	7
S006	CS101	01.10.2025	10
S006	CS201	01.10.2025	8
S006	CS303	01.10.2025	6

Table 11.8: CLASS_SESSION — Scheduled class sessions for each course, identified by course, session number, and year.

Course_ID	*Session_number*	*Year*	*Room*	*Semester*	*Time*	*Day*	*Duration (min)*
CS101	1	2025	L101	Fall 2025	09:00	Monday	90
CS101	2	2025	L101	Fall 2025	11:00	Wednesday	30
CS201	1	2025	L102	Fall 2025	14:00	Monday	90
CS201	2	2025	L102	Fall 2025	14:00	Monday	90
CS303	5	2025	L103	Fall 2025	11:00	Wednesday	120
EE201	1	2025	L206	Fall 2025	15:00	Tuesday	90
EE301	1	2025	L205	Fall 2025	10:00	Tuesday	90
MA301	1	2025	L302	Fall 2025	11:00	Wednesday	90
MA401	1	2025	L303	Fall 2025	16:00	Wednesday	90
MA401	2	2025	L303	Fall 2025	16:00	Friday	30
PH101	1	2025	L401	Fall 2025	13:00	Thursday	120
PH201	1	2025	L102	Fall 2025	17:00	Thursday	120
HU201	1	2025	L101	Spring 2025	09:00	Friday	60
HU201	2	2025	L101	Spring 2025	10:30	Friday	60
HU301	3	2025	L102	Spring 2025	11:00	Friday	60

Table 11.9: ATTENDS — Student attendance records for individual class sessions.

Student_ID	*Course_ID*	*Session_number*	*Year*	*Attendance*
S001	CS303	5	2025	Present
S002	EE201	1	2025	Absent
S002	EE301	1	2025	Present
S003	MA301	1	2025	Present
S003	MA401	1	2025	Absent
S003	MA401	2	2025	Present
S004	PH101	1	2025	Absent
S004	PH201	1	2025	Absent
S005	HU201	1	2025	Present
S005	HU201	2	2025	Absent
S005	HU301	3	2025	Present
S006	CS101	1	2025	Absent
S006	CS101	2	2025	Present

Student_ID	*Course_ID*	*Session_number*	*Year*	*Attendance*
S006	CS201	1	2025	Absent
S006	CS201	2	2025	Present

Table 11.10: TEXTBOOK — Textbooks available in the university library, assigned to courses.

ISBN	*Title*	*Author*	*Edition*	*Publisher*	*Available_copies*
978-0134093413	Introduction to Programming by Examples	Atanas Hristov, Naum Tuntev & Ninoslav Marina	1st	UIST St. Paul Apostle – Ohrid	100
978-2134093493	Object-Oriented Programming with Examples	Eric Matthes	2nd	No Starch Press	90
978-0834080185	Fundamentals of Database Systems	Ramez Elmasri & Shamkant B. Navathe	2nd	Pearson Education	70
978-0131498642	Electric Circuits	James W. Nilsson	10th	Pearson	72
978-6321286413	Complex Circuit Analysis and Design	Michael T. Goodrich	3rd	Wiley	25
978-0471754471	Linear Algebra Done Right	Sheldon Axler	3rd	Springer	35
978-0571754171	Matrix Theory and Advanced Vector Spaces	Michael T. Navathe	2nd	Pearson Education	15
978-0134080185	Physics for Scientists and Engineers	Raymond A. Serway	9th	Cengage Learning	18
978-7134082853	Quantum Mechanics and Advanced Thermodynamics	James A. Goodrich	6th	Wiley	NULL
978-0321286433	Academic Concepts	Shamkant W. Mathes	4th	Springer	94
978-7821286423	Research Methodology and Scientific Writing	Naum Nilsson	4th	Pearson	11
978-0675207379	Academic Writing	Wilson, J. R.	6th	Merrill Pub Co	NULL

Table 11.11: ASSIGNMENT — Textbook assignments: which professor uses which textbook for which course.

Professor_ID	*Course_ID*	*ISBN*
P001	CS101	978-0134093413
P001	CS201	978-2134093493
P006	CS303	978-0834080185
P002	EE201	978-0131498642
P002	EE301	978-6321286413
P003	MA301	978-0471754471
P003	MA401	978-0571754171

Professor_ID	*Course_ID*	*ISBN*
P004	PH101	978-0134080185
P004	PH201	978-7134082853
P005	HU201	978-0321286433
P005	HU301	978-7821286423

11.3 Company/Business Database

The Company/Business Database models the operations of a trading company: departments, warehouses, employees, customers, orders, products, and suppliers. The system supports order processing, inventory management, employee assignment, and supplier relationship tracking. All products in the sample dataset represent traditional Macedonian goods, reflecting the cultural context of the case study introduced in Chapter 4.

11.3.1 Relational Schema

The complete relational schema for the Company/Business Database comprises 13 relations. As in Section 11.2.1, primary key attributes are underlined; foreign key attributes carry an asterisk (*); dual primary–foreign key attributes are underlined and asterisked.

DEPARTMENT (Department_ID, Dept_name, Location, Budget)

WAREHOUSE (Warehouse_ID, Location, Capacity, Department_ID*)

EMPLOYEE (Employee_ID, First_name, Last_name, Email, Salary, Hire_date, Street, City, Country, Zip_code, Department_ID*, Warehouse_ID*, Manages_Dept_ID*)

CUSTOMER (Customer_ID, First_name, Last_name, Email, Join_date, Street, City, Country, Zip_code)

PRODUCT (Product_ID, Product_name, Description, Category, Price, Weight)

SUPPLIER (Supplier_ID, Supplier_name, First_name, Last_name, Email, Street, City, Country, Zip_code)

ORDER (Order_ID, Status, Order_date, Required_date, Shipped_date, Employee_ID*, Customer_ID*)

ORDER_LINE (Order_ID*, Line_number, Unit_price, Quantity, Discount, Product_ID*)

EMPLOYEE_PHONE (Employee_ID*, Phone_number)

CUSTOMER_PHONE (Customer_ID*, Phone_number)

SUPPLIER_PHONE (Supplier_ID*, Phone_number)

SUPPLIES (Product_ID*, Supplier_ID*, Supply_date, Unit_cost)

STORED_IN (Product_ID*, Warehouse_ID*, Quantity_on_hand, Reorder_level)

11.3.2 Sample Data

The following tables present the sample data for each relation in the Company/Business Database. These instances are referenced in Chapters 4 through 10.

Table 11.12: DEPARTMENT — Three company departments across Macedonian cities.

Department_ID	*Dept_name*	*Location*	*Budget*
D1	Sales	Skopje	500,000
D2	Marketing	Bitola	300,000
D3	Human Resources	Ohrid	200,000

Table 11.13: WAREHOUSE — Four warehouses overseen by their respective departments.

Warehouse_ID	*Location*	*Capacity*	*Department_ID*
W1	Skopje	10000	D1
W2	Bitola	7500	D2
W3	Ohrid	5000	D3
W4	Tetovo	1000	D2

Table 11.14a: EMPLOYEE — Personal and employment details for eight employees.

Employee_ID	*First_name*	*Last_name*	*Email*	*Salary*	*Hire_date*
E1	Zoran	Pejovski	zoran.pejovski@email.com	60000	15.01.2020
E2	Nikola	Delcevski	nikola.delcevski@email.com	55000	20.03.2021
E3	Marija	Naumoska	marija.naumoska@email.com	50000	10.05.2022
E4	Anita	Toshevska	anita.toshevska@email.com	45000	22.07.2023
E5	Naum	Stojanoski	naum.stojanoski@email.com	40000	01.09.2020
E6	Hristina	Miloseska	hristina.miloseska@email.com	35000	12.11.2021
E7	Dimitar	Tocko	dimitar.tocko@email.com	30000	28.02.2022
E8	Biljana	Osmani	biljana.osmani@email.com	25000	18.04.2023

Table 11.14b: EMPLOYEE — Location and department details (continued from Table 11.14a; Employee_ID links the two parts).

Employee_ID	*Street*	*City*	*Country*	*Zip_code*	*Department_ID*	*Warehouse_ID*	*Manages_Dept_ID*
E1	Partizanska 106	Skopje	Macedonia	1000	D1	W1	D1
E2	Ilindenska 208	Skopje	Macedonia	1000	D1	W1	NULL
E3	Marshal Tito 38	Bitola	Macedonia	7000	D2	W2	D2

Employee_ID	*Street*	*City*	*Country*	*Zip_code*	*Department_ID*	*Warehouse_ID*	*Manages_Dept_ID*
E4	Goce Delchev 42	Bitola	Macedonia	7000	D2	W2	NULL
E5	Jane Sandanski 58	Ohrid	Macedonia	6000	D3	W3	D3
E6	Dame Gruev 67	Ohrid	Macedonia	6000	D3	W3	NULL
E7	Pitu Guli 79	Prilep	Macedonia	7500	D1	W1	NULL
E8	Nikola Karev 87	Tetovo	Macedonia	1200	D2	W2	NULL

Table 11.15: CUSTOMER — Seven customers with contact and address information.

Customer_ID	*First_name*	*Last_name*	*Email*	*Join_date*	*Street*	*City*	*Country*	*Zip_code*
C1	Ana	Ivanovska	ana.ivanovska@email.com	10.01.2022	Partizanski Odredi 10	Skopje	Macedonia	1000
C2	Petar	Petrovski	petar.petrovski@email.com	15.03.2022	Ilindenska 20	Bitola	Macedonia	7000
C3	Marija	Georgievska	marija.georgievska@email.com	20.05.2022	Goce Delchev 30	Ohrid	Macedonia	6000
C4	Nikola	Nikolovski	nikola.nikolovski@email.com	25.07.2022	Jane Sandanski 40	Prilep	Macedonia	7500
C5	Elena	Trajkovska	elena.trajkovska@email.com	30.09.2022	Dame Gruev 50	Tetovo	Macedonia	1200
C6	Goran	Stojanovski	goran.stojanovski@email.com	05.01.2023	Pitu Guli 60	Kumanovo	Macedonia	1300
C7	Ivana	Jovanovska	ivana.jovanovska@email.com	12.03.2023	Nikola Karev 70	Veles	Macedonia	1400

Table 11.16: PRODUCT — Twelve traditional Macedonian products with pricing and category data.

Product_ID	*Product_name*	*Description*	*Category*	*Price*	*Weight (kg)*
1	Ajvar	Traditional Macedonian spread	Food	372	0.5
2	Tavce Gravce	Baked beans dish	Food	434	0.7
3	Rakija	Fruit brandy	Beverage	682	1
4	Mastika	Mastic-flavored liqueur	Beverage	930	1
5	Sirenje	White cheese	Dairy	465	0.4
6	Kajmak	Creamy dairy product	Dairy	620	0.3
7	Ohrid Pearl	Jewelry	Accessories	9300	0.1
8	Macedonian Ruby	Gemstone	Accessories	31000	0.05

Product_ID	*Product_name*	*Description*	*Category*	*Price*	*Weight (kg)*
9	Filigree	Traditional craft	Souvenir	4650	0.2
10	Galichnik Wedding Costume	Traditional clothing	Apparel	74400	2.5
11	Vranec Wine	Red wine	Beverage	1116	1.5
12	Selsko Meso	Village-style meat dish	Food	868	0.8

Table 11.17: SUPPLIER — Four Macedonian suppliers with contact person details.

Supplier_ID	*Supplier_name*	*First_name*	*Last_name*	*Email*	*Street*	*City*	*Country*	*Zip_code*
S1	Agroplod	Pero	Perovski	contact@agroplod.com.mk	Industrijska 1	Resen	Macedonia	7310
S2	Tikves	Risto	Ristovski	contact@tikves.com.mk	29 Noemvri 2	Kavadarci	Macedonia	1430
S3	Vitaminka	Blagoja	Blagoevski	contact@vitaminka.com.mk	Lece Koteski 23	Prilep	Macedonia	7500
S4	Pelisterka	Zivko	Zivkovski	contact@pelisterka.com.mk	Industrijska 2	Bitola	Macedonia	7000

Table 11.18: ORDER — Eleven customer orders with processing status and key dates.

Order_ID	*Status*	*Order_date*	*Required_date*	*Shipped_date*	*Employee_ID*	*Customer_ID*
Order1	Shipped	15.01.2025	22.01.2025	21.01.2025	E1	C1
Order2	Shipped	20.02.2025	27.02.2025	25.02.2025	E2	C2
Order3	Pending	25.03.2025	30.03.2025	NULL	E3	C3
Order4	Shipped	27.04.2025	06.05.2025	05.05.2025	E4	C4
Order5	Processing	12.05.2025	19.05.2025	NULL	E5	C5
Order6	Shipped	18.05.2025	30.05.2025	29.05.2025	E6	C6
Order7	Canceled	21.05.2025	04.06.2025	NULL	E7	C7
Order8	Shipped	27.05.2025	11.06.2025	10.06.2025	E8	C1
Order9	Pending	01.06.2025	11.06.2025	NULL	E1	C2
Order10	Shipped	20.07.2025	29.07.2025	28.07.2025	E2	C3
Order11	Processing	27.09.2025	02.10.2025	NULL	E3	C4

Table 11.19: ORDER_LINE — Individual line items for each order, referencing the ordered product.

Order_ID	*Line_number*	*Unit_price*	*Quantity*	*Discount*	*Product_ID*
Order1	Line1	372	2	0	1
Order2	Line2	682	1	0	3
Order3	Line1	930	3	0.05	4
Order4	Line1	465	4	0	5
Order5	Line3	620	1	0.1	6
Order6	Line1	9300	6	0	7

Order_ID	*Line_number*	*Unit_price*	*Quantity*	*Discount*	*Product_ID*
Order7	Line1	31000	1	0	8
Order8	Line1	4650	5	0	9
Order9	Line2	1116	2	0	11
Order10	Line3	868	2	0	12
Order11	Line2	74400	2	0.05	10

Table 11.20: EMPLOYEE_PHONE — Multivalued phone numbers for employees.

Employee_ID	*Phone_number*
E1	+38970/212-789
E2	+38975/812-547
E2	+38977/612-348
E3	+38972/423-726
E4	+38971/513-701
E4	+38976/413-621
E5	+38971/612-348
E6	+38972/312-347
E6	+38975/612-517
E7	+38977/423-726
E8	+38975/812-547
E8	+38977/612-348

Table 11.21: CUSTOMER_PHONE — Multivalued phone numbers for customers.

Customer_ID	*Phone_number*
C1	+38970/212-789
C2	+38975/812-547
C3	+38977/612-348
C4	+38972/423-726
C5	+38972/378-795
C6	+38976/413-621
C7	+38971/612-348

Table 11.22: SUPPLIER_PHONE — Multivalued phone numbers for suppliers.

Supplier_ID	*Phone_number*
S1	+38975/812-547
S2	+38977/612-348
S3	+38972/423-726
S4	+38970/212-789

Table 11.23: SUPPLIES — Supply relationships between products and their suppliers, including supply dates and unit costs.

Product_ID	*Supplier_ID*	*Supply_date*	*Unit_cost*
1	S1	15.01.2025	250
2	S2	20.02.2025	280
3	S3	25.03.2025	450
4	S2	27.04.2025	465
5	S3	12.05.2025	300
6	S2	18.05.2025	420
7	S3	21.05.2025	720
8	S2	27.05.2025	650
9	S3	01.06.2025	116
10	S2	20.07.2025	868
11	S4	15.08.2025	700
12	S4	10.09.2025	550

Table 11.24: STORED_IN — Inventory records showing which products are stored in which warehouses, with stock quantities and reorder thresholds.

Product_ID	*Warehouse_ID*	*Quantity_on_hand*	*Reorder_level*
1	W1	57	10
2	W1	21	6
3	W3	18	4
4	W2	32	15
5	W1	14	6
6	W3	12	9
7	W1	9	7
8	W2	23	16
9	W3	6	1
10	W4	7	3
11	W2	72	10
12	W1	18	6

Short excerpts from the reviewers

Prof. Atanas Hristov, PhD

The textbook "Practical Database Design: From Concept to Implementation" represents a significant contribution to domestic academic literature, offering a modern, narrative approach that meets the demands of the information age. The authors successfully bridge the gap between theoretical foundations and real-world implementation, guiding students through the design process in a way that is both educational and inspiring.

Based on the analysis of the material, this work represents a valuable contribution to Macedonian professional literature in the field of information technology and is fully aligned with the course content of "Databases I" at the University of Information Science and Technology "St. Paul the Apostle" — Ohrid. I recommend this manuscript for publication as a university textbook.

Short excerpts from the reviewers

Assoc. Prof. Rasim Salkoski, PhD

The manuscript by authors Capeska Bogatinoska and Karadimce is a thorough and methodologically precise teaching work that covers the fundamental aspects of relational databases, resting on a rigorous theoretical foundation and adhering to E.F. Codd's mathematical principles and the ACID principles of data integrity, which are essential for understanding data consistency and reliability.

The text is marked by a high degree of clarity and consistency in presenting conceptual and logical modeling, as well as SQL implementation, and is well suited to the academic level of undergraduate students at UIST. I propose that the manuscript be accepted and published as the core textbook for the stated courses.

Copyright

© 2026 Dijana Capeska Bogatinoska, Aleksandar Karadimce

All rights reserved, including the right to reproduce this book or portions thereof in any form whatsoever.

For information, address the authors.

CIP - Каталогизација во публикација
Национална и универзитетска библиотека "Св. Климент Охридски", Скопје

004.65(075.8)

CAPESKA Bogatinoska, Dijana
Practical database design: from concept to implementation
[Електронски извор] / Dijana Capeska Bogatinoska, Aleksandar Karadimce.
- Текст во PDF формат, содржи XIX, 578 стр. - Ohrid : University of
information science and technology "St. Paul the Apostle", 2026

Начин на пристапување (URL):
https://uist.edu.mk/wp-content/uploads/2026/07/Practical-Database-Design-complete-final-v1-cover.pdf.
- Наслов преземен од екранот. - Опис на изворот на ден 06.07.2026

ISBN 978-608-66225-5-8

1. Karadimce, Aleksandar [автор]
а) Бази на податоци -- Дизајн -- Модели -- Имплементација --
Високошколски учебници

COBISS.MK-ID 69068549