SECOND EDITION

Information Modeling and Relational Databases

TERRY HALPIN
TONY MORGAN

MORGAN KAUFMANN

Information Modeling and Relational Databases

Second Edition

The Morgan Kaufmann Series in Data Management Systems (Selected Titles)

Information Modeling and Relational Databases, 2nd Edition
Terry Halpin, Tony Morgan

Joe Celko's Thinking in Sets
Joe Celko

Business Metadata
Bill Inmon, Bonnie O'Neil, Lowell Fryman

Unleashing Web 2.0
Gottfried Vossen, Stephan Hagemann

Enterprise Knowledge Management
David Loshin

Business Process Change, 2nd Edition
Paul Harmon

IT Manager's Handbook, 2nd Edition
Bill Holtsnider & Brian Jaffe

Joe Celko's Puzzles and Answers, 2nd Edition
Joe Celko

Making Shoes for the Cobbler's Children
Charles Betz

Joe Celko's Analytics and OLAP in SQL
Joe Celko

Data Preparation for Data Mining Using SAS
Mamdouh Refaat

Querying XML: XQuery, XPath, and SQL/XML in Context
Jim Melton and Stephen Buxton

Data Mining: Concepts and Techniques, 2nd Edition
Jiawei Han and Micheline Kamber

Database Modeling and Design: Logical Design, 4th Edition
Toby J, Teorey, Sam S. Lightstone, Thomas P. Nadeau

Foundations of Multidimensional and Metric Data Structures
Hanan Samet

Joe Celko's SQL for Smarties: Advanced SQL Programming, 3rd Edition
Joe Celko

Moving Objects Databases
Ralf Hartmut Güting and Markus Schneider

Joe Celko's SQL Programming Style
Joe Celko

Data Mining, Second Edition: Concepts and Techniques
Ian Witten and Eibe Frank

Fuzzy Modeling and Genetic Algorithms for Data Mining and Exploration
Earl Cox

Data Modeling Essentials, 3rd Edition
Graeme C. Simsion and Graham C. Witt

Location-Based Services
Jochen Schiller and Agnès Voisard

Database Modeling with Microsoft® Visio for Enterprise Architects
Terry Halpin, Ken Evans, Patrick Hallock, Bill Maclean

Designing Data-Intensive Web Applications
Stephano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, Maristella Matera

Mining the Web: Discovering Knowledge from Hypertext Data
Soumen Chakrabarti

Advanced SQL: 1999—Understanding Object-Relational and Other Advanced Features
Jim Melton

Database Tuning: Principles, Experiments, and Troubleshooting Techniques
Dennis Shasha, Philippe Bonnet

SQL:1999—Understanding Relational Language Components
Jim Melton, Alan R. Simon

Information Visualization in Data Mining and Knowledge Discovery
Edited by Usama Fayyad, Georges G. Grinstein, Andreas Wierse

Transactional Information Systems
Gerhard Weikum and Gottfried Vossen

Spatial Databases
Philippe Rigaux, Michel Scholl, and Agnes Voisard

Managing Reference Data in Enterprise Database
Malcolm Chisholm

Understanding SQL and Java Together
Jim Melton and Andrew Eisenberg

Database: Principles, Programming, and Performance, 2nd Edition
Patrick and Elizabeth O'Neil

The Object Data Standar
Edited by R. G. G. Cattell, Douglas Barry

Data on the Web: From Relations to Semistructured Data and XML
Serge Abiteboul, Peter Buneman, Dan Suciu

Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations
Ian Witten, Eibe Frank

Joe Celko's Data and Databases: Concepts in Practice
Joe Celko

Developing Time-Oriented Database Applications in SQL
Richard T. Snodgrass

Web Farming for the Data Warehouse
Richard D. Hackathorn

Management of Heterogeneous and Autonomous Database Systems
Edited by Ahmed Elmagarmid, Marek Rusinkiewicz, Amit Sheth

Object-Relational DBMSs: 2nd Edition
Michael Stonebraker and Paul Brown,with Dorothy Moore

Universal Database Management: A Guide to Object/Relational Technology
Cynthia Maro Saracco

Readings in Database Systems, 3rd Edition
Edited by Michael Stonebraker, Joseph M. Hellerstein

Understanding SQL's Stored Procedures: A Complete Guide to SQL/PSM
Jim Melton

Principles of Multimedia Database Systems
V. S. Subrahmanian

Principles of Database Query Processing for Advanced Applications
Clement T. Yu, Weiyi Meng

Advanced Database Systems
Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard T. Snodgrass, V. S. Subrahmanian, Roberto Zicari

Principles of Transaction Processing
Philip A. Bernstein, Eric Newcomer

Using the New DB2: IBMs Object-Relational Database System
Don Chamberlin

Distributed Algorithms
Nancy A. Lynch

Active Database Systems: Triggers and Rules For Advanced Database Processing
Edited by Jennifer Widom, Stefano Ceri

Migrating Legacy Systems: Gateways, Interfaces, & the Incremental Approach
Michael L. Brodie, Michael Stonebraker

Atomic Transactions
Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete

Query Processing for Advanced Database Systems
Edited by Johann Christoph Freytag, David Maier, Gottfried Vossen

Transaction Processing
Jim Gray, Andreas Reuter

Database Transaction Models for Advanced Applications
Edited by Ahmed K. Elmagarmid

A Guide to Developing Client/Server SQL Applications
Setrag Khoshafian, Arvola Chan, Anna Wong, Harry K. T. Wong

Information Modeling and Relational Databases

Second Edition

Terry Halpin
Neumont University

Tony Morgan
Neumont University

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann Publishers is an imprint of Elsevier

MORGAN KAUFMANN PUBLISHERS

Publishing Director Chris Williams
Acquisitions Editor Denise E. M. Penrose
Publishing Services Manager George Morrison
Project Manager Andre A. Cuello
Assistant Editor Mary James
Composition Terry Halpin and Tony Morgan
Interior printer Sheridan Books, Inc.
Cover printer Phoenix Color
Cover Design Alisa Andreola

Morgan Kaufmann Publishers is an imprint of Elsevier.
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA

This book is printed on acid-free paper.

© 2008 by Elsevier Inc. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, scanning, or otherwise—without prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, E-mail: permissions@elsevier.com. You may also complete your request online via the Elsevier homepage (http://elsevier.com), by selecting
"Support & Contact" then "Copyright and Permission" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data
(Application Submitted)

ISBN: 978-0-12-373568-3

For information on all Morgan Kaufmann publications,
visit our Web site at www.mkp.com or www.books.elsevier.com

Printed in the United States of America
08 09 10 11 12 10 9 8 7 6 5 4 3 2 1

Working together to grow
libraries in developing countries
www.elsevier.com | www.bookaid.org | www.sabre.org
ELSEVIER BOOK AID International Sabre Foundation

To Norma and Gwen, our wonderful wives.

Terry and Tony

This page intentionally left blank

Contents

Foreword *by John Zachman*
Foreword *by Sjir Nijssen*
Foreword *by Gordon Everest*
Preface

1 Introduction 1

1.1 Information Modeling 2
1.2 Modeling Approaches 6
1.3 Some Historical Background 19
1.4 The Relevant Skills 23
1.5 Summary 24

2 Information Levels and Frameworks 27

2.1 Four Information Levels 28
2.2 The Conceptual Level 32
2.3 Database Design Example 43
2.4 Development Frameworks 48
2.5 Summary 55

3 Conceptual Modeling: First Steps 59

3.1 Conceptual Modeling Language Criteria 60
3.2 Conceptual Schema Design Procedure 62
3.3 CSDP Step 1: From Examples to Elementary Facts 63
3.4 CSDP Step 2: Draw Fact Types and Populate 81
3.5 CSDP Step 3: Trim Schema; Note Basic Derivations 95
3.6 Summary 104

4 Uniqueness Constraints 109

4.1 Introduction to CSDP Step 4 110
4.2 Uniqueness Constraints on Unaries and Binaries 111
4.3 Uniqueness Constraints on Longer Fact Types 122
4.4 External Uniqueness Constraints 128
4.5 Key Length Check 137
4.6 Projections and Joins 150
4.7 Summary 155

5 Mandatory Roles 159

5.1 Introduction to CSDP Step 5 160
5.2 Mandatory and Optional Roles 162
5.3 Reference Schemes 174
5.4 Case Study: A Compact Disc Retailer 193
5.5 Logical Derivation Check 200
5.6 Summary 207

6 Value, Set-Comparison, and Subtype Constraints 211

6.1 Introduction to CSDP Step 6 212
6.2 Basic Set Theory 212
6.3 Value Constraints and Independent Types 216
6.4 Subset, Equality, and Exclusion Constraints 224
6.5 Subtyping 238
6.6 Generalization of Object Types 260
6.7 Summary 268

7 Other Constraints and Final Checks 271

7.1 Introduction to CSDP Step 7 272
7.2 Occurrence Frequencies 272
7.3 Ring Constraints 277
7.4 Other Constraints and Rules 289
7.5 Final Checks 295
7.6 Summary 303

8 Entity Relationship Modeling 305

8.1 Overview of ER 306
8.2 Barker notation 308
8.3 Information Engineering notation 318
8.4 IDEF1X 322
8.5 Mapping from ORM to ER 334
8.6 Summary 342

9 Data Modeling in UML 345

9.1 Introduction 346
9.2 Object-Orientation 348
9.3 Attributes 351
9.4 Associations 357
9.5 Set-Comparison Constraints 364
9.6 Subtyping 372
9.7 Other Constraints and Derivation Rules 376
9.8 Mapping from ORM to UML 388
9.9 Summary 395

10 Advanced Modeling Issues 399

10.1 Join Constraints 400
10.2 Deontic Rules 408
10.3 Temporality 411
10.4 Collection Types 432
10.5 Nominalization and Objectification 439
10.6 Open/Closed World Semantics 450
10.7 Higher-Order Types 456
10.8 Summary 469

11 Relational Mapping 473

11.1 Implementing a Conceptual Schema 474
11.2 Relational Schemas 475
11.3 Relational Mapping Procedure 483
11.4 Advanced Mapping Aspects 510
11.5 Summary 525

12 Data Manipulation with Relational Languages 527

12.1 Relational Algebra 528
12.2 Relational Database Systems 554
12.3 SQL: Historical and Structural Overview 556
12.4 SQL: Identifiers and Data Types 558
12.5 SQL: Choosing Columns, Rows, and Order 562

12.6 SQL: Joins 570
12.7 SQL: In, Between, Like, and Null Operators 582
12.8 SQL: Union and Simple Subqueries 591
12.9 SQL: Scalar Operators and Bag Functions 602
12.10 SQL: Grouping 611
12.11 SQL: Correlated and Existential Subqueries 619
12.12 SQL: Recursive Queries 626
12.13 SQL: Updating Table Populations 629
12.14 Summary 631

13 Using Other Database Objects 637

13.1 SQL: The Bigger Picture 638
13.2 SQL: Defining Tables 638
13.3 SQL: Views 646
13.4 SQL: Triggers 652
13.5 SQL: Routines 655
13.6 SQL: More Database Objects 658
13.7 Transactions and Concurrency 662
13.8 Security and Meta-Data 664
13.9 Exploiting XML 666
13.10 Summary 684

14 Schema Transformations 687

14.1 Schema Equivalence and Optimization 688
14.2 Predicate Specialization and Generalization 692
14.3 Nesting, Coreferencing, and Flattening 702
14.4 Other Transformations 718
14.5 Conceptual Schema Optimization 722
14.6 Normalization 734
14.7 Denormalization and Low Level Optimization 753
14.8 Reengineering 759
14.9 Data Migration and Query Transformation 766
14.10 Summary 769

15 Process and State Modeling 773

15.1 Introduction 774
15.2 Processes and Workflow 777
15.3 State Models 785
15.4 Foundations for Process Theory 795
15.5 Modeling Information Dynamics in UML 800
15.6 Business Process Standards Initiatives 812
15.7 Standard Process Patterns 819
15.8 Summary 832

16 Other Modeling Aspects and Trends 835

16.1 Introduction 836
16.2 Data Warehousing and OLAP 836
16.3 Conceptual Query Languages 843
16.4 Schema Abstraction Mechanisms 852
16.5 Further Design Aspects 857
16.6 Ontologies and the Semantic Web 864
16.7 Postrelational Databases 871
16.8 Metamodeling 881
16.9 Summary 888

ORM glossary 893
ER glossary 903
UML glossary 907
Useful Web Sites 911
Bibliography 913
Index 925

This page intentionally left blank

Foreword

by John Zachman, Founder and President
Zachman International

It gives me great personal pleasure to write this foreword. I wrote the foreword to the first edition of *Information Modeling and Relational Databases* and to be brutally honest, I liked my first foreword and I haven't at all changed my mind, with the exception that I like the second edition even more than the first edition, if that is even possible. Anyone familiar with my work would know that I have been arguing for many years that an enterprise ontology must include more structural components than those typically related to information. Terry Halpin and Tony Morgan have incorporated some additional structural variables in this new edition.

I suppose you would have expected this, but the second edition even surpasses the first, not only in terms of the updated and expanded modeling coverage, now including XML, business processes, and even the Semantic Web, and the plethora of exercises, but in terms of the significance of seven more years of experience and wisdom that can only be accumulated through the concentrated and intense investment of one's life.

Because I liked my first Foreword, it is hard for me to materially improve on it, so I will borrow heavily from its basic content, making adjustments as appropriate.

I have known Terry Halpin for many years. I have known *about* Terry Halpin for many more years than I have actually known him personally. His reputation precedes him, and—take it from me—he is one of those people who is bigger than his reputation and far more humble than his contribution warrants. I have not known Tony Morgan for nearly as long but I know many people who have worked with Tony and have the highest regard for his work.

Both of these men have invested a lifetime in developing these enterprise modeling concepts, not because of the enormous market demand, but because of their intense belief that these concepts are vital for the advancement of our capability as humans to accommodate the extreme complexity and rates of change that characterize Twenty First Century life.

In fact, those of us who have devoted our lives to some of these apparently esoteric modeling pursuits are not doing it to make money, because the general market is hardly even aware of the issues, much less willing to pay for them. We are doing it because we know it is right and we are certain that survival of life as we know it is not dependent on writing more code. It is dependent upon being able to describe the complexities of enterprises (any or all human endeavor) so they can actually be designed, implemented as conceived, and dynamically changed as the environment changes around them.

We all owe Terry and Tony a debt of gratitude for persevering to produce this comprehensive modeling work.

When Terry asked me to write the industrial foreword to the first edition of this book, my first reaction was, "Good Night! Am I qualified to write a foreword for a Terry Halpin book"? I suggested that he send it to me and I would decide whether I could write it for him or not. After he sent me the book, my next problem was, I couldn't put the book down! Can you imagine that? A technical book that keeps you wanting to read the next page?

Yes, it is a technical book, and the second edition is hardly any different. It is a *very* technical book that goes into detail on how to produce graphic models that exquisitely and rigorously capture the semantic details of an information system. But, it is also an easy-to-read book because it spells out clearly, concisely, and so simply the complexities of logic that provide any enterprise and any natural language with its capacity for nuance of expression and richness of description. For every step in the logic, there is provision of illustration and a test for your comprehension. There are hosts of models and exercises of real cases, none so contrived or so convoluted that it takes more energy to understand the case than to get the point.

Yes, Object Role Modeling 2 (ORM 2) is the notation for most of the illustrations, not simply because Terry actually "wrote the book" on ORM, but because of its incomparable ability to capture semantic intent and unambiguously express it graphically. And, yes, there is a discussion of ORM 2 modeling in sufficient detail for modelers to acquire the ORM 2 language capability. But the cases and illustrations are rich with analysis that can help even modelers unfamiliar with ORM to accurately extract the precise semantics of a "universe of discourse."

But to me, all of this is not the strength of this book. The enduring strength of the book is two-fold. First, this is a very clear and vivid demonstration of the incredible complexities of accurately discerning and capturing the intentions of the enterprise and transforming them into the realities of an implementation. There is little wonder why the systems we have been implementing for the last 50 years (total history of "Data Processing") are so inflexible, unadaptable, misaligned, unintegrated, unresponsive, expensive, unmaintainable, and so frustrating to management. We never bothered to produce an accurate description of the concepts of the enterprise in the first place!

If you don't rigorously describe the enterprise to begin with, why would anybody expect to be able to produce a relevant design and implementation that reflected enterprise management's reality or intent, or that could be adapted over time to accommodate their changes?

Tragically, few general managers are likely to read so technical a book as the second edition of *Information Modeling and Relational Databases*. But *all* general managers ought to read this book to get some perspective on the semantic complexity of their own enterprises, of the challenges of accurately capturing that complexity, of the necessity of their own participation and decisions in conceptual modeling, of the sophistication of the engineering that is required to produce quality and flexible implementations, of the fact that systems (automated or not automated) are not magic, they are logic and good judgment and engineering rigor and a lot of hard work.

In fact, every data modeler regardless of his or her syntactic specialty—whether it be Chen, Barker, Finkelstein, IDEF1X, IDEF1x(Object), UML 2, XML, or XYZ—ought to read the book for the same reasons. In fact, modelers of all kinds ought to read the book. In fact, every *programmer* ought to read the book. In fact, anyone who has anything to do with information or information systems ought to read the book!

The second strength of this book lies in the derivations from the high standard of semantic expression established by employing the second version of Object Role Modeling. Having demonstrated that it is possible to be rigorous and graphic in capturing precise semantic intent, the book straight-forwardly evaluates all the other popular graphic modeling notations in terms of their ability to duplicate that expression. There is a comparison with every other modeling notation that I have ever heard of, including the ones I mentioned above like Chen, IDEF1X, UML 2 etc. This is the most objective and precise comparison I have ever seen. The authors are very apologetic about appearing to be critical of other languages, but my observation is that this comparison was the most dispassionate and objective discussion I have ever seen. They even point out the strengths of these other notations and how and where in the overall process they can be used effectively. How's that for objectivity?!

There is one more interesting dimension of these rigorous, precise semantic models —they have to be transformed into databases for implementation. The authors describe in detail and by illustration the transformation to logical models, to physical database design, and to implementation. In this context, it is easy to evaluate and compare the various database implementation possibilities including relational databases, object-oriented databases, object-relational databases, and declarative databases; and they throw in star schemas and temporal databases for good measure! Once again, I cannot remember seeing so dispassionate and objective an evaluation and comparison of the various database structures. Within this context, it is straight-forward to make a considered and realistic projection of database technology trends into the foreseeable future.

This is a book that is going to sit on my bookshelf forever. I would consider it a candidate to be one of the few classics in our rather young, 50-year old discipline of information management. I hope I have the personal discipline to pick it up about once a year and refresh my understanding of the challenges and the possibilities of the information profession. I am confident you will find the second edition of *Information Modeling and Relational Databases* as useful and enlightening as I have, and I hope that there will be many more editions to come!

This page intentionally left blank

Foreword

by Professor Dr. Sjir Nijssen, CTO
PNA Group, The Netherlands

It gives me great personal pleasure to write this foreword. I have known Terry Halpin since 1986. As John Zachman has said about Terry, he is one of those people who is bigger than his reputation and far more humble than his contribution warrants. Terry is one of the most effective and dedicated authors of a new wave in knowledge engineering and requirements specification. I would like to classify ORM (Terry's focus, called Object-Role Modeling) as a fact orientation approach. This by itself is already much broader than data modeling. It is my professional opinion, based on extensive experience during more than 40 years in business and academia, that fact orientation is the most productive data modeling approach, by far. This approach could be considered as a best business practice for SBVR (*Semantics of Business Vocabulary and Business Rules*), the standard adopted by the Object Management Group (OMG) on December 11, 2007.

With fact orientation, it is useful to distinguish between structure and structuring. Both are important in practice and theory. With respect to structuring, one of the subprocesses is verbalization. Verbalization is a major and unique part of the CogNIAM and ORM methodology. This entered the fact orientation community in 1959, when I was training young people to plot the movements of airplanes in an area where radar could not yet see. In 1967 I got the chance of a professional lifetime. My manager said "*I want to hire you, and you have only one mission: nearly every software professional in our company* (Control Data Corporation, one of the most powerful computer companies in that period) *is preoccupied with programming. I want you to ignore procedural programming, and concentrate on the data underlying all programming*".

During the seventies, conceptual modeling—of which ORM is an instance—was developed primarily in Europe by a group of people from various companies and universities. However, two excellent American researchers also contributed substantially: Dr. Michael Senko and Dr. Bill Kent, both of IBM. Anyone reading this book will also

enjoy reading the classics of Mike Senko published in the *IBM Systems Journal* and the pearl *Data and Reality*, the book written by Dr. Kent. In that period, NIAM was conceived. In the late seventies a group of international conceptual modelers undertook in ISO the task of writing the report *Concepts and Terminology for the Conceptual Schema and the Information Base*. It was a report about conceptual modeling, natural language, and logic.

In 1986, when I was professor of computer science at the University of Queensland, a relatively young Terry became my colleague as lecturer in the Computer Science Department. What a fantastic colleague! He very quickly caught on to my lecturing on conceptual modeling. It quickly became apparent to me that he was full of ideas, resulting in many collaborative sessions and the further development of the NIAM method at that time. His own lecture preparation was excellent, generating many exercises, and it was a real pleasure to work with him. We jointly published a book in 1989 which was largely written by Terry. In 1989, Terry completed a doctoral thesis that formalized and enhanced NIAM, and he and I went our own separate ways as I decided to return to The Netherlands.

In the following period, many excellent extensions to the NIAM 1989 version were added by Terry and his team on the one hand, and myself and associates on the other. In retrospect, I consider it a very good approach to work independently for some time and subsequently come to the conclusion that beautiful improvements have been developed. Two years ago, we decided to establish the best combination of the improvements developed independently. One of the strong points of our methodology is its incomparable ability to capture precise semantic intent and unambiguously express it graphically.

I strongly recommend every serious data modeler, business process modeler, and programmer to study this excellent book very carefully. I perused the chapters with pleasure and found it very useful and clearly presented. It is an excellent textbook for universities that intend to provide a first-class conceptual modeling course. UML has received much greater attention than ORM up till now, and I personally find that a shame, because in my opinion there are many areas in which ORM outperforms UML. I expect that this point will become clear to all with UML experience, who have their first encounter with ORM by reading this book. I therefore hope this book will result in further attention to ORM, which would be much deserved.

The relationship between the new OMG standard SBVR and ORM is not explicitly mentioned in the text, but be assured that there is a clear philosophical link between them. People familiar with both standards will recognize this easily.

Terry's coauthor, Tony, has added a very interesting chapter about processes, and has incorporated the task of modeling these processes as part of conceptual modeling. Tony is an excellent teacher, as I have recently had the pleasure of witnessing at the ORM2007 Conference in Portugal: British humor thrown in for free!

It is my distinct pleasure to highly recommend this book to anybody seriously interested in acquiring competence in conceptual analysis or modeling, with the aim of making modeling an understandable form of engineering instead of considering it as just an art.

Foreword

by Dr. Gordon C. Everest, Professor Emeritus and Adjunct, MIS and DBMS
Carlson School of Management, University of Minnesota, USA.

I am delighted and honored to write a foreword to this second edition. It gives me another opportunity to convince those in the world of data modeling that there is a better way. I am absolutely convinced that Object Role Modeling (ORM) is a better way to do data modeling. My underlying motive in this foreword is to sufficiently perk your interest to seriously study ORM, and this book is the best resource available to you.

Data modeling is the foundation of information systems development—if you don't get the database design "right" then the systems you build will be like a house of cards, collapsing under the weight of inevitable future forces for revision, enhancement, integration, and quality improvement. Thus, we need a scheme to guide our data modeling efforts to produce data models that clearly and accurately represent the users' domain of discourse and facilitate human communication, understanding, and validation.

This book is a must for anyone who is serious about data modeling, but with a caution: you must devote enough time and effort to really understand ORM. Fortunately, I have my students as a captive audience for a whole semester—long enough for them to learn and practice ORM and become convinced that it is a better way. With ORM you can raise your data modeling skills to a whole new level.

This book also examines record-based modeling schemes: UML, SQL based on "Ted" Codd's relational model, and Peter Chen's Entity Relationship (ER) diagrams with many variations—Barker as in Oracle, Finkelstein's Information Engineering (IE), and IDEF1X (as in ERwin). Viewing these familiar modeling approaches from an ORM perspective provides an enriched understanding of their underlying nature.

Record-based modeling schemes use three constructs: Entity, Attribute, and Relationship. It is the clustering of attributes into entity records that is the root of many of our problems in data modeling. Normalization is the test to see if we clustered too much, and record decomposition is commonly used as a remedy to correct a violation of the normal forms.

Normalization is the Achilles heel of data modeling. Oh, to be able to avoid normalization altogether? The mere suggestion is intriguing to students and practitioners of data modeling. Well, with ORM you can. The problem stems from the lack of clear definition of relationships when we throw stuff into a record, so that the intra-record structure is implicitly defined or assumed. ORM forces you to separately consider and define all relevant relationships and constraints among the object domains in your universe.

ORM is actually based on only two constructs: objects and relationships (which correspond to the concepts of nouns as subject or object, and verbs as predicates in sentences). Both entities and attributes are treated as objects in ORM (not to be confused with objects in object-oriented technology). Objects play roles in relationships with other objects. Objects have attributes or descriptors by virtue of the roles they play in relationships with other objects. In record-based modeling, there are two kinds of relationships: inter-record, and intra-record among attributes. In ORM all relationships are represented the same way with a single construct. When the ORM model is a valid representation of the world being modeled, the functional and multivalued dependencies are explicitly defined, and hence, the generation of "records" (in a relational table) can be automated and can guarantee that the result will be fully normalized (to 5NF). That's good news for data modelers.

ORM does not supplant ER diagrams or relational database designs, rather it is a stage before. It can enable, enlighten, and inform our development and understanding of ER/relational data models. We build records more for system efficiency, than for human convenience or comprehension. The premature notion of a record (a cluster of attribute domains along with an identifier to represent an entity) actually gets in the way of good data modeling. ORM does not involve records, tables, or attributes. As a consequence, we don't get bogged down in "table think"—there is no need for an explicit normalization process.

The second edition is even more focused on the centrality of data in information systems, and on the importance of semantics. Starting with the realization that users (collectively) know more than we could ever capture in a data model, we must use a data modeling scheme that captures the widest possible range of semantics, and express this meaning graphically. Semantics is paramount, and ORM goes way beyond any record-based modeling scheme in graphically depicting semantics. With this second edition, Terry and Tony have expanded the scope of ORM to include temporality, dynamics, state modeling, and business processes.

Well, is that sufficient to pique your interest in learning more about ORM? If you are a would-be student of ORM and you take data modeling seriously, I encourage you to invest some time to read this book. You won't regret it. You will grow to appreciate ORM and will become a better data modeler for it. In order to develop effective and maintainable information systems we need good data models, and for that we need a good data modeling methodology. ORM allows us to develop database designs at the highest conceptual level, unencumbered by things that are not of primary concern to user domain specialists. My deep desire is to see more and more database designers using ORM. The systems we build and the world we live in will be better for it. Join me in this journey and enjoy the adventure.

Preface

This book is about information systems, focusing on information modeling and relational database systems. It is written primarily for data modelers and database practitioners as well as students of computer science or information management. It should also be useful to anyone wishing to formulate the information structure of business domains in a way that can be readily understood by humans yet easily implemented on computers. In addition, it provides a simple conceptual framework for understanding what database systems really are, and a thorough introduction to SQL and other key topics in data management.

A major part of this book deals with *fact-oriented modeling*, a conceptual modeling approach that views the world in terms of simple facts about objects and the roles they play. Originating in Europe, fact-orientation is today used worldwide and comes in many flavors, including the Semantics of Business Vocabulary and Business Rules (SBVR) approach adopted in 2007 by the Object Management Group. The version of fact-orientation described in this book is second generation *Object-Role Modeling* (ORM 2), and is based on extensions to NIAM (Natural-language Information Analysis Method).

Two other popular notations for information modeling are *Entity-Relationship* (ER) diagrams and *Unified Modeling Language* (UML) class diagrams. For conceptual information analysis, the ORM method has several advantages over the ER and UML approaches. For example, ORM models can be easily verbalized and populated for validation with domain experts, they are more stable under changes to the business domain, and they typically capture more business rules in diagram form.

However ER diagrams and UML class diagrams are good for compact summaries, and their structures are closer to the final implementation, so they also have value. Hence the coverage includes chapters on data modeling in ER and UML, and indicates how ER and UML data models can be easily generated from ORM models.

To make the text more approachable to the general reader with an interest in databases, the language is kept simple, and a formalized, mathematical treatment is deliberately avoided. Where necessary, relevant concepts from elementary logic and set theory are discussed prior to their application. Most of the material in this book has been class tested in courses to both industry and academia, and the basic ORM method has been taught successfully even at the high school level. The content is modularized, so that instructors wishing to omit some material may make an appropriate selection for their courses.

The first chapter motivates the study of conceptual modeling, and briefly compares the ORM, ER, and UML approaches. It also includes an historical and structural overview of information systems. Chapter 2 provides a structural background, explaining the conceptual architecture of, and development frameworks for, information systems. It introduces a number of key concepts that are dealt with more thoroughly in later chapters, and should be read in full by the reader with little or no database experience.

Chapter 3 is fundamental. Following an overview of conceptual modeling language criteria and the ORM Conceptual Schema Design Procedure (CSDP), this chapter covers the first three steps of the CSDP. The first step (verbalizing familiar examples in terms of elementary facts) may seem trivial, but it should not be rushed, as it provides the foundation for the model. The rest of this chapter covers the basic graphical notation for fact types, and then offers guidance on how to classify objects into types and identify information that can be arithmetically derived.

Chapter 4 begins the task of specifying constraints on the populations of fact types. The most important kind of constraint (the uniqueness constraint) is considered in detail. Then some checks on the elementarity of the fact types are discussed. This chapter also introduces the join and projection operations at the conceptual level—the relational version of these operations is important in the later work on relational databases.

Chapter 5 covers mandatory role constraints, including a check for detecting information that can be logically derived. Reference schemes are then examined in some depth. Some of the more complex reference schemes considered here could be skipped in a short course. The CSDP steps covered so far are then reviewed by applying them in a case study, and the logical derivation check is then considered.

Chapter 6 covers value, set comparison (subset, equality, and exclusion), and subtyping constraints. Section 6.6 deals with advanced aspects of subtyping—though important in practice, the material in this section could be skimmed over in a first reading.

Chapter 7 deals with the final step of the conceptual schema design procedure. Less common constraints are considered (e.g., occurrence frequencies and ring constraints), and final checks are made on the design. Sections 7.3–7.5 are somewhat advanced, and could be skipped in a short course.

Chapter 8 discusses the Entity Relationship (ER) approach, starting with Chen's original notation then moving on to the three most popular notations in current use: the

Barker ER notation, the Information Engineering notation, and the IDEF1X notation (actually a hybrid of ER and relational notations). Comparisons with ORM are included along the way.

Chapter 9 examines the use of UML class diagrams for data modeling, including a detailed comparison with ORM. Business rule constructs in ORM with no graphic counterpart in UML are identified and then captured in UML using user-defined constraints or notes.

Chapter 10 considers several advanced aspects of information modeling, such as join constraints, historical fact types, collection types, open/closed world semantics, and higher-order types. The discussion of deontic rules and nominalization is fundamental to understanding the SBVR flavor of fact-orientation. This chapter is technically the most challenging in the book, and could be skipped in an introductory course.

Chapter 11 describes how a conceptual model may be implemented in a relational database system. The first three sections are fundamental to understanding how a conceptual schema may be mapped to a relational schema. Section 11.4 considers advanced mapping aspects, and could be omitted in a short course.

Chapter 12 provides a foundational introduction to relational databases and SQL queries. Section 12.1 covers relational algebra—although not used as a practical query language, the algebra is important for understanding the basic relational operations supported by SQL. Section 12.2 provides an overview of how the relational model of data compares with data models adopted by some relational database management systems. Sections 12.3–12.14 cover the main features of SQL, with attention to the SQL-89, SQL-92, SQL:1999, SQL:2003, and SQL:2008 standards, and some popular dialects.

Chapter 13 discusses further aspects of SQL (e.g., data definition, triggers, and stored procedures), the use of other languages such as XML in conjunction with SQL, and introduces some practical issues such as security, metadata and concurrency.

Chapter 14 discusses how and when to transform one schema into another schema at the same level (conceptual or logical). Sections 14.1–14.4 examine the notion of conceptual schema equivalence, and ways in which conceptual schemas may be reshaped. As one application of this theory, section 14.5 specifies a procedure for optimizing a database design by performing conceptual transformations before mapping. Section 14.6 provides a concise coverage of normalization theory, including some new insights. Section 14.7 briefly considers denormalization and low-level optimization. Sections 14.8–14.9 illustrate the role of conceptual optimization in database reengineering, and conclude with a discussion of data migration and query transformation. Sections 14.4, 14.5, 14.7, 14.8, and 14.9 are of an advanced nature and may be skipped in a short course. In a very short course, the whole chapter could be skipped.

Chapter 15 broadens the treatment of information systems analysis by examining behavioral aspects of business using process and state models. The fundamental concepts underlying business processes and workflows are explained, including popular graphical notations, process patterns, and process standards. Some ways of integrating behavioral models with information models are also considered.

Chapter 16 examines other modeling aspects and trends. Topics covered include data warehousing, conceptual query languages, schema abstraction mechanisms, further design aspects, ontologies and the semantic web, post-relational databases (e.g. object databases and object-relational databases) and metamodeling. Though these topics are important and interesting, they could be omitted in a short course.

In line with the ORM method, this text adopts a "cookbook" approach, with plenty of diagrams and examples. Each chapter begins with a brief overview, and ends with a chapter summary of the major points covered, with chapter notes to provide fine points and further references. One of the major features of the book is its large number of exercises, which have been thoroughly class-tested. A bibliography of all cited references is included at the end of the book, where you will also find glossaries of technical symbols and terms for ORM, ER, and UML (class diagrams only). A comprehensive index provides easy access to explanations of technical topics.

For readers familiar with the previous edition of this book, the major differences are now summarized. The coverage of ORM and UML has substantially updated to cover their latest versions (ORM 2 and UML 2), which necessitated the redrawing of almost all diagrams in the earlier edition. Whole new chapters have been added (Advanced Modeling Issues, Using Other Database Objects, and Process and State Modeling), as well as new chapter sections (e.g. ontologies and the semantic web). All previous chapters have been substantially revised, with several topics covered in greater depth, and there are many new exercises. The new content has led to a much larger book, which now has two coauthors, with Terry Halpin responsible for Chapters 1–11, 14, 16, most of Chapter 12, and part of Chapter 13, and Tony Morgan responsible for Chapter 15, part of Chapter 12, and most of Chapter 13.

U.S. spelling is used throughout the book. U.S. punctuation rules have also been used, except for quoted items, where item separators and sentence terminators (e.g., commas and periods) appear after, rather than just before, closing quotes.

Online Resources

To reduce the size and hence cost of the book, supplementary material has been made available online at the publisher's Web site (*www.mkp.com/imrd2/*) for downloading. There are at least three appendices. Appendix A provides an overview of the evolution of computer hardware and software. Appendix B discusses two kinds of subtype matrix that can be used to determine subtype graphs from significant populations. Appendix C discusses advanced aspects of SQL, focusing on set-comparison queries and group extrema queries. Appendices on other topics may be added as the need arises.

The answers to the exercises are contained in two files, one for the odd-numbered questions and one for the even-numbered questions. The answers to the odd-numbered questions are openly accessible, but the answers to the even-numbered questions are password protected, in order to provide classroom instructors with a range of exercises for classroom discussion. Additional material on ORM and other topics is available via the URLs listed in the Useful Web Sites section at the back of the book.

Electronic versions of the figures, as well as further exercises and related pedagogic material, are included in supplementary online resources that are available to instructors using this book as a course text. These resources and a password for exercise answers are available to classroom instructors at *http://textbooks.elsevier.com*.

ORM Software

ORM is supported by a variety of modeling tools. At the time of writing, the Neumont ORM Architect (NORMA) tool provides the most complete support for the ORM 2 notation discussed in this book. NORMA is an open source plug-in to Microsoft Visual Studio .NET and may be downloaded, along with supporting documentation, either from *www.ormfoundation.org* or from *http://sourceforge.net/projects/orm*.

The previous version of ORM (ORM 1) is supported as the ORM Source Model Solution in a high end version of Microsoft Visio (Halpin et al. 2003). A discontinued ORM tool known as VisioModeler is freely available as a download from Microsoft's MSDN Web site. Although VisioModeler does not run under Windows Vista, and the product is somewhat outdated in its database driver support, it does allow you to create ORM models under earlier versions of Windows and map them to a range of database management systems. To download the VisioModeler tool, point your browser at *http://msdn.microsoft.com/downloads*, and then search for "VisioModeler 3.1 (Unsupported Product Edition)".

Other tools supporting different flavors of fact orientation include Doctool, CaseTalk, and Infagon. Other fact-oriented tools under development at the time of writing include ActiveFacts and CogNIAM. Links to these tools may be found in the Useful Web Sites section at the back of this book.

Acknowledgments

We greatly appreciate the editorial assistance of Mary James at Morgan Kaufmann, the suggestions made by the anonymous reviewers, the technical editing by Dr. Andy Carver, and the copy editing by Melissa Revell. We also thank Technologies 'N Typography for permission to use their TNT fonts.

Some of this book's content is based on articles written for *The Business Rules Journal*, the *Journal of Conceptual Modeling*, and material from editions of the earlier book *Conceptual Schema and Relational Database Design*, previously published by Prentice Hall Australia and WytLytPub. The first edition of that book was coauthored by Dr. Sjir Nijssen, who wrote the final four chapters (since replaced).

The ORM approach discussed in this book is based on our revisions and extensions to the NIAM method, which was largely developed in its original form by Dr. Eckhard Falkenberg, Dr. Sjir Nijssen, and Prof. Robert Meersman, with other contributions from several researchers. Some of the work discussed in this book was developed jointly with current or former colleagues, including Dr. Anthony Bloesch, Dr. Linda Bird, Dr. Peter Ritson, Dr. Erik Proper, Dr. Andy Carver, and Dr. Herman Balsters. It

has been a pleasure working with these colleagues, as well as with the hundreds of students and practitioners with whom trial versions of the material in this book were tested. We also gratefully acknowledge permission by The University of Queensland and Neumont University to include a small selection of our past assessment questions within the exercises.

A modeling method as good as ORM deserves a good CASE tool. Over the last decade, talented staff at ServerWare, Asymetrix Corporation, InfoModelers Incorporated, Visio Corporation, Microsoft Corporation, and Neumont University have worked to develop state of the art CASE tools to support the specific ORM method discussed in this book. The following talented individuals currently working as lead software engineers on the NORMA tool deserve special mention: Matt Curland and Kevin Owen.

Finally we thank our wives, Norma and Gwen, for being so understanding and supportive while we were busily occupied in the writing task.

1 Introduction

1.1 Information Modeling
- Information = data + semantics
- Database systems
- The need for good design

1.2 Information Modeling Approaches
- ORM
- ER
- UML

1.3 Historical Background
- Computer language generations
- Database kinds

1.4 The Relevant Skills
- Modeling
- Querying

1.5 Summary

1.1 Information Modeling

It's an unfortunate fact of life that names and numbers can sometimes be misinterpreted. This can prove costly, as experienced by senior citizens who had their social security benefits cut off when government agencies incorrectly pronounced them dead because of misreading "DOD" on hospital forms as "date of death" rather than the intended "date of discharge".

A more costly incident occurred in 1999 when NASA's $125 million Mars Climate Orbiter burnt up in the Martian atmosphere. Apparently, errors in its course settings arose from a failure to make a simple unit conversion. One team worked in U.S. customary units and sent its data to a second team working in metric, but no conversion was made. If a man weighs 180, does he need to go on a drastic diet? No if his mass is 180 lb, but yes if it's 180 kg. *Data* by itself is not enough. What we really need is *information*, the meaning or *semantics* of the data. Since computers lack common sense, we need to pay special attention to semantics when we use computers to model some aspect of reality.

This book provides a modern introduction to database systems, with the emphasis on information modeling. At its heart is a very high level semantic approach that is *fact-oriented* in nature. If you model databases using either traditional or object-oriented approaches, you'll find that fact orientation lifts your thinking to a higher level, illuminating your current way of doing things. Even if you're a programmer rather than a database modeler, this semantic approach provides a natural and powerful way to design your data structures.

A **database** is basically a collection of related data (e.g., a company's personnel records). When interpreted by humans, a database may be viewed as a set of related *facts*—an *information base*. In the context of our semantic approach, we'll often use the popular term "database" instead of the more technical "information base". Discovering the kinds of facts that underlie a business domain, and the rules that apply to the facts, is interesting and revealing. The quality of the database *design* used to capture these facts and rules is critical. Just as a house built from a good architectural plan is more likely to be safe and convenient for living, a well-designed database simplifies the task of ensuring that its facts are correct and easy to access. Let's review some basic ideas about database systems, and then see how things can go wrong if they are poorly designed.

Each database models a business domain—we use this term to describe any area of interest, typically a part of the real world. Consider a library database. As changes occur in the library (e.g., a book is borrowed) the database is updated to reflect these changes. This task could be performed manually using a card catalog, or be automated with an online catalog, or both. Our focus is on automated databases. Sometimes these are implemented by means of special-purpose computer programs, coded in a general-purpose programming language (e.g., C#). More often, database applications are developed using a **database management system** (DBMS). This is a software system for maintaining databases and answering queries about them (e.g., DB2, Oracle, SQL Server). The same DBMS may handle many different databases.

Typical applications use a database to house the persistent data, an in-memory object model to hold transient data, and a friendly user interface for users to enter and access data. All these structures deal with information and are best derived from an information model that clearly reveals the underlying semantics of the domain. Some tools can use information models to automatically generate not just databases, but also object models and user interfaces.

If an application requires maintenance and retrieval of lots of data, a DBMS offers many advantages over manual record keeping. Data may be conveniently captured via electronic interfaces (e.g., screen forms), then quickly processed and stored compactly on disk. Many data errors can be detected automatically, and access rights to data can be enforced by the system. People can spend more time on creative design rather than on routine tasks more suited to computers. Finally, developing and documenting the application software can be facilitated by use of *computer-assisted software engineering* (CASE) tool support.

In terms of the dominant employment group, the Agricultural Age was supplanted late in the 19th century by the Industrial Age, which is now replaced by the Information Age. With the ongoing information explosion and mechanization of industry, the proportion of information workers is steadily rising. Most businesses achieve significant productivity gains by exploiting information technology. Imagine how long a newspaper firm would last if it returned to the methods used before word processing and computerized typesetting. Apart from its enabling employment opportunities, the ability to interact efficiently with information systems empowers us to exploit their information content.

Although most employees need to be familiar with information technology, there are vast differences in the amount and complexity of information management tasks required of these workers. Originally, most technical computer work was performed by computer specialists such as programmers and systems analysts. However, the advent of user-friendly software and powerful, inexpensive personal computers led to a redistribution of computing power. End users now commonly perform many information management tasks, such as spreadsheeting, with minimal reliance on professional computer experts.

This trend toward more users "driving" their own computer systems rather than relying on expert "chauffeurs" does not eliminate the need for computer specialists. There is still a need for programming in languages such as C# and Java. However, there is an increasing demand for high level skills such as modeling complex information systems.

The area of *information systems engineering* includes subdisciplines such as requirements analysis, database design, user interface design, and report writing. In one way or another, all these subareas deal with information. Since the database design phase selects the underlying structures to capture the relevant information, it is of central importance.

To highlight the **need for good database design**, let's consider the task of designing a database to store movie details such as those shown in Table 1.1. The *header* of this table is shaded to help distinguish it from the *rows of data*. Even if the header is not shaded, we do not count it as a table row. The first row of data is fictitious.

Table 1.1 An output report about some motion pictures.

Movie#	*Movie Title*	*Released*	*Director*	*Stars*
1	Cosmology	2006	Lee Lafferty	
2	Kung Fu Hustle	2004	Stephen Chow	Stephen Chow
3	The Secret Garden	1987	Alan Grint	Gennie James Barret Oliver
4	The Secret Garden	1993	Agnieszka Holland	Kate Maberly Heydon Prowse
5	The DaVinci Code	2006	Ron Howard	Tom Hanks Ian McKellen Audrey Tautou

Different movies may have the same title (e.g., *The Secret Garden*). Hence movie numbers are used to provide a simple identifier. We interpret the data in terms of facts. For example, movie 5 has the title *The DaVinci Code*, was released in 2006, was directed by Ron Howard, and starred Tom Hanks, Ian McKellen, and Audrey Tautou. Movie 1, titled *Cosmology*, had no stars (it is a documentary). This table is an *output report*. It provides one way to view the data. This might not be the same as how the data is actually stored in a database.

In Table 1.1 each *cell* (row–column slot) may contain many values. For example, Movie 3 has two stars recorded in the row 3, column 5 cell. Some databases allow a cell to contain many values like this, but in a *relational* database each table cell may hold at most one value. Since relational database systems are dominant in the industry, our implementation discussion focuses on them. How can we design a relational database to store these facts?

Suppose we use the structure shown in Table 1.2. This has one entry in each cell. Here, "?" denotes a *null* (no star is recorded for *Cosmology*). Some DBMSs display nulls differently (e.g., "<NULL>" or a blank space). To help distinguish the rows, we've included lines between them. But from now on, we'll omit lines between rows.

Table 1.2 A badly-designed relational database table.

Movie:

movieNr	*movieTitle*	*releaseYr*	*director*	*star*
1	Cosmology	2006	Lee Lafferty	?
2	Kung Fu Hustle	2004	Stephen Chow	Stephen Chow
3	The Secret Garden	1987	Alan Grint	Gennie James
3	The Secret Garden	1987	Alan Grint	Barret Oliver
4	The Secret Garden	1993	Agnieszka Holland	Kate Maberly
4	The Secret Garden	1993	Agnieszka Holland	Heydon Prowse
5	The DaVinci Code	2006	Ron Howard	Tom Hanks
5	The DaVinci Code	2006	Ron Howard	Ian McKellen
5	The DaVinci Code	2006	Ron Howard	Audrey Tautou

Each relational table must be named. Here we called the table "Movie". See if you can spot the problem with this design before reading on.

The table contains *redundant* information. For example, the facts that movie 5 is titled *The DaVinci Code*, was released in 2006, and was directed by Ron Howard are shown three times (once for each star). We might try to fix this by deleting the extra copies in the movieTitle, releaseYr, and director columns, but this artificially makes some rows special and introduces problems with nulls.

In addition to wasting space, the Table 1.2 design can lead to errors. For example, there is nothing to stop us adding a row for movie 2 with a different title (e.g., *Kung Fun*), a different release year, a different director, and another star. Our database would then be inconsistent with the business domain, where a movie has only one title and release year, and only one director is to be recorded[1].

The corrected design uses two relational tables, Movie and Starred (Figure 1.1). The table design is shown in schematic form above the populated tables. In this example, a movie may be identified either by its movie number or by the combination of its title, release year, and director. In database terminology, each of these identifiers provides a *candidate key* for the Movie table, shown here by underlining each identifier.

Movie (movieNr, movieTitle, releaseYear, director)

Starred (movieNr, star)

Movie:

movieNr	*movieTitle*	*releaseYear*	*director*
1	Cosmology	2006	Lee Lafferty
2	Kung Fu Hustle	2004	Stephen Chow
3	The Secret Garden	1987	Alan Grint
4	The Secret Garden	1993	Agnieszka Holland
5	The DaVinci Code	2006	Ron Howard

Starred:

movieNr	*star*
2	Stephen Chow
3	Gennie James
3	Barret Oliver
4	Kate Maberly
4	Heydon Prowse
5	Tom Hanks
5	Ian McKellen
5	Audrey Tautou

Figure 1.1 A relational database representation of Table 1.1.

[1]In rare cases, a movie may have multiple directors, but in this business domain we are interested in only one director per movie.

In this case, we chose movieNr as the primary way to identify movies throughout the database. This is shown here by doubly underlining the movieNr column to indicate that it is the *primary key* of the Movie table. If a table has only one candidate key, a single underline denotes the primary key.

The constraints that each movie has only one title, release year, and director are enforced by checking that each movie number occurs only once in the Movie table. The constraints that each movie must have a title, release year, and director are enforced by checking that all movies occur in the Movie table and excluding nulls from the title, release year, and director columns. In the schema, this is captured by the dotted arrow (indicating that if a movie is listed in the Starred table it must be listed in the Movie table) and by not marking any columns as optional. In relational database terms, this arrow depicts a *foreign key constraint*, where the movieNr column in the Starred table is a foreign key referencing the primary key of the Movie table. The primary key of the Starred table is the combination of its columns, indicated here by underlining.

These concepts and notations are fully explained later in the book. Even with this simple example, care is needed for database design. With complex cases, the design problem is much more challenging. The rest of this book is largely concerned with helping you to meet such challenges.

Designing databases is both a science and an art. When supported by a good method, this design process is a stimulating and intellectually satisfying activity, with tangible benefits gained from the quality of the database applications produced. The next section explains why Object-Role Modeling (ORM) is chosen as our first modeling method. Later sections provide historical background and highlight the essential communication skills. The chapter concludes with a summary and a supplementary note section, including references for further reading.

1.2 Modeling Approaches

When we design a database for a particular business domain, we create a *model* of it. Technically, the business domain being modeled is called the **universe of discourse** (UoD), since it is the universe (or world) that we are interested in discoursing (or talking) about. The UoD or business domain is typically "part" of the "real world". To build a good model requires a good understanding of the world we are modeling, and hence is a task ideally suited to people rather than machines. The main challenge is to describe the UoD clearly and precisely. Great care is required here, since errors introduced here filter through to later stages in software development. The later the errors are detected, the more expensive they are to remove.

A person who models the UoD is called a *modeler*. If we are familiar with the business domain, we may do the modeling ourselves. If not, we should consult with others who, at least collectively, understand the business domain. These people are called *domain experts* or *subject matter experts*. Modeling is a collaborative activity between the modeler and the domain expert.

Since people naturally communicate (to themselves or others) with words, pictures, and examples, the best way to arrive at a clear description of the UoD is to use *natural language*, *intuitive diagrams*, and *examples*. To simplify the modeling task, we examine the information in the smallest units possible: *one fact at a time*.

The model should first be expressed at the *conceptual* level, in concepts that people find easy to work with. Figure 1.1 depicted a model in terms of relational database structures. This is too far removed from natural language to be called conceptual. Instead, relational database structures are at the level of a *logical* data model. Other logical data models exist (e.g., network, XML schema, and object-oriented approaches), and each DBMS is aligned with at least one of these. However, in specifying a draft conceptual design, the modeler should be free of implementation concerns. It is a hard enough job already to develop an accurate model of the UoD without having to worry at the same time about how to translate the model into data structures specific to a chosen DBMS.

Implementation concerns are of course important, but should be ignored in the early stages of modeling. Once an initial conceptual design is created, it can be mapped down to a logical design in any data model we like. This flexibility also makes it easier to implement and maintain the same application on more than one kind of DBMS.

Although most applications involve processes as well as data, we'll focus on the data, because this perspective is more stable, and processes depend on the underlying data. Three information modeling approaches are discussed: Entity-Relationship modeling (ER), fact-oriented modeling, and object-oriented modeling.

Any modeling method comprises a notation as well as a procedure for using the notation to construct models. To seed the data model in a scientific way, we need examples of the kinds of data that the system is expected to manage. We call these examples *data use cases*, since they are cases of data being used by the system. They can be output reports, input screens, or forms and can present information in many ways (tables, forms, graphs, etc.). Such examples may already exist as manual or computer records. Sometimes the application is brand new, or an improved solution or adaptation is required. If needed, the modeler constructs new examples by discussing the application area with the domain expert.

As an example, suppose our information system has to output room schedules like that shown in Table 1.3. Let's look at some different approaches to modeling this. It is not important that you understand details of the different approaches at this stage. The concepts are fully explained in later chapters.

Table 1.3 A simple data use case for room scheduling.

Room	*Time*	*Activity Code*	*Activity Name*
20	Mon 9 a.m.	ORC	ORM class
20	Tue 2 p.m.	ORC	ORM class
33	Mon 9 a.m.	XQC	XQuery class
33	Fri 5 p.m.	STP	Staff party
…	…	…	…

Figure 1.2 An ER diagram for room scheduling.

Entity-Relationship modeling was introduced by Peter Chen in 1976 and is still the most widely used approach for data modeling. It pictures the world in terms of entities that have attributes and participate in relationships. Over time, many versions of ER arose. There is no single, standard ER notation.

Different versions of ER may support different concepts and may use different symbols for the same concept. Figure 1.2 uses a popular ER notation long supported by CASE tools from Oracle Corporation. Here, entity types are shown as named, soft rectangles (rounded corners). Attributes are listed below the entity type names. An octothorpe "#" indicates the attribute is a component of the primary identifier for the entity type, and an asterisk "*" means the attribute is mandatory. Here, an ellipsis "…" indicates other attributes exist but their display is suppressed.

Relationships are depicted as named lines connecting entity types. Only binary relationships are allowed, and each half of the relationship is shown either as a solid line (mandatory) or as a broken line (optional). For example, each RoomHourSlot must have a Room, but it is optional whether a Room is involved in a RoomHourSlot. A bar across one end of a relationship indicates that the relationship is a component of the primary identifier for the entity type at that end. For example, RoomHourSlot is identified by combining its hour and room. Room is identified by its room number, and Activity by its activity code.

A fork or "crow's foot" at one end of a relationship indicates that many instances of the entity type at that end may be associated (via that relationship) with the same entity instance at the other end of the relationship. The lack of a crow's foot indicates that at most one entity instance at that end is associated with any given entity instance at the other end. For example, an Activity may be allocated many RoomHourSlots, but each RoomHourSlot is booked for at most one Activity.

To its credit, this ER diagram portrays the domain in a way that is independent of the target software platform. For example, classifying a relationship end as mandatory is a conceptual issue. There is no attempt to specify here how this constraint is implemented (e.g., using mandatory columns, foreign key references, or object references). However, the ER diagram is incomplete (can you spot any missing constraints?). Moreover, the move from the data use case to the model is not obvious. While an experienced ER modeler might immediately see that an entity type is required to model RoomHourSlot, this step might be challenging to a novice modeler.

Let's see if fact-oriented modeling can provide some help. Our treatment of fact-orientation focuses on **Object-Role Modeling**. ORM began in the early 1970s as a semantic modeling approach that views the world simply in terms of *objects* (things) playing *roles* (parts in relationships). For example, you are now playing the role of reading this book, and the book is playing the role of being read. ORM has appeared in a variety of forms such as Natural-language Information Analysis Method (NIAM). The version discussed in this book is based on extensions to NIAM and is supported by industrial software tools.

Regardless of how data use cases appear, a domain expert familiar with their meaning should be able to verbalize their information content in natural language sentences. It is the modeler's responsibility to transform that informal verbalization into a formal yet natural verbalization that is clearly understood by the domain expert. These two verbalizations, one by the domain expert transformed into one by the modeler, comprise steps 1a and 1b of ORM's conceptual analysis procedure. Here we verbalize sample data as fact instances that are then abstracted to fact types. Constraints and perhaps derivation rules are then added, and themselves validated by verbalization and sample fact populations.

To get a feeling of how this works in ORM, suppose that our system is required to output reports like Table 1.3. We ask the domain expert to read off the information contained in the table, and then we rephrase this in formal English. For example, the subject matter expert might express the facts on the top row of the table as follows: Room 20 at 9 a.m. Monday is booked for the activity 'ORC' which has the name 'ORM class'.

As modelers, we rephrase this into two elementary sentences, identifying each object by a definite description: the Room numbered '20' at the HourSlot with day-hour-code 'Mon 9 a.m.' is booked for the Activity coded 'ORC'; the Activity coded 'ORC' has the ActivityName 'ORM class'. Once the domain expert agrees with this verbalization, we abstract from the fact instances to the *fact types* (i.e., the types or kinds of fact). We might then depict this structure on an ORM diagram and populate it with sample data and counter data (explained shortly) as shown in Figure 1.3.

Figure 1.3 An ORM diagram for room scheduling, with sample and counter data.

By default, *entity types* are shown in ORM as named, soft rectangles (rounded corners) and must have a reference scheme, i.e., a way for humans to refer to instances of that type. Simple reference schemes may be shown in parentheses (e.g., "(.nr)"), as an abbreviation of the relevant association, e.g., Room has RoomNr. *Value types* such as types of character strings need no reference scheme and are shown as named, dashed, soft rectangles (e.g., ActivityName).

This book uses the notation of *ORM 2* (second generation ORM), as supported by the NORMA (Neumont ORM Architect) tool, an open source plug-in to Microsoft Visual Studio .NET. The previous version of ORM, as supported by Microsoft Visio for Enterprise Architects, depicts object types as ellipses, not soft rectangles. As a configuration option, NORMA allows object types to be displayed as ellipses or hard rectangles.

Unless indicated otherwise, in this book the term "ORM" is understood to mean ORM 2. When specific reference is made to the previous version of ORM, the term "ORM 1" is used. The ORM glossary at the end of this book includes a side-by-side comparison of ORM 1 and ORM 2 notations.

In ORM, a *role* is a part played in a *fact type* (relationship or association). A relationship is shown as a named sequence of one or more role boxes, each connected to the object type whose instances play that role. Figure 1.3 includes a ternary (three-role) association, Room at HourSlot is booked for Activity, and a binary (two-role) association Activity has ActivityName.

Unlike ER, ORM makes no use of attributes in its base models. All facts are represented in terms of objects (entities or values) playing roles. Although this often leads to larger diagrams, an attribute-free approach has advantages for conceptual analysis, including simplicity, stability, and ease of validation. If you are used to modeling in ER or the Unified Modeling Language (UML) (see later), this approach may seem strange at first, but please keep an open mind about it.

ORM allows relationships of any *arity* (number of roles). Each fact type has at least one predicate reading, corresponding to one way of traversing its roles. Any number of readings may be provided for each role ordering. For a binary association, forward and inverse predicate readings may be shown separated by a slash "/". As in logic, a predicate is a sentence with object holes in it.

Mixfix notation enables the object terms to be mixed in with the predicate reading at various positions (as required in languages such as Japanese). An object placeholder is indicated by an ellipsis "…" (e.g., the ternary predicate "… at … is booked for …"). For unary postfix predicates (e.g., "… smokes") or binary infix predicates (e.g., "… has …") the ellipses may be omitted.

For each fact type, a *fact table* may be added with a sample population to help validate the constraints. Each column in a fact table is associated with one role. The lines beside the role boxes depict internal *uniqueness constraints*, indicating which roles or role combinations must have unique entries.

ORM schemas may be represented in diagrammatic or textual form, and some ORM tools can automatically transform between the two representations. Models are *validated* with domain experts in two main ways: *verbalization* and *population*.

For example, the uniqueness constraints on the ternary association verbalize as: **For each** Room **and** HourSlot**, that** Room at **that** HourSlot is booked for **at most one** Activity; **For each** HourSlot **and** Activity**, at most one** Room at **that** HourSlot is booked for **that** Activity.

The ternary fact table shows a satisfying population (each Room-HourSlot combination is unique, and each HourSlot-Activity combination is unique). The uniqueness constraints on the binary verbalize as: **Each** Activity has **at most one** ActivityName; **Each** ActivityName refers to **at most one** Activity. The 1:1 nature of this association is illustrated by the population, where each column entry occurs only once in its column.

The solid dot on Activity is a *mandatory role* constraint, indicating that each instance in the population of Activity must play that role. This verbalizes as **Each** Activity has **some** ActivityName. A role that is not mandatory is *optional*. Since sample data are not always significant, additional data (such as STM in the binary fact type) may be needed to illustrate some rules. The optionality of the other role played by Activity is shown by the absence of STM in its population.

Since ORM schemas can be specified in unambiguous sentences backed up by illustrative examples, it is not necessary for domain experts to understand the diagram notation at all. Modelers, however, find diagrams very useful for thinking about the universe of discourse.

To double check a constraint, a *counterexample* to that constraint may be presented. The counterrows appended to the fact tables test the uniqueness constraints. For instance, the first row and counterrow of the ternary indicate that room 20 at 9 a.m. Monday is booked for both the ORC and XQC activities. This challenges the constraint "**For each** Room **and** HourSlot**, that** Room at **that** HourSlot is booked for **at most one** Activity". This constraint may be recast in negative form as: **It is impossible that the same** Room at **the same** HourSlot is booked for **more than one** Activity. The counterexample provides a test case to see if this situation is actually possible.

Concrete examples help domain experts to decide whether something really is a rule. This additional validation step is very useful in cases where the domain expert's command of language suffers from imprecise or even incorrect use of logical terms (e.g., "each", "at least", "at most", "exactly", "the same", "more than", "if").

To challenge the constraint that at most one room at the same time is booked for the same activity, the first row and second counterrow of the ternary fact table in Figure 1.3 indicate that both room 20 and room 33 are used at 9 a.m. Monday for the ORC activity. Is this kind of thing possible? If it is (and for some application domains it would be) then this constraint is not a rule, in which case the constraint should be dropped and the counterrow added to the sample data.

However, if our business does not allow two rooms to be used at the same time for the same activity, then the constraint is validated and the counterexample is rejected (although it can be retained as an illustrative counterexample).

Compare Figure 1.2 with Figure 1.3. ER is often better than ORM for displaying compact overviews. However, ER models are *further removed from natural language* and may be harder for the domain expert to conceptualize. In this case, it was more natural to verbalize the first schedule fact as a ternary, but all popular ER notations with industrial support are restricted to binary (two-role) relationships.

Being only binary does not make a language less expressive, since an n-ary association ($n > 2$) may always be transformed into binaries by co-referencing or nesting (see later). However, such a transformation may introduce an object type that appears artificial to the domain expert, which can hinder communication. Wherever possible, we should try to formulate the model in a way that appears natural to the domain expert.

ER notation is *less expressive* than ORM for capturing constraints or business rules. For example, the ER notation used for Figure 1.2 was unable to express the constraint that activity names are unique or the constraint that it is impossible that more than one room at the same hour slot is booked for the same activity.

ER encourages decisions about *relative importance* at the conceptual analysis stage. Sometimes this may be seen as an advantage. For example, it is fairly natural to think of activity names as attributes of activities, and hence treat names as less important than activities themselves.

Sometimes, however, early distinctions on relative importance can be disadvantageous. For example, instead of using RoomHourSlot in Figure 1.2, we could model the room schedule information using ActivityHourSlot. Which of these choices is preferable may depend on what other kind of information we might want to record. However, because we have been forced to make a decision about this without knowing what other facts need to be recorded, we may need to change this part of the model later.

In general, if you model a feature as an attribute and find out later that you need to record something about it, you are typically forced to remodel it as an entity type or relationship because attributes can't have attributes or participate in relationships.

For instance, suppose we record phone as an attribute of Room and then later discover that we want to know which phones support voice mail. Since you rarely know what all the future information requirements will be, an attribute-based model is inherently *unstable*. Moreover, applications using the model often need to be recoded when a model feature is changed. Since ORM is essentially immune to changes like this, it offers far greater *semantic stability*.

We have already seen that ORM models facilitate validation by both verbalization and population. Attributes make it awkward to use sample data populations. Moreover, populating optional attributes introduces null values, which may be a source of confusion to nontechnical people.

In light of the aforementioned considerations, it appears that ORM's fact-oriented approach offers at least some advantages over ER modeling for conceptual analysis. This doesn't mean that you should discard ER, since it has advantages too (e.g., compact diagrams). You can have your cake and eat it too by using ORM for the initial conceptual analysis and automatically generating an ER view from it when desired.

Even if you decide to use ER throughout, ignoring the ORM notation completely, you should find that applying or adapting the modeling steps in ORM's conceptual schema design procedure to the ER notation will help you design better ER models.

Now let's consider *Object-Oriented* (OO) *modeling*, an approach that encapsulates both data and behavior within objects. Although used mainly for designing object-oriented program code, it can also be used for database design. Many object-oriented approaches exist, but by far the most influential is the **Unified Modeling Language**, which has been adopted by the Object Management Group (OMG).

Among its many diagram types, UML includes *class diagrams* to specify static data structures. Class diagrams may be used to specify operations as well as low level design decisions specific to object-oriented code (e.g., attribute visibility and association navigability). When stripped of such implementation detail, UML class diagrams may be regarded as an extended version of ER.

A UML class diagram for our example is shown in Figure 1.4. To overcome some of the problems mentioned for the ER solution, a ternary association is used for the schedule information. Because of its object-oriented focus, UML does not require conceptual identification schemes for its classes. Instead, entity instances are assumed to be identified by internal object identifiers (oids).

UML has no standard notation to signify that attribute values must be unique for their class. However, UML does allow user-defined constraints to be added in braces or notes in any language. We've added {P} to denote primary uniqueness and {U1} for an alternate uniqueness—these symbols are not standard and hence not portable. The uniqueness constraints on the ternary are captured by the 0..1 (at most one) multiplicity constraints. Here "*" is shorthand for "0..*", meaning "0 or more". Attributes are mandatory by default.

How well does this UML model support validation with the domain expert? Let's start with verbalization. Although often less than ideal, implicit use of "has" could be used to form binary sentences from the attributes, but what about the ternary? About the best we can do is something like "Booking involves Room and HourSlot and Activity"—which is pretty useless. What if we replaced the association name with a mixfix predicate, as we did in ORM, e.g., "… at … is booked for …"?

This is no use, because UML association roles (or association ends as they are now called) are not ordered. So formally we can't know if we should read the sentence type as "Room at HourSlot is booked for Activity", or "Activity at HourSlot is booked for Room" etc. This gets worse if the same class plays more than one role in the association (e.g., Person introduced Person to Person).

UML requires association roles to have names (ORM allows role names, but does not require them), but role names don't form sentences, which are always ordered in natural language. UML's weakness with regard to verbalization of facts carries over into its verbalization of constraints and derivation rules.

Figure 1.4 A UML class diagram for room scheduling.

The UML specification recommends the Object Constraint Language (OCL) for formal expression of such rules, but OCL is simply too mathematical in nature to be used for validation by nontechnical domain experts. In principle, a higher level language could be designed for UML that could be automatically transformed to OCL.

Since verbalization in UML has inadequate support, let's try validation with sample populations. Not much luck here either. To begin with, attribute-based notations are almost useless for multiple instantiation and they introduce nulls into base populations, along with all their confusing properties.

UML does provide object diagrams that enable you to talk about attributed single instances of classes, but that doesn't help with multiple instantiation. For example, the 1:1 nature of the association between activity codes and names is transparent in the ORM fact table in Figure 1.3, but is harder to see by scanning several activity objects.

In principle, we could introduce fact tables to instantiate binary associations in UML, but this wouldn't work for non-binary associations. Why not? InUML you can't specify a reading direction for an association unless it's a binary. So there is no obvious connection between an association role and a fact column as there is in ORM.

The best we can do is to name each role and then use role names as headers to the fact table. However, the visual connection of the fact columns to the class diagram would be weak because of the nonlinear layout of the association roles, and the higher the arity of the association, the worse it gets.

In its favor, UML is far richer than ORM or ER in its ability to capture other aspects of application design (e.g., operations, activities, component packaging, and deployment). UML includes diagramming techniques, such as state machine and activity diagrams, to capture business processes. Any full specification of a business domain needs to address these dynamic aspects. If the application is to be implemented in object-oriented code, UML enables more precise descriptions of the programming code structures to be specified (e.g., attribute visibility and association navigability).

If we restrict our attention to conceptual data modeling, however, the ORM notation is significantly richer than ER or UML in its capacity to express business constraints on the data, as well as being far more orthogonal and less impacted by change. As a simple example, consider the output report of Table 1.4. You might like to try modeling this yourself before reading on.

Table 1.4 Another sample output report about Movies.

Movie		*Director*		*Reviewers*	
Nr	*Title*	*Name*	*Born*	*Name*	*Born*
1	The DaVinci Code	Ron Howard	US	Fred Bloggs	US
				Ann Green	US
2	Crocodile Dundee	Peter Faiman	AU	Ann Green	US
				Ima Viewer	GB
				Tom Sawme	AU
3	Star Peace	Ann Green	US	?	?
…	…	…	…	…	…

Figure 1.5 A UML class diagram for Table 1.4.

One way to model this report in UML is shown in Figure 1.5. Although the population of the sample report suggests that movie titles are unique and that a person can direct only one movie, let's assume that the domain expert confirms that this is not the case. We should adapt our sample population to illustrate this (e.g., add a new movie 4 with the same title 'Star Peace' directed by Ron Howard).

Assuming people are identified simply by their name, Movie and Person classes may be used as shown. The role names "director" and "reviewer" are used here to distinguish the two roles played by Person. Similarly, role names are provided to distinguish the roles played by Movie. In this example, all four role names are required. Association names may be used as well if desired.

Unlike Chen's original ER notation, UML binary associations are typically depicted by lines without a diamond. While this is convenient, the use of diamonds in longer associations is somewhat inconsistent, and the avoidance of unary relationships is unnatural. In principle, UML does allow diamonds as an alternative notation for binary associations, but in practice this is rarely seen.

In contrast, ORM's depiction of relationships as a sequence of one or more roles, where each role is associated with a fact table column, provides a uniform, general notation that facilitates validation by both verbalization and sample populations.

The multiplicity constraints indicate that each movie has exactly one director but may have many reviewers and that a person may direct or review many movies. But there is still a missing business rule. Can you spot it?

Figure 1.6 models the same domain in ORM. Here the "◀" before "has" reverses the normal left-to-right reading direction. The rule missing from the UML model is captured graphically by the circled "X" constraint between the role-pairs comprising the "directed" and "reviewed" associations. This is called an *exclusion constraint.*

Figure 1.6 ORM model for Table 1.4, with counterexample for exclusion constraint.

This exclusion constraint verbalizes as **No** Person directed **and** reviewed **the same** Movie or, reading it the other way, **No** Movie was directed by **and** was reviewed by **the same** Person. To validate this rule with the domain expert, you should verbalize the rule and also provide a counterexample. For example, in your model is it possible for Movie 1 to be directed by Ron Howard and also reviewed by Ron Howard? Figure 1.6 includes this counterexample. If the exclusion constraint really does apply, at least one of those two facts must be wrong.

Some domain experts are happy to work with diagrams and some are not. Some are good at understanding rules in natural language and some are not. But all domain experts are good at working with concrete examples. Although it is not necessary for the domain expert to see the diagram, being able to instantiate any role directly on the diagram makes it easy for you as a modeler to think clearly about the rules.

Although UML has no graphic notation for general exclusion constraints, it does allow you to document constraints in a note attached to the relevant model elements. If a concept is already part of your modeling language, it's easier to think of it.

Since the exclusion constraint notation is not built in to the UML language, it is easy to miss the constraint in developing the model. The same thing goes for ER. In contrast, the ORM modeling procedure prompts you to consider such a constraint and allows you to visualize and capture the rule formally. An ORM tool can then map the constraint automatically into executable code to ensure that the rule is enforced in the implementation.

ORM diagrams always display *semantic domains* as object types. The ORM diagram in Figure 1.7(a) includes *role names* for birthdate and deathdate, shown in square brackets next to the relevant roles. These roles are clearly compatible, as they are both played by the object type Date. In ORM, role names may be used like attribute names in automatically generated attribute-views, as well as in rules specified in attribute style (e.g., deathdate > birthdate).

ER diagrams typically hide attribute domains. For example, the birthdate and deathdate attributes in the Barker ER model shown in Figure 1.7(b) should be based on the domain Date, but this is not represented visually. In ER, attribute domains can be listed in another document.

In UML class diagrams, attribute domains may be listed after the attribute name and multiplicity (if shown), as in Figure 1.7(c). The "[0..1]" multiplicity indicates "at most one", so the attribute is optional and single-valued. All too often in practice, only *syntactic*, or value, domains are specified (e.g., String).

Figure 1.7 ORM object types are semantic domains.

An ER diagram might show population and elevation as attributes of City, and an associated table might list the domains of these attributes simply as Integer, despite the fact that it is nonsense to equate a population with an elevation.

Conceptual object types, or semantic domains, provide the conceptual "glue" that binds the various components in the application model into a coherent picture. Even at the lower level of the relational data model, E.F. Codd, the founder of the relational model, argues that "domains are the glue that holds a relational database together" (Codd 1990, p. 45).

The object types in ORM diagrams are the semantic domains, so the *connectedness* of a model is transparent. This property of ORM also has significant advantages for conceptual queries, since a user can query the conceptual model directly by navigating through its object types to establish the relevant connections. This notion is elaborated further in later chapters.

ER and UML diagrams often fail to express relevant *constraints on, or between, attributes*. Figure 1.8 provides a simple example. Notice the circled dot over an "X" in the ORM model in Figure 1.8(a). This specifies two constraints: the dot is a mandatory constraint over the disjunction of the two roles (each truck is either bought or leased) and the "X" indicates the roles are exclusive (no truck is both bought and leased). The two constraints collectively provide an *xor* (exclusive-or) constraint (each truck plays exactly one of the roles).

Unlike most versions of ER, UML does provide an xor constraint, but only between associations. Since the UML model in Figure 1.8(b) models these two fact types as attributes instead of associations, it cannot capture the constraint graphically (other than adding a note).

Notice again how the ORM diagram reveals the semantic domains. For instance, tare may be meaningfully compared with maximum load (both are masses) but not with length. In UML this can be made explicit by appending domain names to the attributes. At various stages in the modeling process, it is helpful for the modeler to see all the relevant information in the one place.

Figure 1.8 (a) ORM model. (b) UML model, revealing less detail.

Another ORM feature is its flexible support for *subtyping*, including multiple inheritance, based on formal subtype definitions. For example, the subtype LargeUSCity may be defined as a City that is in Country 'US' and has a Population > 1000000. As discussed in a later chapter, subtype definitions provide stronger constraints than declarations about whether subtypes are exclusive or exhaustive.

In principle, because there are infinitely many kinds of constraints, a textual constraint language is often required for completeness to supplement the diagram. This is true for ER, ORM, and UML models. However, the failure of ER and UML diagrams to include standard notations for many important ORM constraints makes it harder to develop a comprehensive model or to perform *transformations* on the model.

For example, suppose that in any movie an actor may have a starring role or a supporting role but not both. This can be modeled by two fact types: *Actor* has starring role in *Movie*; *Actor* has supporting role in *Movie*. The "but not both" condition is expressed in ORM as a pair-exclusion constraint between the fact types. Alternatively, these fact types may be replaced by a single longer fact type: *Actor* in *Movie* has role of *RoleKind* {star, support}.

Transformations are rigorously controlled in ORM to ensure that constraints in one representation are captured in an alternative representation. For instance, the pair-exclusion constraint is transformed into the constraint that each Actor-Movie pair has only one RoleKind. The formal theory behind such transformations is easier to apply when the relevant constraints can be visualized.

Unlike UML and ER, ORM was built from a linguistic basis. To reap the benefits of verbalization and population for communication with and validation by domain experts, it's better to use a language that was designed with this in mind. The ORM notation is easy to learn and has been successfully taught even to high school students.

We are not arguing here that ER and UML have no value. They do. We are simply suggesting that you consider using ORM's modeling techniques, and possibly its graphic notation, to facilitate your original conceptual analysis before using an attribute-based notation such as that of ER, UML, or relational tables.

Once you have validated the conceptual model with the domain expert, you need to map it to a DBMS or program code for implementation. At this lower level, you will want to use an attribute-based model, so that you have a compact picture of how facts are grouped into implementation structures. For database applications, you will want to see the table structures, foreign key relationships, and so on. Here a relational or object-relational model offers a compact view, similar to an ER or UML model.

ORM models often take up much more space than an attribute-based model, since they show each attribute as a relationship. This is ideal for conceptual analysis, where we should validate one fact type at a time. However, for logical design, we typically group facts into attribute-based structures such as tables or classes. At the logical design stage, attribute-based models are more useful than ORM models. For example, relational schema diagrams provide a simple, compact picture of the underlying tables and foreign key constraints between them. Also, UML is well suited for the logical and physical design of object-oriented code, since it allows implementation detail on the data model (e.g., attribute visibility and association navigation) and can be used to model behavior and deployment.

Having used ER, ORM, and UML in practice, we've found that ORM often makes it easier to get the model right in the first place and to change the model as the business domain evolves. We believe in the method so strongly that we've made it the basis for much of the modeling discussion in this book. Once you understand ORM's modeling principles, you'll find it much easier to gain a proper understanding of data modeling in ER and UML.

Arguments about modeling approaches can become heated, and not everyone is as convinced of the virtues of ORM as we are. All we ask is that you look objectively at the ideas presented in this book and consider using whatever you find helpful.

Although the book focuses on ORM, it also covers data modeling in other popular notations (e.g., ER, IDEF1X, UML, and relational). These other notations have value too. Even if you decide to stay with ER or UML as your conceptual analysis approach, an insight into ORM should make you a better modeler regardless.

1.3 Some Historical Background

This section briefly overviews the evolution of computing languages for information systems and then outlines the historical development of the main kinds of logical data structures used in database systems. We begin with a simple example to illustrate how the level of a language impacts how easy it is to formulate questions.

Table 1.5 summarizes how **five generations of computing languages** might be used to request a computer to list the name, mass, and moons (if any) of each planet, assuming the information is stored in an astronomical database. The higher the generation, the closer to natural language, and usually the less you have to say. Nowadays nobody uses machine code or assembler to access databases. Most database applications are coded using fourth generation languages (4GLs), perhaps in combination with third generation languages (3GLs).

Table 1.5 Five generations of computer languages.

Generation	*Language example*	*Sample code for same task*
5	ConQuer	✓Planet **that** has ✓Mass **and possibly** is orbited by ✓Moon
4	SQL	**select** X1.planetName, X1.mass, X2.moonName **from** Planet **as** X1 **left outer join** Moon **as** X2 **on** X1.planetName = X2.planetName
3	Pascal	Two pages of instructions like: **for** i := 1 **to** n **do begin** write (planetName[i], mass[i]);
2	8086 Assembler	Many pages of instructions like: ADDI AX, 1
1	8086 machine code	Many pages of instructions like: 00000101 00000001 00000000

Third generation languages, such as C# and Java, are *procedural*, emphasizing the procedures used to carry out the task. With 3GLs we typically need to specify how to access data one record at a time. Fourth generation languages, such as SQL, are primarily *declarative* in nature: one declares *what* has to be done rather than *how* to do it. With a 4GL, a single statement can be used to perform operations on whole tables, or sets of rows, at once. Hence 4GLs are *set oriented* rather than record oriented.

Fifth generation languages (5GLs), such as ConQuer (an ORM query language), allow you to specify queries naturally, without knowing the underlying data structures used to store the information. The widespread use of fifth generation languages is still in the future.

The first database management systems were developed in the early 1960s, starting with simple file managers. Various logical data architectures have been proposed as a basis for specifying the structure of databases. In the hierarchic data model, the database schema is basically a tree of linked record types, where each record type has a different structure (unlike many trees where each node is of the same type).

Records may include one or more fields, each of which can hold only a single value. Record types are related by parent–child links (e.g., using pointers), where a parent may have many children but each child has only one parent. Hence the type structure is that of a tree, or hierarchy.

For example, in Figure 1.9 the parent record type Department has two child record types: Product and Employee. Each record type contains a sequence of named fields, shown here as boxes, and the parent–child links are shown as connecting lines. For discussion purposes, one record instance has been added below each record type. As an exercise, try reading off all the facts that are contained in this database before reading on.

To begin with, there are five facts stored in the record instances. To make these facts more obvious, Figure 1.9 includes arcs connecting the relevant fields, one arc for each fact. Although these arcs are a useful annotation, they are not part of the schema notation. If we are familiar with the business domain, we can verbalize these arcs into relationships. For example, we might verbalize the five facts as follows.

Figure 1.9 A hierarchic database schema with sample data, and fact arcs.

Department 10 is located in Building 69.
Department 10 has Budget 200000 USD.
Product ‘IS2’ has ProductName ‘InfoStar 2’.
Employee 357 has EmployeeName ‘Jones E’.
Employee 357 is of Sex ‘F’.

Are there more facts? Yes! The parent–child links encode the following two facts:

Department 10 develops Product ‘IS2’.
Department 10 employs Employee 357.

Hierarchic DBMSs such as IBM’s Information Management System can efficiently manage hierarchic structures (e.g., a file directory system). However, having to explicitly navigate over predefined record links to get at the facts can be somewhat challenging. The complexity rapidly rises if the application is not hierarchic in nature.

Suppose that the same product may be developed by more than one department. Conceptually, the Department develops Product association is now many:many. Since parent–child links are always 1:many, a workaround is needed to handle this situation. For example, to record the facts that departments 10 and 20 both develop product ‘IS2’ we could have the two department record instances point to separate copies of the record instance for product ‘IS2’.

Although the type and instance link structures are still trees, the fact that product ‘IS2’ is named ‘InfoStar 2’ now appears twice in the database. Hence, we need to control this redundancy. Moreover, while retrieving products developed by a given department is easy, retrieving all the departments that developed a product is not so easy.

The *network data model* was developed by the Conference on Data Systems and Languages (CODASYL) Database Task Group. This model is more complex than the hierarchic model. Most of the data is stored in records, a single field of which may contain a single value, a set of values, or even a set of value groups. Record types are related by owner-member links, and the graph of these connections may be a network: a record type may have many owners and also own many record types.

As in the hierarchic model, facts are stored either in records or as record links. An owner-member link between record types is restricted to a 1:many association. To handle a many:many association, such as the case discussed earlier, we might introduce a new record type (e.g., Development) with many:1 associations to the other record types (in this case, Department and Product).

In general, encoding of facts in access paths such as interrecord links complicates the management of the application and makes it less flexible. For example, some new queries will have to wait until access paths have been added for them, and internal optimization efforts can be undone as the application structure evolves.

Partly to address such problems, Dr. Edgar (“Ted”) Codd introduced a simpler model: the *relational data model*. A year after his original 1969 IBM research report on the subject, Codd published a revised version for a wider audience (Codd, 1970) where he first argued that relations should be normalized so that each data entry would be atomic—we now call this first normal form. Other normal forms were defined later.

Figure 1.10 All the facts in a relational database are stored in the tables themselves.

The most significant feature of the relational model is that *all the facts are stored in tables*, which are treated as mathematical relations. For example, Figure 1.10 shows the relational database for our sample application. Again, the database is annotated with arcs corresponding to the facts stored. Notice the extra deptNr columns in the Employee and Product tables. The facts that Department 10 employs Employee 357 and develops product 'IS2' are stored in the table rows themselves. Access paths between tables are not used to specify facts (as allowed in hierarchic or network models).

To specify queries and constraints, table columns may be associated by name. This allows *ad hoc* queries to be specified at will and simplifies management of the application. Note that constraints specified between tables are not the same as access paths.

For example, in Figure 1.11, arrows "link" the deptNr column of the Employee and Product tables to the deptNr column of the Department table. However these "links" merely express the constraints that any value in the deptNr column of the Employee and Product tables must also occur as a value in the deptNr column of the Department table. These constraints do not express employment and product development facts.

Figure 1.11 "Links" between tables express constraints, not facts.

The relational model is logically cleaner than the network and hierarchic models, but it initially had poor performance, which led to its slow acceptance. However, by the late 1980s, efficient relational systems had become commonplace. Although network and hierarchic database systems are still in use today, relational DBMSs are the preferred choice for developing most new database applications. A DBMS should ideally provide an integrated data dictionary, dynamic optimization, data security, automatic recovery, and a user-friendly interface. The main query languages used with relational databases are *SQL* (informally known as "Structured Query Language") and *QBE* (Query By Example). Many systems support both of these.

SQL has long been accepted as an international standard and is commonly used for communication of queries between different database systems. For this reason, SQL is the main query language discussed in this book.

Recently the eXtensible Markup Language (*XML*) has also become widely used for communication between different systems, but this is currently focused on sharing data for purposes such as electronic commerce and web publication. Many SQL-based DBMSs now support storing of XML data, as well as querying XML data directly using a query language such as XQuery.

Data architectures exist for *object-oriented databases* and *deductive databases*, but these have a long way to go in terms of standardization and maturity before they have a chance of widespread acceptance. Although relational systems give adequate performance for most applications, they are inefficient for some applications involving complex data structures (e.g., VLSI design).

To overcome such difficulties, many relational systems are being enhanced with object-oriented features, leading to *object relational database systems.* It appears that such extended relational systems will ensure the dominance of relational databases for the near future, with XML databases and object databases being their main competitors.

1.4 The Relevant Skills

Since relational database systems are dominant, they are the main focus of our implementation discussion, with some attention also being given to XML. Although conceptual interfaces to databases are still in their infancy, developing models at this higher level is more productive and avoids wasting time acquiring knowledge and skills that will rapidly become obsolete. Consider the impact of the electronic calculator on school mathematics curricula (e.g., removal of the general square root algorithm).

Fundamentally, there are two skills that will always be relevant to interacting with an information system. Both of these skills relate to *communicating* with the system about our particular application area. Recall that the application domain is technically known as the *universe of discourse*. The two requirements are to:

- **describe** the universe of discourse
- **query** the system about the universe of discourse

The first skill entails describing the *structure* or *design* of the UoD and describing its *content* or *population*: the structural aspect is the only challenging part of this. Obviously, the ability to clearly describe the application domain is critical if you wish to add a model to the system. Complex models should normally be prepared by expert modelers working with domain experts. The main aim of this text is to introduce you to the fundamentals of information modeling. If you master the methods discussed, you will be well on your way to becoming an expert in modeling information systems.

Issuing queries is often easy using a 4GL, but the formulation of complex queries can still be difficult. Occasionally, the ability to understand some answers given by the system requires knowledge about how the system works, especially its limitations. This book provides a conceptual basis for understanding relational structures and queries, explains the algebra behind relational query languages, and includes a solid introduction to SQL.

No matter how sophisticated the information system, if we give it the wrong picture of our UoD, we can't expect to get much sense out of it. This is one aspect of the GIGO (Garbage In Garbage Out) principle. Most problems with database applications result from bad database design. This book shows how to model information at a very high level using natural concepts. While providing due attention to popular data modeling approaches such as ER and UML, it also provides an in-depth treatment of the higher level ORM approach.

For immediate use, conceptual schemas can be mapped onto the lower level structures used by today's database systems. This mapping can be performed automatically using an appropriate CASE tool or manually using an appropriate procedure. This book discusses how to map a conceptual model to a relational database system, as well as to XML schema, and how query languages may be used to retrieve data from such systems. It also provides an overview of other related methods and modern trends.

1.5 Summary

This chapter provided a motivation for studying conceptual modeling and presented a brief historical and structural overview of information systems. Database management systems are widely used and are a major productivity tool for businesses that are information oriented. For a database to be used effectively, its data should be correct, complete, and efficiently accessed. This requires that the database is well-designed. Designing a database involves building a formal model of the business domain or universe of discourse (UoD). To do this properly requires a good understanding of the UoD and a means of specifying this understanding in a clear, unambiguous way.

Object-Role Modeling (ORM) simplifies the analysis and design process by using natural language, intuitive diagrams, and examples, and by examining the information in terms of simple, *elementary facts*. By expressing the model in terms of natural concepts, such as *objects* and *roles*, this fact-oriented method provides a truly *conceptual* approach to modeling.

Other valuable modeling approaches include Entity-Relationship (ER) modeling and object-oriented modeling. In practice, ER is still the most popular, high level approach to the design of databases. While many popular versions of ER exist, the Unified Modeling Language (UML) is by far the most influential object-oriented approach.

Although ER and UML models are typically more compact than ORM models, they are arguably less suitable than ORM for formulating, transforming, or evolving a conceptual information model. ER models and UML class diagrams are further removed from natural language, lack the expressibility and simplicity of a role-based constraint notation, are less stable in the face of domain evolution, are harder to populate with fact instances, and may hide information about the semantic domains that glue the model together. However, ER and UML models better highlight the major features of the domain being modeled by representing currently less important features as attributes.

In this book, ORM is used as our basic conceptual modeling method. ER models and UML class diagrams are useful as well, especially for providing compact summaries, and are best developed as views of ORM models. For database applications, conceptual models typically need to be mapped to attribute-based logical and physical models. ER models provide designs that are closer to relational database structures. For object-oriented applications, UML models can incorporate implementation details as well as behavior and deployment aspects not covered by the ORM and ER approaches.

Programming tasks are typically coded in third generation languages such as C# and Java. Fourth generation database languages such as SQL are declarative in nature, enabling users to declare what has to be done without the fine detail of how to do it, and are set oriented rather than record oriented.

Fifth generation languages such as ConQuer enable users to query conceptual models directly. Hierarchic and network database systems store some facts in record types and some facts in links between record types. Relational database systems store all facts in tables. No matter how "intelligent" software systems become, people are needed to describe the universe of discourse and to ask the relevant questions about it.

Chapter Notes

Full bibliographic entries for references are included in the bibliography at the back of this book. Codd (1969, 1970) introduced the relational model of data. For a historical discussion of these two papers, see Date (1998). Codd (1990) suggests future directions for the relational model. The classic paper that introduced Entity-Relationship modeling is Chen (1976). Kent (2000) is a reprint of a classic book that provides many insights into the nature of information and data.

Many papers on Object-Role Modeling are accessible at http://www.orm.net and at www.ormfoundation.org. Simsion and Witt (2005) provide a readable coverage of various data modeling topics. An overview of UML can be found in Booch et. al. (1999). Muller (1999) discusses use of UML for database design. Halpin and Bloesch (1999) compare data modeling in ORM and UML. Date (2000) provides a clear introduction to most aspects of database systems.

Further background on network and hierarchic database management systems is included in Appendices C–E of Connolly and Begg (1999). Gorman (1998) provides a structural review of historical data models and argues that the modern SQL standard is a data model in its own right that is closer to some prerelational models than it is to the relational model.

2 Information Levels and Frameworks

2.1 Four Information Levels
- Conceptual
- Logical
- Physical
- External

2.2 The Conceptual Level
- Asserted fact types
- Constraints
- Derivation rules
- Transactions

2.3 Database Design Example
- From external to conceptual
- From conceptual to relational

2.4 Development Frameworks
- Information systems life cycle
- Specific frameworks

2.5 Summary

2.1 Four Information Levels

Advanced information systems are sometimes described as "intelligent". Just what intelligence is, and whether machines will ever be intelligent, are debatable questions. In the *Turing Test* of intelligence, based on a revised version of a test proposed by Alan Turing in 1950,[1] an opaque screen is placed between a typical human (see *A* in Figure 2.1) and the object being tested for intelligence. The human can communicate with the object only by means of computer (with keyboard for input and screen for output). The human may communicate in natural language about any desired topic.

If the human can't tell from the object's responses whether it is an intelligent human or a machine, then arguably the object should be classified as intelligent. Futurist Ray Kurzweil (2005) wagered $10,000 that a machine will pass the Turing Test by 2029 and feels it is likely that by 2020 a $1000 computer will reach the computational ability of the human brain (estimated at 10^{16} instructions per second).[2]

Two key conditions in the test are that natural language is used and that there are no restrictions on the discussion topics. Once we place restrictions on the language and confine the discussion to a predefined topic, we can find examples where a computer has performed at the level of a human expert (e.g., chess, diagnosis of blood diseases, mineral exploration). Such systems are called "expert systems" because they perform as well as a human expert in a specific domain. Expert systems have passed "restricted Turing Tests" specific to given universes of discourse.

Expert systems use sophisticated programs, often in conjunction with large but highly specific databases. A fifth generation information system (5GIS) is like a "user-definable" expert system in that it allows the user to enter a description of the universe of discourse and then conduct a conversation about this, all in natural language. Just how well the system handles its end of the conversation depends on how powerful its user interface, database management, and inference capabilities are.

Figure 2.1 The Turing test: can *A* distinguish between *B* and *C*?

[1]Turing's original test was based on the Imitation Game, requiring only that a computer be at least as good as a man in formulating responses intended to convey the idea that the man was actually a woman. For further discussion, see http://en.wikipedia.org/wiki/Turing_test.

[2]Kurzweil (2005) also predicts that by 2035 a personal computer will match the combined intellect of the whole human race and that by 2045 it will match one billion times this combined intellect (a profound event he calls "the singularity")! Do you think this is likely?

Although desirable, it is not necessary that a 5GIS always be able to operate at an expert level when we communicate with it. It must, however, allow us to communicate with it in a natural way. Natural languages such as English and Japanese are complex and subtle. It will be many years before an information system will be able to converse freely in unrestricted natural language. We should be content in the meantime if the system supports dialog in a formalized subset of natural language. There may be many "formal, natural languages", one for English, one for Japanese, and so on. A 5GIS should be able to respond in the same language used by the human. For example, suppose we posed the query:

What is the age of Selena?

and we received the reply

sanjusai.

This would not help unless we knew that this is Japanese for "30 years old". Even if we can translate Japanese to English, we might still misinterpret the reply, because instead of assigning people an age of zero years when born, the Japanese assign an age of one year. So an age of 30 years in the Japanese system corresponds to an age of 29 years in the Western system. In addition to the requirement for a *common language*, effective communication between two people requires that each assigns the *same meaning* to the words being used. To achieve this, they should (a) share the same context or universe of discourse and (b) speak in sentences that are unambiguous with respect to this UoD.

With our example, the confusion over whether Selena's age is 29 or 30 years results from different age conventions, one Western and one Japanese. Natural speech abounds with examples that can be disambiguated only by context. Consider "Pluto is owned by Mickey". This is true for the world of Walt Disney's cartoon characters. But suppose someone unfamiliar with Mickey Mouse and his dog Pluto interpreted this within an astronomical context, taking "Pluto" to refer to the minor planet Pluto—a drastic communication failure! It is essential to have a clear way of describing the UoD to the information system.

An information system may be viewed from *four levels*: *conceptual*, *logical*, *physical*, and *external*. The conceptual level is the most fundamental, portraying the business domain naturally in human concepts. At this level, the blueprint of the UoD is called the **conceptual schema**. This describes the *structure* or *grammar* of the business domain (e.g., what types of object populate it, what roles these play, and what constraints apply).

While the conceptual schema indicates the structure of the UoD, the **conceptual database** at any given time indicates the *content* or instances populating a specific *state* of the UoD. Since information adds meaning to the data, the term "information base" is more appropriate (van Griethuysen 1982). However, we'll use the briefer, more popular term "database". Conceptually, the database is a set of sentences expressing propositions taken to be true of the UoD. Since sentences may be added or deleted, the database may undergo transitions from one state to another. However, at

any particular time, the sentences populating the database must individually and collectively conform to the domain-specific grammar or design plan that is the conceptual schema. To summarize:

> The conceptual schema specifies the structure for all permitted states and transitions of the conceptual database.

To enforce this law we now introduce a third system component: the **conceptual information processor** (CIP). This supervises updates to the database by the user and answers user queries. Figure 2.2 shows the basic conceptual architecture of an information system. This diagram assumes that the conceptual schema is already stored in the system. For each application area, a different conceptual schema is entered.

Although the diagram may seem to suggest that the user is interacting directly with the CIP, the user's interaction with the system is external rather than conceptual. The conceptual schema is not concerned with convenient user interfaces or with the physical details of how the database can be efficiently maintained. These concerns are catered for by including external and internal components within the overall architecture.

When interpreted by people, the conceptual schema and database both provide knowledge about the UoD. The combination of conceptual schema and (conceptual) database is called a conceptual **model** of the UoD. The model or *knowledge base* is a formal description of the UoD, and the CIP controls the flow of information between the model and humans. Some authors use the term "model" in the more restricted sense of "schema". This book uses the term "model" to include both the schema and a set of facts that populate the schema.

Recall that the *domain expert* is a person or group of people collectively familiar with the business domain. The *modeler* or conceptual designer is a person or team that specifies the conceptual schema by formalizing the domain expert's knowledge. An *end user* makes use of the implemented system. For a small system, the domain expert, modeler, and user might be the same person. A large system might have several partial domain experts, a team of analysts and designers, and thousands of end users.

Figure 2.2 Information system: Conceptual level.

Figure 2.3 Access to the model.

The modeler inputs the conceptual schema to the system and has read/write access to it. Typically, most users have read-only capability for the schema, but read/write access to the database. Different interfaces might be created for different users so that some users have access to only part of the model. Thus, different users may access different subschemas of the global conceptual schema. This situation is summarized in Figure 2.3. Some users may have read-only access to the database.

An **external schema** specifies the UoD design and operations accessible to a particular user or group of users. Here we specify *what* kind of facts may be read, added, or deleted, and *how* they are displayed. For *security* reasons, different user groups are often allocated different access rights (e.g., to ensure that sensitive information is not made public). For *user convenience*, views may be constructed to hide information irrelevant to a user group, or to collect and display information more efficiently. Different interfaces may be designed to cater for users with different expertise levels even when accessing the same underlying information.

Conceptual schemas are designed for clear communication, especially between modelers and domain experts. While they give a clear picture of the UoD, they are usually converted to a lower level structure for efficient implementation. For a given application, an appropriate logical data model (e.g., relational, object-oriented) is chosen, and the conceptual schema is mapped to a **logical schema** expressed in terms of the abstract structures for data and operations supported in that data model. For example, in a relational schema facts are stored in tables, and constraints are expressed using primary and foreign key declarations and so on.

The logical schema may now be realized as a **physical schema** in a specific DBMS. For example, a relational schema might be implemented in Microsoft SQL Server or IBM's DB2. The physical schema includes details about the physical storage and access structures used in that system (indexes, file clustering, etc.). Different physical choices can be made for the same DBMS, and different DBMSs often differ in what choices are possible. Hence different physical schemas might be chosen for the same logical schema. Operations at the external level are converted by the system into operations at the physical level. Logical and physical schemas reside at the *internal level*.

One advantage of the conceptual level is that it is the most *stable* of all the levels. It is unaffected by changes in user interfaces or storage and access techniques. Suppose a conceptual schema is implemented in a relational DBMS, and later we wish to use an object database instead. Unless the UoD has changed, the conceptual schema can remain the same. We need only apply a different mapping procedure and then migrate the data.

If a language is the object of study it is said to be the *object language*. The language used to study it is then called the *metalanguage*. For example, you might use English as a metalanguage to study Japanese as an object language. An object language may be its own metalanguage, for example, English may be used to learn about English. Any conceptual schema may be expressed as a set of sentences, and hence may be viewed as a database in its own right.

This enables us to construct a *metaschema* for talking about conceptual schemas. This metaconceptual schema specifies the design rules that must be obeyed by any conceptual schema (e.g., each role is played by exactly one object type). CASE tools used to assist in designing conceptual schemas make use of such a metaschema to ensure that schemas entered by the modeler are well formed or "grammatical".

While on the subject of grammar, let us agree to accept both "schemas" and "schemata" as plural of "schema". Although "data" is the plural of "datum", we'll adopt the common practice of allowing "data" to be used in both singular and plural senses. The first exercise in the book is shown next. Answers to exercises are included in the online supplement.

Exercise 2.1

1. Classify each of the following as A (external), B (conceptual), or C (internal).
   (a) This level deals with physical details of how to efficiently store and access data.
   (b) This level is concerned with providing a convenient and authorized view of the database for an individual user.
   (c) This level is concerned with representing information in a fundamental way.

2.2 The Conceptual Level

At the conceptual level, all communication between people and the information system is handled by the conceptual information processor. This communication may be divided into three main stages:

1. The modeler enters the conceptual schema for the UoD into the system. The CIP accepts the schema if and only if it is consistent with the metaconceptual schema.

2. The user updates the database by adding or deleting specific facts. The CIP accepts an update if and only if it is consistent with the conceptual schema.

3. The user queries the system about the UoD and is answered by the CIP. The CIP can supply information about the conceptual schema or the database, provided it has stored the information or can derive it.

In these three stages, the CIP performs as a *design filter*, *data filter*, and *information supplier*, respectively. To process update and query requests, the CIP accesses the conceptual schema, which includes three main sections as shown in Figure 2.4.

A *fact* (or fact instance) is a proposition that the business takes to be true. The **fact types** section lists the kinds of facts that may be represented in the database. Fact types indicate what types of *object* are permitted in the UoD (e.g., Employee, Country), how these are *referenced* by values in the database (e.g., SSN, CountryCode), and their *relationships* (e.g., was born in, lives in).

The **constraints** section lists the constraints or restrictions on populations of the fact types. These may be static or dynamic. *Static constraints* apply to every state of the database. For example, suppose a database stores information about countries and their populations. Although a country's population may change over time, at any given time each country has at most one value recorded for its current population. *Dynamic constraints* determine what transitions are allowed between states of the database. For instance, a person's age status may change from child to adult, but not vice versa.

Constraints are also known as *validation rules* or *integrity rules*. A database is said to have integrity when it is consistent with the UoD being modeled. Although most database constraints can be represented graphically on conceptual schema diagrams, some constraints may need to be represented in other ways (e.g., using conceptual sentences, formulae, tables, graphs, or program code).

The **derivation rules** section includes rules that may be used to derive new facts from other facts. A fact that is not a *derived fact* is an *asserted fact* (also known as a primitive or base fact). Derivation rules may involve *mathematical calculation* or *logical inference*. For example, an average score may be derived by summing individual scores and dividing by the number of them. Many operators and functions are defined for various data types. This permits a large variety of possible queries without the need to document each derivation. Typical mathematical facilities include arithmetic operators, such as +, –, * (multiply), and / (divide); set operators such as ∪ (union); and functions for counting, summing, and computing averages, maxima, and minima.

In addition to such generic derivation facilities, specific *derived fact types* known to be required may be defined in the schema by means of derivation rules. Some mathematically computed fact types and almost all logically inferred fact types fit into this category. Fact types derived by use of logical inference typically involve rules that make use of logical operators such as **and** or **if**.

Fact types	*Constraints*	*Derivation rules*

Figure 2.4 The three main sections of the conceptual schema.

Although derivable facts could be stored in the database, it is usually better to avoid this. Apart from saving storage space, the practice of omitting derived facts from the stored data usually makes it much easier to deal with updates. For instance, suppose we want to regularly obtain individual ages (in years) of students in a class and, on occasion, the average age for the class. If we stored the average age in the database, we would have to arrange for this to be recomputed every time a student was added to or deleted from the class, as well as every time the age of any individual in the class increased.

As an improvement, we might store the number of students as well as their individual ages, and have the average age derived only upon request. But this is still clumsy. Can you think of a better design?

As you probably realized, there is no need to store the number of students in a class since this can be derived using a count function. This avoids having to update the class size every time someone joins or leaves the class. Moreover, there is no need to store even the individual ages of students. Computer systems have a built-in clock that enables the current date to be accessed. If we store the birth date of the students, then we can have the system derive the age of any student upon request by using an appropriate date subtraction algorithm. This way, we don't have to worry about updating ages of students as they become a year older.

Sometimes, what is required is not the current age but the age on a certain date (e.g., entrance to schooling, age grouping for a future sports competition). In such cases where a single, stable age is required for each person, it may be appropriate to store it.

Before considering a derivation example using logical inference, note the difference between *propositions* and *sentences*. Propositions are asserted by declarative sentences and are always true or false (but not both). The same proposition may be asserted by different sentences, for example, "Paris is the capital of France" and "The French capital is Paris". While humans can deal with underlying meanings in a very sophisticated way, computers are completely literally minded: they deal in sentences rather than their meanings. If we want the computer to make connections that may be obvious to us, we have to explicitly provide it with the background or rules for making such connections.

Suppose that facts about brotherhood and parenthood are stored in the database. For simplicity, assume that the only objects we want to talk about are people identified by their first names. We might now set out facts as follows:

Alan is a brother of Betty.
Charles is a brother of Betty.
Betty is a parent of Fred.

You can look at these facts and readily see that both Alan and Charles are uncles of Fred. In doing so, you're using your understanding of the term "uncle". If we want a computer system to be able to make similar deductions, we need to provide it with a rule that expresses this understanding. For example:

Given any X, Y
X is an uncle of Y **if there is some** Z **such that** X is a brother of Z **and** Z is a parent of Y.

This may be abbreviated to

X is an uncle of Y **if** X is a brother of Z **and** Z is a parent of Y.

This rule is an example of a *Horn clause*. The head of the clause on the left of "if" is derivable from the conditions stated on the right-hand side. Horn clauses are used in languages such as Prolog, and enable many derivation rules to be set out briefly. In ORM, such rules are specified using a more readable syntax, e.g.,

For each Person_1 **and** Person_2,
Person_1 is an uncle of Person_2 **if** Person_1 is a brother of **some** Person_3
who is a parent of Person_2.

To appreciate how the CIP works, let's look at an example. The notation is based on a textual ORM language called FORML (Formal ORM Language). A graphical version is explained in later chapters. For simplicity, some constraints are omitted. The UoD structure is set out in the conceptual schema shown.

Reference schemes: Person(.firstname); Country(.code); Year(CE)

Base fact types:
F1 Person lives in Country.
F2 Person was born in Year.
F3 Person is a brother of Person.
F4 Person is a parent of/is a child of Person. *Roles:* parent, child.

Constraints:
C1 **Each** Person lives in **some** Country.
C2 **Each** Person lives in **at most one** Country.
C3 **Each** Person was born in **at most one** Year.
C4 **No** Person is a brother of **itself**.
C5 **No** Person is a parent of **itself**.

Derivation rules:
D1 Person_1 is an uncle of Person_2 **if**
Person_1 is a brother of **some** Person_3 **who** is a parent of Person_2.
D2 nrChildren **of** Person = **count each** child **of that** Person.

The *reference schemes* section declares the kinds of *object* of interest and how they are referenced. Objects are either entities or values. *Entities* are real or abstract things that are referenced by relating *values* (e.g., names) to them in some way. Here we have three kinds of entity: Person, Country, and Year. In this simple UoD, people are identified by their first name (rarely true in practice!). Counties are identified by codes (e.g., 'AU' for Australia). Years are identified by CE (Common Era) values (e.g., World War II ended in 1945 CE). Years are time segments, so are entities, not values.

The *fact types* section declares which kinds of facts are of interest. This indicates how object types participate in relationships. In this book, object type names in English are shown with their first letter in capitals, and role names start with a lowercase letter. Since database columns have attribute names like "birthyear", you may feel it is better to reword "Person was born in Year" as "Person has Birthyear". However sup-

pose we add another fact type "Person has Deathyear". The formal connection between Birthyear and Deathyear is now hidden. Expressing the new fact type as "Person died in Year" reveals the semantic connections to "Person was born in Year" and makes comparisons between years of birth and death meaningful (the underlying domain is Year).

Attribute names are often used in ER, UML, and relational schemas to express facts. However, this can be unnatural (compare "Person has deathyear" with "Person died in Year"), and is incomplete unless domain names such as "Year" are added.

The *constraints* section declares constraints on the fact types. The examples here are all *static* constraints (i.e., each is true for each state of the database). Reserved words in the schema language are shown in bold. Constraint C1 means that for each person referenced in the database, we know at least one city where they live. C2 says nobody can live in more than one city (at the same time). C3 says nobody was born in more than one year; note that we might not know their year of birth. Constraint C4 says that nobody is his/her own brother, and C5 says that nobody is his/her own parent (brotherhood and parenthood is "irreflexive").

The *derivation rules* section declares a logical rule for unclehood and a function for computing the number of children of any person. The unclehood rule is specified in relational style, using fact type readings. The nrChildren rule is specified in attribute style using a role name.

At the conceptual level, each fact in the database is a simple or *elementary fact*. Basically this means it can't be split up into two or more simpler facts without loss of information. We may *add* or insert a fact into the database, and we may *delete* a fact from it. However we may not modify or change just a part of a fact. By "fact" we mean "fact *instance*" not "fact type".

The operation of adding or deleting a single fact is an *elementary update* or *simple transaction*. In our sample conceptual query language, add and delete requests start with "**add**:" and "**del**:", and queries start with "**list**" or end with "?". The CIP either accepts updates or rejects them with an indication of the violation. The CIP answers legal queries and rejects illegal ones.

To explain the conceptual notions underlying database transactions, we now discuss some examples of CIP interactions. If this seems tedious, remember that we are talking about the conceptual level, not the external level where the user actually interacts with the system. At the external level, the user typically enters, deletes, or changes values in a screen version of a form or table. Conceptually, however, we may think of such an operation being translated into the appropriate delete and add operations on elementary facts before being passed on to the CIP. Suppose we now start populating our conceptual database as follows:

User:	*CIP:*
add: Person 'Terry' lives in Country 'US'.	Accepted

Because the CIP recognizes the type of this sentence and sees that no constraints are violated, it adds it to the database and issues the reply "Accepted". If an update is in-

consistent with the conceptual schema, the CIP rejects it, indicating the reason for rejection by verbalizing the constraint(s) violated. You should now be able to follow the following dialog. To save space here, we list just the number of the constraint violated rather than its verbalization.

add: Person 'Norma' was born in Year 1950.	Rejected. C1 violated.
add: Person 'Norma' lives in Country 'US'.	Accepted.
add: Person 'Norma' was born in Year 1950.	Accepted.
add: Person 'Terry' is a brother of Person 'Paul'.	Rejected. C1 violated.
add: Person 'Paul' lives in Country 'AU'.	Accepted.
add: Person 'Terry' is a brother of Person 'Paul'.	Accepted.
add: Person 'Paul' is a brother of Person 'Paul'.	Rejected. C4 violated.
add: Person 'Terry' plays Sport 'Judo'.	Rejected. Unknown fact type.

Now suppose that Norma moves from the United States to Australia. How would we record this change? Here are two incorrect attempts:

add: Person 'Norma' lives in Country 'AU'.	Rejected. C2 violated.
del: Person 'Norma' lives in Country 'US'.	Rejected. C1 violated.

Is there any way to change Norma's country of residence? Yes. Sometimes it is convenient, or even necessary, to submit many elementary updates in a single, *compound transaction*. To show this, we use "**begin**" and "**end**" as brackets to open and close the transaction. With compound transactions, the CIP considers only the total or *collective effect* of the transaction. It is quite possible that the CIP will accept a compound transaction even if, taken individually, some of its component simple updates would be rejected. To change Norma's location we proceed thus:

begin	
del: Person 'Norma' lives in Country 'US'.	
add: Person 'Norma' lives in Country 'AU'.	
end	Accepted.

The CIP accepts a transaction if and only if its overall effect is consistent with the conceptual schema. The user entered this compound transaction as a sequence of two elementary updates. It is more natural here to specify the delete before the add operation. However, the CIP treats any compound transaction as a *set* of updates, disregarding the order of the components, and looks only at the total effect. So even if the add operation had been specified before the delete, this would have been accepted with the same result.

Note that the CIP cannot accept just part of a compound transaction: it is case of "all or nothing". Here are a few more simple interactions:

add: Person 'Linda' lives in Country 'AU'.	Accepted.
add: Person 'Selena' lives in Country 'AU'.	Accepted.
add: Person 'Terry' is a parent of Person 'Linda'.	Accepted.
add: Person 'Terry' is a parent of Person 'Selena'.	Accepted.
nrChildren **of** Person 'Terry'?	2.
list each Person **who** is a brother of Person 'Paul'.	Terry.

list each Person **who** lives in Country 'AU'.	Linda Paul Selena
list each Person **who** is an uncle of Person 'Selena'.	No Person.
add: Person 'Paul' is a brother of Person 'Terry'.	Accepted.
list each Person **who** is an uncle of Person 'Selena'.	Paul.
list each Person **who** is an aunt of Person 'Selena'.	Rejected. Unknown fact type.

Now suppose the UoD is changed by adding the following constraint:

C6 **Each** Person was born in **some** Year.

Our current database fails to meet this constraint. Suppose we start over with an empty database and try to add the following fact. How would the CIP respond?

add: Person 'Bernard' is a brother of Person 'John'.

This update request is rejected. It violates two constraints (C1 and C6), as all people listed in the database must have both their country and birth year recorded. Chapter 5 uses the terminology "mandatory roles" to describe such constraints.

In general, the order in which constraints are listed does not matter. However, if an update request violates more than one constraint, this order may determine which constraint is reported as violated if the CIP is configured to report at most one violation. In this case, the CIP would respond thus to the previous request: "Rejected. C1 violated". If the CIP is configured to report all constraint violations, it would instead respond "Rejected. C1, C6 violated". As an exercise, convince yourself that with C6 added, the following update requests are processed as shown:

add: Person 'Ken' lives in Country 'GB'.	Rejected. C6 violated.
begin	
add: Person 'Ken' lives in Country 'GB'.	
add: Person 'Ken' was born in Year 1960.	
end	Accepted.
add: Person 'Ken' was born in Year 1959.	Rejected. C3 violated.
begin	
del: Person 'Ken' was born in Year 1960.	
add: Person 'Ken' was born in Year 1959.	
add: Person 'Erik' lives in Country 'NL'.	
add: Person 'Erik' was born in Year 1970.	
add: Person 'Ken' is a brother of Person 'Erik'.	
end	Accepted.

The CIP uses the conceptual schema to supervise updates of the database and to respond to questions. Think of the designer of the conceptual schema as the *"law giver"* and the schema itself as the "*law book*" containing the laws or ground rules for the UoD. The CIP is the "*law enforcer*", as it ensures that these laws are adhered to whenever the user tries to update the database. Like any friendly police person, the CIP is also there to provide information on request.

Whenever we communicate to a person or an information system, we have in mind a particular universe of discourse. Typically, this concerns some small part of the real universe, such as a particular business. In rare cases, we might choose a fictional UoD (e.g., one populated by comic book characters) or perhaps a fantasy world we have invented for a novel that we are writing. Fictional worlds may or may not be (logically) possible.

You can often rely on your own intuitions as to what is logically possible. For instance, a world in which the Moon is colored green is possible, but a world in which the Moon is simultaneously green all over and red all over is not. A possible world is said to be *consistent* and an impossible world is *inconsistent*.

As humans, we carry prodigious amounts of information around in our minds. It is highly likely that somewhere in our personal web of beliefs some logical contradictions are lurking. In most cases it does not matter if our belief web is globally inconsistent. When reasoning about a particular UoD, however, consistency is essential. It easy to show that once you accept a logical inconsistency, it is possible to deduce anything (including rubbish) from it—an extreme case of the GIGO (Garbage In Garbage Out) principle.

There are basically two types of garbage: logical and factual. Inconsistent designs contain logical garbage. For example, we might declare two constraints that contradict one another. A good design method supported by a CASE tool that enforces metarules can help to avoid such problems. Many factual errors can be prevented by the enforcement of constraints on the database. For example, a declared constraint that "**Each** Person was born in **at most one** Country" prevents us from assigning a person more than one birth country.

Even if the schema is consistent, and the CIP checks that the database is consistent with this world design, it is still possible to add false data into the knowledge base. For example, if we tell the CIP that Einstein was born in France it might accept this even though in the actual world Einstein was born in Germany. If we want our knowledge base to remain factually correct, it is still our responsibility to ensure that all the sentences we enter into the database express propositions that are true of the actual world.

The following exercise provides practice with the concepts discussed in this section and introduces some further constraint types treated formally later.

Exercise 2.2

1. (a) Assuming the conceptual schema is already stored, what are the two main functions of the conceptual information processor?
   (b) What are the three main components of the conceptual schema?
   (c) "The CIP will reject a compound transaction if any of its component update operations is inconsistent with the conceptual schema". True or false?

2. Assume the following conceptual schema is stored. Constraints apply to each database state. C1 means that each person referred to in the database must have his/her fitness rating recorded there. C3 says the possible fitness values are whole numbers from 1 to 10. C4 means no person can be recorded as expert at more than one sport, and C5 says a person

can be recorded as being an expert at a sport only if that person is also recorded as playing the same sport.

Reference schemes: Person(.firstname); Sport(.name); FitnessRating(.nr)

Base fact types:
F1 Person has FitnessRating.
F2 Person plays Sport.
F3 Person is expert at Sport.

Constraints:
C1 **Each** Person has **some** FitnessRating.
C2 **Each** Person has **at most one** FitnessRating.
C3 **The possible values of** FitnessRating **are** 1 **to** 10.
C4 **Each** Person is expert at **at most one** Sport.
C5 **Each** Person **who** is expert at **some** Sport **also** plays **that** Sport.

Derivation rules:
D1 Person is a martial artist **if**
Person plays Sport 'judo' **or** Person plays Sport 'karatedo'.
D2 nrPlayers **of** Sport = **count each** Person **who** plays **that** Sport.

The database is initially empty. The user now attempts the following sequence of updates and queries. For each update, circle the letter if the update is accepted (based on the cumulative state of the database). In cases of rejection, supply a reason (e.g., state which part of the schema is violated). For queries, supply an appropriate response from the CIP.

(a) **add**: Person 'Ann' has FitnessRating 9.
(b) **add**: Person 'Fred' plays Sport 'tennis'.
(c) **add**: Person 'Bob' has FitnessRating 7.
(d) **add**: Person 'Ann' has FitnessRating 8.
(e) **add**: Person 'Chris' has FitnessRating 7.
(f) **add**: Person 'Fred' has FitnessRating 15.
(g) **add**: Person 'Ann' plays Sport 'judo'.
(h) **add**: Person 'Bob' is expert at Sport 'soccer'.
(i) **add**: Person 'Ann' is expert at Sport 'judo'.
(j) **add**: Person 'Ann' programs in Language 'SQL'.
(k) **add**: Person 'Ann' plays Sport 'soccer'.
(l) **add**: Person 'Chris' plays Sport 'karatedo'.
(m) **del**: Person 'Chris' has FitnessRating 7.
(n) **begin**
add: Person 'Bob' has FitnessRating 8;
del: Person 'Bob' has FitnessRating 7.
end
(o) **add**: Person 'Ann' is expert at Sport 'soccer'.
(p) **add**: Person 'Bob' plays Sport 'soccer'.
(q) Person 'Ann' plays Sport 'judo'?
(r) **list each** Person **who** plays Sport 'karatedo'.
(s) nrPlayers **of** Sport 'soccer'?
(t) **list each** Person **who** is a martial artist.
(u) **list possible values of** FitnessRating.
(v) **what** is the meaning of life?

3. In the following schema, constraints apply to the database, not necessarily to the real world. Although each student in reality has a marital status, it is optional to record it (e.g., some

students may wish to keep their marital status private). Constraint C1 combines two weaker constraints, since "exactly one" means "at least one (i.e., some), and at most one".

Reference schemes: Student(.firstname); Degree(.code); MaritalStatus(.name)

Base fact types: F1 Student is enrolled in Degree
F2 Student has MaritalStatus

Constraints: C1 **Each** Student is enrolled in **exactly one** Degree.
C2 **Each** Student has **at most one** MaritalStatus.
C3 **The possible values of** MaritalStatus **are** 'single', 'married', 'widowed', 'divorced'.
C4 MaritalStatus **transitions**: ("1" = "allowed")

From \ To	*single*	*married*	*widowed*	*divorced*
single	0	1	0	0
married	0	0	1	1
widowed	0	1	0	0
divorced	0	1	0	0

The database is initially empty. The user attempts the following sequence of updates and queries. For each update, circle the letter if the update is accepted; if rejected, supply a reason. Assume questions are legal, and supply an appropriate response.

(a) **add**: Student 'Fred' is enrolled in Degree 'BSc'.This level is concerned with providing
(b) **add**: Student 'Sue' has MaritalStatus 'single'.
(c) **begin**
  add: Student 'Sue' has MaritalStatus 'single'.
  add: Student 'Sue' is enrolled in Degree 'MA'.
  end
(d) **add**: Student 'Fred' is enrolled in Degree 'BA'.
(e) **add**: Student 'Fred' is studying Subject 'CS112'.
(f) **list possible values of** MaritalStatus.
(g) **add**: Student 'Bob' is enrolled in Degree 'BSc'.
(h) **add**: Student 'Sue' has MaritalStatus 'married'.
(i) **begin**
  del: Student 'Sue' has MaritalStatus 'single'.
  add: Student 'Sue' has MaritalStatus 'married'.
  end
(j) **add**: Student 'Bob' has MaritalStatus 'single'.
(k) **begin**
  del: Student 'Bob' has MaritalStatus 'single'.
  add: Student 'Bob' has MaritalStatus 'divorced'.
  end
(l) Student 'Sue' is enrolled in Degree 'BSc'?
(m) **list each** Student **who** is enrolled in Degree 'BSc'.
(n) **list each** Student **who** is enrolled in Degree 'MA'.
(o) **add**: 3 students are enrolled in Degree 'BE'.

What is the final state of the database?

4. Assume the following conceptual schema.

Reference schemes: Person(.firstname)

Base fact types: F1 Person is male.
F2 Person is female.
F3 Person is a parent of Person.

Constraints: C1 **Each** Person is male **or** is female.
C2 **No** Person is male **and** is female.
C3 -- Each person has at most 2 parents
Each Person_2 **instance occurs at most 2 times in** Person_1 is a parent of Person_2.
C4 **No** Person is a parent of **itself**.

Derivation rules: D1 Person_1 is a grandparent of Person_2 **if**
Person_1 is a parent of **some** Person_3 **who** is a parent of Person_2.

Assume the database is populated with the following data. The user now attempts the following sequence of updates and queries. Indicate the CIP's response in each case.

Males: David, Paul, Terry
Females: Alice, Chris, Linda, Norma, Selena

(a) **add**: Person 'Jim' is male.
(b) **add**: Person 'Bernie' is a parent of Person 'Terry'.
(c) **begin**
Person 'Terry' is a parent of Person 'Selena'.
Person 'Norma' is a parent of Person 'Selena'.
end
(d) **add**: Person 'David' is a parent of Person 'David'.
(e) **begin**
Person 'Norma' is a parent of Person 'Paul'.
Person 'Alice' is a parent of Person 'Terry'.
end
(f) **add**: Person 'Chris' is male.
(g) **add**: Person 'Chris' is a parent of Person 'Selena'.
(h) **what** Person is a grandparent of Person 'Selena'?

Formulate your own derivation rules for the following:

(i) X is a father of Y; (j) X is a daughter of Y; (k) X is a granddaughter of Y

5. Consider the following conceptual schema:

Ref. schemes: Employee(.nr); Department(.name); Language(.name)

Base fact types: F1 Employee works for Department.
F2 Employee speaks Language.

Constraints: C1 **Each** Employee works for **some** Department.
C2 **Each** Employee works for **at most one** Department.
C3 **Each** Employee speaks **some** Language.

(a) Provide an update sequence to add the facts that employees 101 and 102, who both speak English, work for the Health department.
(b) Invent some database populations that are inconsistent with the schema.

2.3 Database Design Example

This section provides a simple overview of how conceptual schemas may be implemented in relational database systems. The topic is covered in detail in later chapters. Consider a company that is based in an English-speaking country, but carries out some of its business overseas. Some of its employees may speak languages other than English, which can help when dealing with foreign customers. The company wishes to move from a manual to an automated personnel record system. Prototype examples of some online personnel record forms are shown in Figure 2.5. For simplicity, only a few details are displayed.

To model this small domain, we first verbalize information on these forms. At this stage we should ignore any user interface details, such as the geometric positioning of fields on the form or accelerator key choices (shown by underlined characters). Try this yourself for the first employee record before reading on.

Even at the conceptual level, there are often many correct ways to verbalize the information. Here is one way to read the relevant facts from the first record:

The Employee with employee number 006 has the EmployeeName ‘Adams, Ann’.
Employee 006 is of the Gender named ‘Female’.
Employee 006 speaks the ForeignLanguage named ‘Spanish’.

From the sample data, it appears that we could also identify employees by their name. However, this is usually not a good idea. To begin with, employees can change their name (e.g., on marriage). Moreover, in a large company it is often possible for two employees to have the same name. Let’s suppose this possibility is confirmed by the domain expert. In this case, the sample data was not significant in this regard.

We modeled the gender information of the first employee by treating Gender as an entity type with ‘Female’ as one of its possible names and associating that employee instance with that gender instance. Instead of using names (‘Male’, ‘Female’) for genders, we could use codes (e.g., ‘M’, ‘F’). The second fact would then read “Employee 006 is of Gender ‘F’” or more explicitly “The Employee with employee number 006 is of the Gender with code ‘F’”.

Figure 2.5 Personnel forms for three employees.

Because codes are more compact, and their meaning is obvious, let's use them instead of names as our primary way of referring to the genders. If ever needed, we could add another two fact instances to relate the gender codes to the full gender names (actually used in the user interface). Yet another way to model the gender information is to use unaries (e.g., Employee 006 is female). Later in the book we'll discuss how to transform between different ways of conceptualizing and give you guidelines to help you choose between alternatives.

Because, unlike gender, the names of foreign languages have no obvious or common abbreviations, we'll stay with using names instead of codes to identify languages. If you verbalize the other two employee records, you'll just find different instances of the same fact types. Abstracting from the instances to the types and adding constraints, we may now set out the conceptual schema as follows:

Reference schemes: Employee(.nr); EmployeeName(); Gender(.code); ForeignLanguage(.name)

Base Fact types:
F1 Employee has EmployeeName.
F2 Employee is of Gender.
F3 Employee speaks/is spoken by ForeignLanguage.

Constraints:
C1 **Each** Employee has **exactly one** EmployeeName.
C2 **Each** Employee is of **exactly one** Gender.
C3 **The possible values of** Gender **are** 'M', 'F'.

In setting out the reference schemes, we've abbreviated "number" as "nr". This works well with other languages such as German, as the first and last letters of their word for "Number" also form "Nr". Some people use "no" from the Italian "numero", but this can sometimes be confused with the negative "no". Some others use "num". The choice really is up to you, and many CASE tools allow you to declare the abbreviations you prefer to use. If there is any likelihood of confusion, you should avoid abbreviations at the conceptual level.

The empty parentheses after "EmployeeName" indicate that this is just a value type (in this case a character string) and hence needs no reference scheme.

To ensure that terms (e.g. "Employee") used in a model are well understood, *concept definitions*, or at least clear descriptions, of each term should be included. Such definitions need not display on the diagram, but should be accessible in model reports.

Each of constraints C1 and C2 is actually a pair of elementary constraints. The phrase "exactly one" is shorthand for "at least one and at most one". The constraints do not mention the fact type Employee speaks ForeignLanguage. Hence some employees may speak no foreign language (e.g., Cooper), some may speak many (e.g., Bond), and many may speak the same foreign language (e.g., Spanish). This "*default*" case may be explicitly declared using the following *verbalization*:

It is possible that the same Employee speaks **more than one** Language
and that
the same Language is spoken by **more than one** Employee.

The conceptual database for this schema contains three fact tables, one for each elementary fact type. This conceptual approach makes the schema easier to understand, prevents redundancy, and simplifies expression of constraints.

Once designed, a conceptual schema may be mapped to a *logical schema* that is described in terms of the generic logical data model (e.g., relational, network, or hierarchic) chosen for implementation purposes. The physical schema is then constructed by adapting the logical schema to the specific DBMS, improving performance by adding indexes, and so on. In this book, the implementation discussion focuses on relational DBMSs. A schema specified in terms of the relational data model is called a *relational database schema*, or **relational schema** for short.

A relational database contains *named tables*, divided *horizontally* into *unnamed rows*, and *vertically* into *named columns*. A row–column location is a *cell*. Each cell contains only one data *value*. Figure 2.6 shows the data from Figure 2.5 stored in a relational database. The Employee table has three rows and three columns. The Speaks table has four rows and two columns. In naming the columns, we chose to abbreviate "employee" as "emp". The number of rows of a table increases or decreases as records are added or deleted, but the number of columns stays the same.

Within a given schema, tables must have different names. Within the same table, columns must have different names. Columns are often called "fields", although this term is sometimes used for cells. Column names are also called *attribute* names. Each attribute is based on some *domain*, or pool of values from which its entries are drawn.

For example, if we added a manager column this would have the same domain as empNr. The relational concept of domain is similar to our conceptual notion of object type, but in practice most relational DBMSs support only syntactic domains (value type declarations such as smallint), not semantic domains.

In contrast to a conceptual database, a relational database allows facts of different types to be grouped into the same table. For example, Figure 2.6 groups the three elementary fact types into two tables. The Employee table caters for fact types F1 and F2, whereas the Speaks table caters for fact type F3.

The first row of the Employee table expresses two facts: the Employee with employee number 006 has the EmployeeName 'Adams, Ann'; the Employee with employee number 006 is of the Gender with code 'F'. In general, *each row of a table in a relational database contains one or more elementary facts*. A rare exception to this rule may occur when a table is used to store only existential facts (e.g., a list of country codes). This rare case is discussed later in the context of independent object types.

Employee:

empNr	*empName*	*gender*
006	Adams, Ann	F
007	Bond, James	M
008	Cooper, Susan	F

Speaks:

empNr	*foreignLanguage*
006	Spanish
007	French
007	Japanese
007	Spanish

Figure 2.6 Relational database tables for Figure 2.5.

With relational databases, care is needed in grouping fact types into tables. This book provides a simple procedure to map a conceptual schema onto a relational schema. This procedure shows that the choice of tables in Figure 2.6 is correct. If instead we grouped all three fact types into just one table, this would cause problems similar to those discussed earlier with the movie star example (Table 1.2).

Relational databases are based on the relational model of data developed by E. F. Codd. Here, the table is the only data structure. Tables with rows of data stored in the database are called *base tables*. Derived tables with only their definitions stored (not their content) are called *views*. Within a base table, duplicate rows are not allowed.

The *order of the rows or the columns doesn't matter*. For example, Table 2.1 reorders the columns and rows of the Employee table but the meaning is unchanged (the same six facts are present). The order of rows doesn't matter because each fact is totally contained within a single row. The reason that the order of columns doesn't matter is that each column is named, so we know what its values mean. In a sense, the columns are "ordered by their name" rather than their position.

Although the order of rows and columns is irrelevant at the logical level, at the physical level the order does exist and needs to be taken into account when tuning the database for efficient access.

Query results are also tables, but are unnamed and may contain duplicate rows. In formulating queries, you can specify the order in which the data is to be displayed, regardless of the actual order used to store the data in base tables.

We may treat a base table as a set of rows and each row as a sequence or *tuple* of data values (or, more strictly, a tuple of attribute-value pairs). In this sense, a table is a set of tuples. Since a mathematical relation is defined as a set of tuples, each table is a *relation*—hence the name "relational database".

We have seen that conceptual facts are grouped into tables in a relational schema. What about *constraints*? Relational systems typically supply only a few built-in, declarative constraint types. Columns may be specified as *mandatory* or optional for their table. In our example all columns of the Employee table are mandatory (each entry must be an actual value, not null). This goes partway toward capturing the constraints that each employee is identified by empNr, has at least one name, and is of at least one gender. We additionally require that each employee listed in the Speaks table is also listed in the Employee table (this is a *subset* or *referential integrity* constraint).

To restrict each employee to at most one name and at most one gender, entries in the empNr column of the Employee table must be *unique* (i.e., no duplicates). This may be enforced by declaring empNr as the *primary key* for the Employee table.

Table 2.1 The order of columns or rows has no impact on the meaning of the data.

Employee:

empName	*empNr*	*gender*
Cooper, Susan	008	F
Bond, James	007	M
Adams, Ann	006	F

Figure 2.7 The relational schema for Figure 2.5.

A *key* is a minimal combination of columns where no duplicates are allowed. In the relational model, each table has a primary key that provides the primary way of accessing its rows. Finally, the constraint on gender values may be enforced by a simple *check clause*.

The relational schema is set out compactly in Figure 2.7. Column names are placed in parentheses (round brackets) after the table name, and *keys are underlined*. In this notation, columns are assumed mandatory for their table by default. The dotted arrow expresses the subset constraint mentioned earlier. The possible values for gender codes are placed in braces (curly brackets). For simplicity, domains are omitted. This notation is logical, not physical. The concrete syntax for declaring the tables and constraints depends on the relational database language used (often a dialect of SQL).

The process of declaring a schema is often called data definition, and the language component specially designed for this task is called a *data definition language* (DDL). Our example was simple enough for the relational schema to be fully specified within the DDL component of standard SQL. Often, however, some constraints have to be coded separately.

A relational schema may contain table definitions, constraints, and derivation rules. Most derivation rules in a conceptual schema may be specified in a relational DBMS by defining views (virtual tables) or computed columns, or by coding triggers or stored procedures. If a query type is commonly used and requires derivation, it is explicitly included in the derivation rules section of the conceptual schema. In contrast, ad hoc queries are catered for by including derivation functions and operators in the query language itself, so that the user may express the required derivation rule within the query formulation itself.

In a relational database language such as SQL, it is fairly easy to express many mathematical and logical derivation rules (e.g., averages, or grandparenthood derived from parenthood). Some derivation rules that are difficult to express in SQL can be easily expressed in other languages. For example, the following Prolog rules recursively define ancestry in terms of parenthood:

X is an ancestor of Y **if** X is a parent of Y.
X is an ancestor of Y **if** X is a parent of Z **and** Z is an ancestor of Y.

The SQL standard supports recursive queries, but commercial SQLs differ in their support for this feature. Commercial SQLs that do not directly support recursion typically enable calls from SQL to other languages (e.g., C# or Prolog) to perform the needed recursion or other desired task.

Languages such as Prolog are used in artificial intelligence (AI) applications. In the past, AI systems tended to work with small populations of a large number of complex

fact types, and database systems worked with large populations of fewer and simpler fact types. One of the aims of next generation technology is to combine powerful inference capabilities with efficient database management.

Once the schema is defined, *data manipulation* may begin. The initially empty tables are populated with data (i.e., rows are *inserted*). Rows may then be *deleted* or *modified*. A row is modified if one or more of its values is changed: this is sometimes called "update" in the context of SQL, but we often use the term "update" generically to include insert, delete, and modify operations. Collectively, these three operations are called *data maintenance*. If the constraints are properly specified, the DBMS automatically enforces them whenever a maintenance operation occurs.

Conceptually, the row modify operation of a relational DBMS simply adds or deletes elementary facts. For example, changing 'Adams, Ann' to 'Smith, Ann' on the third row of the Employee table of Figure 2.6 simply deletes the fact that employee 006 is named 'Adams, Ann' and adds the fact that employee 006 is named 'Smith, Ann'. Of course, at the external level, the user would simply modify values in a screen version of a form (as in Figure 2.5) or a table row (as in Figure 2.6).

Another aspect of data manipulation is *data retrieval*. This typically involves three steps: the user requests information; the system searches the database and/or the schema to locate the relevant information and uses this to obtain the required result; and the system then outputs this result. The second stage may involve sorting data and/or deriving results (e.g., an average value). Unlike maintenance, retrieval operations leave the database unaltered.

Data presentation refers to special techniques for displaying information on the screen (screen display) or for producing formatted printed output (report writing). We use the term "*output report*" generically to include screen and hard copy reports. Typically, users of a relational database application do not interact directly with the relational tables in which the data is stored. Usually they enter and access the data via a *forms interface*. Issues such as design and management of screen forms, and *security* enforcement, often constitute a major part of the application development.

For reasons such as correctness, clarity, completeness, and portability, database designs should be first specified as conceptual schemas, before mapping them to actual database schemas.

2.4 Development Frameworks

Information systems development involves at least *four worlds*: the subject world, the system world, the usage world, and the development world (Jarke et al. 1992). The *subject world* is the universe of discourse, or business domain, typically part of the real world. The *system world* is the information system's model of the UoD, and hence is a formal abstraction. The *usage world* is the organizational world in which the system is to function and includes the community of users of the system, their

user interfaces to the system, associated activities, and so on. This is sometimes called the environment of discourse. The *development world* covers the environment and processes used to develop the system, including the modelers and programmers, their design methods and rationale, project schedules, and so on.

The system world is sometimes called the *information system in the narrow sense*, while the combination of the four worlds is the *information system in the broad sense*, since it includes all related human factors. Specialists who focus on the system world are called "*dry*" (they deal with formal aspects), whereas those that focus on the other three worlds are called "*wet*" (they are willing to get their hands wet dealing with the informal, softer aspects). In practice, both dry and wet approaches are needed.

Determining *automation boundaries* (how much of the subject world will be managed by the system world) is a critical decision. We should first determine the services to be provided by the business (i.e., the essential business processes). Then for each of these services we need to decide whether it will be fully automated, semi-automated, or manual.

Developing an information system for a business domain is essentially a *problem-solving* process. This general process may be broken down into four main stages: (1) defining the problem, (2) devising a plan, (3) executing the plan, and (4) evaluating what happened. Two of the most generally useful problem-solving strategies are to divide the problem into a number of *subproblems* and deal with these individually, and to try a *simpler* version of the problem first.

When the problem-solving process involves development of computer software, it may be refined into the five-stage *software life cycle* shown in Figure 2.8: *specify* (say what the software is to do), *design* (decide how to do it), *implement* (code it), *test* (check that it works), and *maintain* (keep it working). In the naïve "waterfall" version of this process, the path through these five stages is a simple sequence. In practice, however, the process is typically *cyclic* in nature, as indicated by the dotted arrows.

If a need for change is detected at a later stage of the cycle, it is often necessary to return to one of the earlier stages. Moreover, large development projects are usually best divided into small projects, which can be further divided into components, each having its own cycle. The additional cycles at the right of Figure 2.8 depict further iterations, each of which can be refined into a multistage cycle as shown on the left. In this *iterative* approach to development, the more essential or critical components are often developed first, to minimize the risk of the project failing overall.

When the software is an information system, the development cycle for each component may be refined further into the following typical phases.

Figure 2.8 The software life cycle, with iterations.

- Feasibility study
- Requirements analysis
- Conceptual design: data, processes
- Logical design: data, processes
- Basic physical design: data, processes
- Basic external design: data; processes
- Prototyping
- Completion of design
- Implementation of production version
- Testing and validation
- Release of software, documentation, and training
- Maintenance

In practice, these phases often overlap, feedback cycles are common, and some phases might be omitted. Many detailed procedures exist to flesh these phases.

A **feasibility study** identifies the main objectives of the proposed information system and determines which components may be implemented with known resources (e.g., budget allocations and staff). It examines the cost-effectiveness of alternative proposals and assigns priorities to the various system components. The cost/benefit analysis might reveal that some objectives are unrealistic or that some of the objectives can be best achieved by improved manual procedures rather than by use of computers.

Assuming the project is approved, a detailed **requirements analysis** is undertaken to determine just what the system is required to do. Components of the system are delineated, and people familiar with the relevant application areas are interviewed, including domain experts, intended users, and policy makers. Interviews may be supplemented by questionnaires. Relevant documentation is examined (e.g., forms, reports, charts, policy manuals, even code from legacy systems to be reengineered). Where no such documentation is available, the domain experts are requested to invent examples of the kinds of information to be recorded. Simple diagrams are often used to clarify how the information system is to interact with the business environment. The main operations or transactions to be supported are identified and prioritized, and estimates are made of their expected data volumes, frequencies, and required response times.

The output of this requirements collection and analysis phase is a *requirements specifications* document detailing functional requirements, nonfunctional requirements (e.g., performance), and maintenance information (e.g., anticipated changes). This document should be *unambiguous*, *complete*, *verifiable* (there is a way to check whether the requirements are satisfied), *consistent* (requirements do not contradict one another), *modifiable* (changes can be made easily and safely), *traceable* (requirements can be tracked to their origins, and are identifiable across different versions of the document), and *usable* (by current and future users of the document).

Various textual and graphical notations are used for different aspects of the requirements specification. Some process-oriented notations (e.g., use case diagrams, context diagrams, data flow diagrams) are discussed in a later chapter. For large, com-

plex projects the requirements analysis stage might take several months. As fact-oriented modeling uses verbalization of familiar examples to clarify the UoD, its conceptual design method is also useful for analysis of data requirements.

One way to measure the progress of a requirements activity employs the following three dimensions of requirements engineering: *specification* (opaque, fair, complete), *representation* (informal, semiformal, formal), and *agreement* (personal view, common view). The further one is along these dimensions, the better (Pohl, 1994).

With the understanding that phases may overlap, the next stage in the information systems life cycle is **conceptual design**. This is sometimes simply called "*analysis*", to distinguish it from the later stages of logical and physical design. With large applications, subproblems or components of a more manageable size might be selected, an *architecture* specified for coupling components, and a conceptual *subschema* designed for each. The subschemas may then be *integrated* within a *global* conceptual schema. In this text, the problems discussed are small enough for us to design the whole conceptual schema without needing to first design subschemas. However, the main design process itself is broken up into various stages. For example, fact types are identified before adding constraints.

Experienced modelers often notice similarities between new applications and previous ones they or others have designed. In this case, significant savings in the design effort may result from judicious **reuse** of strategies adopted in their earlier models. For example, if we have already modeled a university library system, and now have to model a video rental business, there are many features of the earlier model that may be reused. We might choose to identify an object type Loan in a similar way and either adopt or adapt several fact types from the earlier application. By abstracting similar, specific concepts (e.g., "book", "compact disc") to more general concepts (e.g., "rentable item"), it is easier to recognize a new application as related to earlier ones.

In recent years, a substantial and growing body of work on *design patterns* and *best practices* has been developed to facilitate reuse. Many resources are available that provide basic design patterns and templates for a variety of business domains. Large commercial packages for Enterprise Resource Planning (ERP) can be adapted to handle various functions of a business (e.g., order processing, payroll management). To widen the scope of their reuse, such packages often model practices at a very high level of abstraction, which can make them harder to apply to concrete cases. The rise of component technology, design patterns, and ERP packages has seen a move from "Build or Buy" to "Build and Buy and Reuse".

Over the years, many dimensions or perspectives have been suggested for capturing different aspects of information systems design. One classic survey of information systems design methods identified three design perspectives: data, process, and behavior (Olle et al. 1991). The *data-oriented perspective* focuses on what kinds of data are stored in the database, what constraints apply to these data, and what kinds of data are derivable. The *process-oriented perspective* examines the processes or activities performed to help understand the way a particular business operates. Processes and their agents are described, as well as information flows between processes and other components. Often, a complex process is refined into several subprocesses. The *behavior-*

oriented perspective looks at how *events* trigger actions. Often, an activity analysis may be rephrased in terms of an event analysis, or vice versa.

It is important to specify the information needed and conditions sufficient for a process to execute or "fire". For example, the process of retrieving an account balance might be triggered by the event of a client requesting an account balance, and require input of information from the client (e.g., client number and account type) as well as from the relevant database tables. Since the dividing line between process and behavior is often blurred, we'll follow one common practice of bundling any discussion of processes, events, operations, activities, and so on into a single perspective, which we'll call "process". From that viewpoint there are just *two main perspectives*: data and process. Various techniques such as *work flow models* have been developed to enable *business process modeling*, as discussed in Chapter 15.

Decades ago, systems were designed largely from the process perspective, but this proved to be inefficient and unreliable in many cases. The data perspective became dominant for information systems development, since it was more fundamental and much more stable. Business processes change continually, but the underlying data tends to undergo only minor changes by comparison. Recently, object-oriented modeling has provided one way to combine data and process perspectives, with objects encapsulating data and operations. Unfortunately, this combination is often specified at a subconceptual level, and hence this integration is not as rosy as it seems.

If designed correctly, the conceptual schema provides a formal model of the structure of the UoD. Once this semantic modeling is completed, we perform a *mapping* of the conceptual design to a **logical design** expressed in terms of the generic data model of the implementation target(s). For example, we might map a conceptual schema to a relational schema. CASE tools can perform this mapping automatically. Once the logical design is determined, a **basic physical design** can be undertaken. The logical design is adapted to the specific DBMS(s) or coding language(s) being used, and various strategies (e.g., indexes or clustering) are specified to improve the efficiency of the physical design.

The **basic external design** of an information system involves determining which kinds of data and operations will be accessible to which user groups, and designing appropriate human–computer interfaces for these groups. Typically, access rights tables are constructed for different user types, decisions are made about what functions to support on different screen forms, and the layout of screen forms and menus is decided. The external design can include cross references to the internal design (e.g., to show which forms access which tables or classes).

Except for trivial applications, the next stage of the life cycle usually involves **prototyping**. A prototype is a simplified version of the intended product that is used to gain early feedback from users on the quality of the design. It aims to cover the major functions in the requirements specification, but usually omits most of the error checking and finer details required for the final version, and uses only a small set of sample data. Because early feedback is more important than efficiency at this stage, the prototype might be coded in a higher level language than the one(s) ultimately used. For user-interface design, the initial prototypes may simply be drawn on paper: these are

called "paper prototypes". The prototypes are demonstrated to the clients, and their feedback is used to revise the requirements and designs where necessary.

Once a prototype is accepted, the (hopefully) **final design** of the actual product or component is undertaken. The **implementation** of the internal and external design is now completed by writing the actual code for the production version. The product is now subjected to extensive **testing and validation** (also known as *quality assurance*). The software is run using carefully chosen data and operations to check that it functions as expected. Selected users may then be issued with prerelease versions (e.g., alpha and one or more betas) to help find other errors. Typically, the further advanced the prerelease version, the wider the audience of test users. With some critical applications (e.g., military security systems), correctness proofs might be developed to verify that the system meets its specifications. Finally the **release** version of the product is made available to the public, along with extensive *documentation* and *training*.

Software **maintenance** involves making modifications to the software after its initial release. Maintenance may be of three main types: *corrective* (eliminate bugs); *adaptive* (alter software to cater for changes in the environment or UoD); and *perfective* (add improvements). Maintenance continues as long as the product is available. The overall development process is often *iterative*, applying the development cycle to one major component at a time rather than trying to build the whole product at once. The earlier errors are detected in the development cycle, the easier they are to fix. The cost of correcting an error at the implementation stage can be orders of magnitude greater than the cost of fixing it at the analysis stage.

Important factors in developing information systems include practicality (is automation really desirable?), correctness, clarity (designs and code should be readable and well documented), efficiency (memory requirements, speed, production costs), portability, maintainability, adaptability, user proofing, and support.

Early in the chapter we saw how information schemas can exist at four different levels (conceptual, logical, physical, and external). The inclusion of the logical level refines the original three-schema architecture developed by an international standards group. This notion of levels provides a vertical way of partitioning an information system's architecture. It is also useful to provide a horizontal partitioning. One way of doing this is to use the "six friends" Rudyard Kipling referred to in his 1902 story "The Elephant's Child": "I keep six honest serving friends … Their names are What and Why and When and How and Where and Who". If we include a column for each of these six questions and show each of the four levels as a row, we obtain a tabular, *level/perspective framework* with 24 cells.

The most influential framework to incorporate both vertical and horizontal partitioning is John Zachman's framework for information systems architecture (Zachman, 1987). It uses six focus columns for Kipling's interrogatives, but has five perspective rows instead of four levels. Hence the *Zachman framework* has 30 cells, as shown in Table 2.2. Any systems engineering project needs to consider which of these cells is relevant, as well as how to map between the levels.

Table 2.2 Zachman framework for enterprise architecture.

	Data (what)	*Function (how)*	*Network (where)*	*People (who)*	*Time (when)*	*Motivation (why)*
Scope						
Enterprise model						
System model						
Technology model						
Detailed representations						

Row 1 (the scope) provides a very high level view. Row 2 (the enterprise model) corresponds roughly to the conceptual or "owner view". Row 3 (the system model) is basically the logical level, or "designer view". Row 4 (the technology model) corresponds to most of the physical level, or "builder view", with row 5 providing more detail in the "subcontractor" view. External level aspects appear in various cells.

This book focuses on column 1 (data) of rows 2 and 3, with some discussion of column 2 (function) and row 3. In practice, projects may involve other cells in this framework. For example, we may need to consider where the system will be deployed, who the stakeholders are, when system components are due, and why some business rules need to be enforced (columns 3–6).

Table 2.3 shows an *extended four-schema framework* based on both the four-schema approach and the Zachman framework. Here the logical and physical levels are refined into two layers. The *persistent layer* deals with data that persists between application sessions: it is here that we specify the structures to store data in a database (e.g., a relational schema or an XML schema). The *transient layer* deals with structures to temporarily store data in main memory during a session: the object model may be declared at the logic level as a class diagram and at the physical level as code structures in a programming language such as C# or Java.

Table 2.3 Extended four-schema framework for system development.

	Data (what)	*Function (how)*	*Network (where)*	*People (who)*	*Time (when)*	*Motivation (why)*
Conceptual						
External						
Logical-Persistent						
Logical-Transient						
Physic-Persistent						
Physical-Transient						

Formal connections between cells should be established. For example, we should be able to map (or *forward engineer*) a conceptual model to a logical model and then a physical model, or go back the other way (*reverse engineering*).

If the application uses separate models for the transient and persistent data, then *object-relational mapping* procedures are required to map between the two layers. Having a framework like this helps us to evaluate how well we are catering for different aspects of an information system.

Many other frameworks have been proposed in industry for architecting an information system. Four of these are now mentioned briefly. Further details on these approaches may be found in the references listed in the chapter notes.

The Object Management Group Model Driven Architecture (MDA) approach includes three levels: Computation Independent Model (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM). The CIM level is partly conceptual, the PIM level is logical, and the PSM level is physical.

The Open Group Architecture Framework (TOGAF) includes four levels or domains (Business, Application, Data, Technology) and provides a detailed Architecture Development Method (ADM) that may be tailored to an organization's needs and employed to manage the execution of architecture planning activities.

The International Standards Organization (ISO) Standard Reference Model of Open Distributed Processing (RM-ODP) incorporates five viewpoints: Enterprise, Information, Computational, Engineering, and Technology (ISO 2000a,b).

The Framework of Information System Concepts (FRISCO) developed by an IFIP working group provides a taxonomy of concepts used in information systems work (Falkenberg et al. 1998). Although not widely adopted, the FRISCO report provides some useful insights into information system development.

2.5 Summary

An information system for a given application may be viewed from at least *four levels*: conceptual, logical, physical, and external. At each level the formal model or knowledge base comprises a *schema* that describes the structure or design of the UoD and a database that is populated with the fact instances. Each schema determines what states and transitions are permitted for its database(s).

The *conceptual schema* does this in terms of simple, human-oriented concepts. The *logical schema* groups information into structures supported by the generic logical architecture (e.g., relational). The *physical* schema specifies the physical storage and efficient access structures (e.g., indexes) for the specific DBMS or platform being used to implement the application. For the same global conceptual schema, different *external* schemas can be constructed for different user groups depending on what information is to be accessible and how the information is to be displayed.

A conceptual schema comprises three main sections: *fact types*, *constraints*, and *derivation rules*. Asserted or base facts are primitive (i.e., they are not derived from other facts). Derived facts are computed or inferred by applying derivation rules to other

facts. In declaring fact types, we indicate what kinds of *object* there are, how these are *referenced*, and what *roles* they play. Each role in a *relationship* is played by only one object type.

The simplest kind of object is a *value* (e.g., character string). *Entities* are real or abstract objects that are identified by their relationships to other objects (typically values). For example, a country may be identified by its name.

Static constraints determine the allowable populations of fact types. *Dynamic constraints* restrict transitions between these populations. For example, each city is the capital of at most one country, and no adult may become a child.

Each fact in a conceptual database is typically *elementary*. The addition or deletion of a fact is a simple update. In a *compound transaction*, several simple updates may be included; in this case, constraints apply only to the net effect of the complete update sequence, not to each individual update. Updates and queries on a conceptual database or conceptual schema are responded to by the conceptual information processor (CIP).

Each DBMS conforms to a *logical* data model (e.g., network, hierarchic, or relational). If a relational DBMS is chosen for the implementation, the conceptual schema is mapped to a *relational schema*, or relational database schema. Here all the stored facts are placed in named *tables*, with named columns and unnamed rows. Each row of a relational table typically corresponds to one or more elementary facts. Each cell (row–column position) contains at most one value. Within a table, no row may be duplicated. The population of a relational table may be changed by inserting, deleting, or modifying a row. Most conceptual constraints and derivation rules can be expressed either within the relational DBMS language or by interfacing to another language.

The *information systems life cycle* typically involves the following stages: feasibility study; requirements analysis; conceptual design of data and processes; logical design; basic physical design; basic external design; prototyping; implementation of production version; testing and validation; release of software, documentation, and training; and maintenance. Feedback from a later stage may indicate a need to rework earlier stages.

Careful attention to earlier stages reduces the overall effort involved, as does reuse of design strategies used previously on similar applications. For large projects, an iterative approach to development is generally recommended, where the life cycle is applied to one component before moving onto another component.

Various frameworks for information system architecture exist. The Zachman framework is composed of five rows that view the system from different perspectives (scope, enterprise model, system model, technology model, and details) and six columns with a different focus (data, function, network, people, time, and motivation).

The extended four-schema framework uses the same six columns, but has six rows for conceptual, external, logical-persistent, logical-transient, physical-persistent, and physical-transient perspectives. Other well-known system architectures include TOGAF and MDA.

Chapter Notes

The classic discussion of the conceptual, internal, and external schemas of an information system is given in van Griethuysen (ed. 1982). Some people who worked on this ISO standard also worked on the FRISCO Report (Falkenberg et al. 1998). Along with many other authors (e.g., Elmasri and Navathe 1994), we consider the relational model of data to be a *logical* model rather than a conceptual model, mainly because it is too distant from natural language. From this viewpoint, a relational schema represents the logical portion of an internal schema (omitting details specific to the chosen DBMS, and storage and access details such as indexes). Codd (1990, pp. 33–34) proposes an alternative interpretation of the three-level architecture, in which the base relations of a relational database provide the conceptual schema, and views provide an external schema.

A standard reference on requirements analysis is the ANSI/IEEE standard 830-1984. A comparative review of different approaches to information system work is given in Olle et al. (1991). An informative overview of key issues in information systems modeling is provided by Mylopoulos (1998). Pohl (1994) gives a detailed treatment of the specification, representation, and agreement dimensions of requirements engineering. Bubenko (2007) provides an historical perspective on information systems modeling.

Data model patterns for various business domains are available in a number of books, including Hay (1996, 2006) and Silverston (2001a,b). These sample models are usually expressed in one of the ER notations covered later in this book.

Zachman (1987) introduces his architectural framework, and its use in modeling repository work is soon suggested (Matthews and McGee, 1990). The Zachman Institute for Framework Advancement promotes the use of the Zachman framework in industry (see *www.zifa.com*). Hay (2003) provides an extensive discussion of the Zachman framework, and argues that some of the row descriptors should be renamed. At the DAMA International Symposium in March, 2008, John Zachman will announce a new version of his framework, called Zachman Framework[2]. As this book had to be completed before this event, no details on this second version were available at the time of writing,

Details on TOGAF may be accessed online at *www.opengroup.org/togaf/* and at *http://en.wikipedia.org/wiki/TOGAF*. For resources on the MDA approach of the OMG, see *http://www.omg.org/mda/*. A free copy of the FRISCO Report may be downloaded from *www.liacs.nl/~verrynst/frisco.html*.

This page intentionally left blank

3 Conceptual Modeling: First Steps

3.1 Conceptual Modeling Language Criteria
- Desirable features
- Language design trade-offs

3.2 Conceptual Schema Design Procedure
- Modularize
- Apply the seven-step CSDP
- Integrate

3.3 CSDP Step 1
- Verbalize information examples informally
- Translate into elementary facts
- Varieties of information

3.4 CSDP Step 2
- Draw the fact types
- Populate with sample facts

3.5 CSDP Step 3
- Check for object types to be combined
- Identify arithmetically derived fact types

3.6 Summary

3.1 Conceptual Modeling Language Criteria

Before discussing the conceptual schema design procedure of Object-Role Modeling (ORM), let's review the design principles that underlie the ORM language itself. Some of these ideas were mentioned before, but we generalize the discussion here so that you can apply the principles to evaluate modeling languages in general. As you progress through the book, consider how these principles are realized in the various languages discussed.

A modeling *method* comprises both a *language* and a *procedure* describing how to use the language to build models. This procedure is often called the *modeling process*. A language has associated *syntax* (marks), *semantics* (meaning), and *pragmatics* (use). Written languages may be graphical (diagrams) and/or textual. The terms "abstract syntax" and "concrete syntax" are sometimes used respectively to distinguish underlying concepts (e.g., object type) from their representation (e.g., named, soft rectangle).

Conceptual modeling portrays the business domain at a high level, using terms and concepts familiar to the business users, ignoring logical and physical level aspects (e.g., underlying database or programming structures used for implementation) and external level aspects (e.g., screen forms used for data entry). The following criteria drawn from various sources (van Griethuysen 1982; ter Hofstede 1993; Bloesch and Halpin 1996) provide a basis for evaluating conceptual modeling languages.

- Expressibility
- Clarity
- Simplicity and orthogonality
- Semantic stability
- Semantic relevance
- Validation mechanisms
- Abstraction mechanisms
- Formal foundation

The *expressibility* of a language is a measure of what it can be used to say. The more a language can express, the greater its expressive power. Ideally, a conceptual language should be able to completely model all details about the business domain that are conceptually relevant. This is called the *100% Principle* (van Griethuysen 1982). ORM is a method for modeling and querying an information system at the conceptual level and for mapping between conceptual and other levels. Although ORM extensions for process modeling are being researched, the main focus of ORM is information modeling (popularly known as data modeling), since the data perspective is more stable and provides a formal foundation on which operations may be defined. Overall, UML has a wider scope than ORM, since UML use case, behavior, and implementation diagrams model aspects beyond static structures. For conceptual data modeling, however, ORM's diagram notation has much greater expressive power than UML class diagrams or ER diagrams.

The *clarity* of a language is a measure of how easy it is to understand and use. To begin with, the language should be unambiguous. The more expressive a language is, the harder it is to maintain clarity. Ideally, the meaning of diagrams or textual expressions in the language should be intuitively obvious. This ideal is rarely achieved. A more realistic

goal is that the language concepts and notations should be easily learned and remembered. To meet this goal, a language should exhibit *simplicity* and *orthogonality*. By avoiding attributes, ORM's role-based notation is simplified, yet easily understood by populating it with fact instances. Orthogonality allows use of an expression wherever its meaning or value may be used. ORM's constructs were designed from the ground up to be orthogonal. For example, ORM constraints can be used and combined whenever this is meaningful. As we will see later, this is not true of languages like UML.

Semantic stability is a measure of how well models or queries expressed in the language retain their original intent in the face of changes to the business domain. The more changes we need to make to a model or query to cope with a business change, the less stable it is. Models and queries in ORM are semantically more stable than in ER or UML since they are not impacted by changes that cause attributes to be remodeled as relationships or vice versa.

Semantic relevance requires that only conceptually relevant details need be modeled. Any aspect irrelevant to the meaning (e.g., implementation choices, machine efficiency) should be avoided. This is called the *Conceptualization Principle* (van Griethuysen 1982). ORM models and queries are purely conceptual. In contrast, UML class models include implementation aspects (e.g., attribute visibility and association navigability).

Validation mechanisms are ways in which domain experts can check whether the model matches the business domain. For example, static features may be checked by verbalization and instantiation, and dynamic features may be checked by simulation. Unlike ER and UML, ORM models are always easily verbalized and populated.

Abstraction mechanisms allow unwanted details to be removed from immediate consideration. This is very important with large models (e.g., wall-size schema diagrams). Because ORM diagrams tend to be more detailed and larger than corresponding ER or UML diagrams, abstraction mechanisms are often used. For example, a global schema may be modularized into various scopes based on span or perspective (e.g., a single page of a data model or a single page of an activity model). Successive refinement may be used to decompose higher level views into more detailed views. Although not a language issue, tools can provide additional support such as layering and object zoom (see later). Such mechanisms can be used to hide detail, showing just that part of the model relevant to a user's immediate needs. With minor variations, these techniques can be applied to ORM, ER, and UML. ORM also includes an attribute abstraction procedure to generate ER and UML diagrams as views.

A formal foundation is needed to ensure unambiguity and executability (e.g., to automate the storage, verification, transformation, and simulation of models) and to allow formal proofs of equivalence and implication between models. Although ORM's richer, graphical constraint notation provides a more complete diagrammatic treatment of schema transformations, use of textual constraint languages can partly offset this advantage. For their data modeling constructs, ORM, ER, and UML may be given an adequate formal foundation.

Language design often involves a number of *trade-offs* between competing criteria. One well known trade-off is that between *expressibility* and *tractability*: the more expressive a language is, the harder it is to make it efficiently executable (Levesque 1984). Another trade-off is between *parsimony* and *convenience*. Although *ceteris paribus*, the

fewer concepts the better (cf. Occam's razor), restricting ourselves to the minimum possible number of concepts may sometimes be too inconvenient. For example, it is possible to write computer programs in pure binary using just strings of ones and zeros, but it's much easier to write programs in higher level languages such as C#. As an example from logic, it's more convenient to use several operators such as "not", "and", "or" and "if-then" even though we could use just one (e.g., "nand"). See the chapter notes for further discussion of this example.

One basic question relevant to the parsimony–convenience trade-off is whether to use the attribute concept as a primitive modeling construct. We've already argued in favor of a negative answer to this question. ORM uses relationships instead of attributes in order to facilitate capturing, validating, and evolving the conceptual schema, while still allowing compact attribute-views to be automatically derived for summary or implementation purposes. Traditional ER supports single-valued attributes, whereas UML supports both single-valued and multivalued attributes. We argue later that multivalued attributes are usually inappropriate for conceptual modeling, although they can be useful for logical and physical modeling.

The rest of this chapter provides an overview of ORM's conceptual schema design procedure and a detailed discussion of the first three steps in this procedure. The ORM approach has been used productively in industry since the mid 1970s; details of its history can be found in the chapter notes.

3.2 Conceptual Schema Design Procedure

When developing an information system, we first specify what is required and produce a design to meet these requirements. For reasons outlined earlier, we recommend first developing this design at the conceptual level using Object-Role Modeling. This entails describing the structure of the UoD formally in terms of an ORM conceptual schema. If an ORM context is understood, we'll often shorten "ORM conceptual schema" to just "conceptual schema".

The procedure for designing a conceptual schema small enough to manage as a single unit is referred to as the **Conceptual Schema Design Procedure** (CSDP). Large business domains are first divided into subareas (which may overlap). These subareas are normally prioritized to determine which ones to develop first, and a conceptual *subschema* is designed for each using the CSDP. With a team of modelers, a consensus on terminology is reached, so the same names are used for the same concepts. Later the subschemas are *integrated* or merged into a global conceptual schema that covers the whole UoD. This integration is often performed iteratively. This top-down design approach may be summarized as follows:

- Divide the universe of discourse into manageable subareas.
- Apply the CSDP to each subarea.
- Integrate the subschemas into a global conceptual schema.

For each manageably sized business domain or subarea, the conceptual schema design is performed in **seven steps**:

1. Transform familiar examples into elementary facts.
2. Draw the fact types, and apply a population check.
3. Check for entity types to be combined, and note any arithmetic derivations.
4. Add uniqueness constraints, and check the arity of fact types.
5. Add mandatory role constraints, and check for logical derivations.
6. Add value, set-comparison, and subtyping constraints.
7. Add other constraints and perform final checks.

Often all seven steps are performed for each model component as it is discussed with the domain expert, rather than applying step 1 to all components, then step 2 and so on.

The procedure begins with the analysis of examples of information to be output by, or input to, the information system. Basically, the first three steps are concerned with identifying the fact types. In later steps we add constraints to the fact types. Throughout the procedure, checks are performed to detect derived facts and to ensure that no mistakes have been made. The rest of this chapter considers the first three steps in detail.

With large business domains, the preliminary segmentation and final integration add two further stages, resulting in nine steps overall. Although the CSDP is best learned in the sequence of steps shown, in practice we might apply the steps somewhat differently. For example, we might add constraints as soon as the fact types are entered, rather than waiting for all the fact types to be entered before adding any constraints.

In the commercial world, there are many existing applications that have been implemented using lower level approaches, resulting in database designs that may be inconsistent, incomplete, inefficient, or difficult to maintain. Such systems are often poorly documented. These problems can be overcome by **reengineering** the existing applications using conceptual modeling techniques. For example, sample populations from existing database tables can be used as input to the CSDP.

Even without sample populations, an existing database schema can be *reverse engineered* to a tentative conceptual schema by using information about constraints and data types and by making assumptions about use of names. The conceptual design can then be validated and completed by communicating with a domain expert. The conceptual schema can then be *forward engineered* by applying a conceptual optimization procedure and then mapping to the target database system to provide an improved and maintainable implementation. This reengineering approach is discussed in a later chapter.

3.3 CSDP Step 1: From Examples to Elementary Facts

To specify what is required of an information system, we need to answer the question: *What kinds of information do we want from the system?* Clearly, any information to be output from the system must be stored in the system or be derivable by the system. Our first step is to begin with *familiar examples* of relevant information, and *express these in terms of elementary facts*. As a check on the quality of our work, we ask the following questions. *Are the entities well identified? Can the facts be split into smaller ones without losing information?* This constitutes step 1 of the conceptual schema design procedure.

CSDP step 1: Transform familiar examples into elementary facts.

For process modeling, it helps to begin with examples of the processes to be carried out by the system. In 1987 such process examples were coined "use cases", which nicely suggests cases of the system being used. UML recommends such use cases to drive the modeling process. Although process use cases help with designing process models, in practice the move from such use cases to data models is often somewhat arbitrary and frequently results in data models that need substantial reworking.

Business process modeling is part of the overall process of modeling a business domain. In practice, it is often helpful to begin with a high level view of the services to be provided by the system and then later refine these services into processes that describe how those services are provided. Since processes typically operate on data, however, we feel that a detailed description of those processes should wait until we have a clear understanding of the data. While the topic of business process modeling is the focus of a later chapter, our current and main concern is to ensure high fidelity data models.

If you want to get the data model right, start with examples of the *data* to be delivered by the system. By analogy with the UML term, we call these "*data use cases*", since they are cases of data being used. However this is just another name for the "familiar information examples" concept introduced to the ORM schema design procedure in the 1970s. If you ever do use UML's process use cases to drive the modeling process, you should at least flesh them out with data samples before working on the class diagrams.

If we are modeling a business domain previously managed manually or by computer, information examples are readily available. If not, we work with the domain expert to provide examples. Three types of examples are output reports, input forms, and sample queries. These might appear as tables, forms, diagrams, or text. To verbalize such examples in terms of elementary facts, we need to understand what an **elementary fact** is.

To begin with, an elementary fact is a simple assertion, or atomic proposition, about the UoD. The word "fact" indicates that the assertion is taken to be true by users in that business domain. Here "taken to be true" has the sense of epistemic commitment—the business users are prepared to act as if they believed the assertion to be true. Whether the proposition is actually true is of no concern to the system. In everyday speech, facts are true statements about the real world. However, in computing terminology we resign ourselves to the fact(!) that it is possible to have false "facts" in the database (just as we use the word "statement" for things that aren't really statements in languages like SQL).

We may think of the UoD as a set of *objects playing roles*. Elementary facts are assertions that particular objects play particular roles. The simplest kind of elementary fact asserts that a single object plays a given role. For example, consider a small domain in which people are identified by their first names. One fact about this domain might be:

1. Ann smokes.

Here we have one object (Ann) playing a role (being a smoker). Strictly, we should be more precise in identifying objects (e.g., expand "Ann" to "the person with firstname 'Ann'"); but let's tidy up later. With sentences like (1), the role played by the object is

sometimes called a *property* of the object. Here an elementary fact asserts that a certain object has a certain property. This is also called a *unary relationship*, since only one role is involved. Usually however, a relationship involves at least two roles. For example:

2. Ann employs Bob.
3. Ann employs Ann.

In (2) Ann plays the role of employer and Bob plays the role of employee. In (3) Ann is self-employed and plays both roles. In general, *an elementary fact asserts that a particular object has a property, or that one or more objects participate together in a relationship*.

Here "*elementary*" indicates the fact *cannot be "split" into smaller units of information* (with the same objects) that collectively provide the same information as the original. Elementary facts usually do not use logical connectives (e.g., **not**, **and**, **or**, **if**) or logical quantifiers (e.g., **all**, **some**). For example, sentences (4)–(9) are not elementary facts.

4. Ann smokes **and** Bob smokes.
5. Ann smokes **or** Bob smokes.
6. Ann does **not** smoke.
7. **If** Bob smokes **then** Bob is cancer prone.
8. **All** people who smoke are cancer prone.
9. **If some** person smokes **then that** person is cancer prone.

All of these sentences express information. Proposition (4) is a logical conjunction. It should be split into two elementary facts: Ann smokes; Bob smokes. Proposition (5) is a disjunction, (6) is a negation, and (7) is a conditional fact. Most database systems do not allow such information to be stored conveniently, and are incapable of making relevant inferences (e.g., deducing that Bob smokes from the combination of (5) and (6)). For most commercial applications, there is no need to store such information.

Often the absence of positive information (e.g., Ann smokes) is taken to imply the negative (Ann does not smoke): this is the usual "closed world" assumption. With an "open world" approach, negative information can be explicitly stored using negative predicates or status object types, in conjunction with suitable constraints. For example, the predicates "smokes" and "is a non-smoker" are mutually exclusive, and the fact type Person has SmokerStatus {'S', 'NS'} requires the constraint that **Each** Person has **at most one** SmokerStatus. Sometimes the choice of whether to store positive or negative information depends on which occupies less space, and the borderline between positive and negative may become blurred (e.g., consider Person dislikes Food and Patient is allergic to Drug).

Universally quantified conditionals such as (8) and (9) may be catered for by subset constraints or derivation rules. Derivation rules can be specified readily in SQL (e.g., by means of a view), and are also easily coded in languages such as Prolog. For example: cancerProne(X) **if** person(X) **and** smokes(X).

Elementary facts assert that objects play roles. How are these objects and roles specified? For now, we consider only basic objects: these are either *values or entities*. A **value** is a *constant* that is *self-identifying* in the sense that when you see the constant written down in some context you always know what is being referred to. As a result, values can be referenced directly, without needing to identify them with a description.

For now, it is sufficient to recognize two kinds of value: *character string* and *number*. Character strings are shown inside quotes (e.g., 'USA'). Numbers are denoted without quotes, typically using the usual Hindu-Arabic decimal notation (e.g., 37 or 5.2). Numbers are abstract objects denoted by character strings called numerals. For example, the number 37 is denoted by the numeral '37'. We assume that any information system supports strings and numbers as built-in data types. Values are displayed textually, but are internally represented by bit-strings.

Conceptually, an **entity** (e.g., a particular person or car) is referenced in an information system by means of a *definite description*. For example, kangaroos hop about on an entity identified as "the Country named 'Australia'". Entities may also be called "described objects". Unlike values, some, indeed most, entities can change with time. An entity may be a tangible object (e.g., the City named 'Paris') or an abstract object (e.g., the Course with code 'CS114'). We consider both entities and values to be objects that exist in the UoD. Object-oriented approaches instead use the term "object" in a more restrictive sense that approximates our use of "entity". Usually we want to talk about just the entities, but to reference them we make use of values. Sometimes we want to talk about the values themselves. Consider the following sentences:

10. Australia has six states.
11. "Australia" has nine letters.

This is a case of what logicians call the *use/mention distinction*. In (10) the word "Australia" is used to reference an entity. In (11) the word "Australia" is simply mentioned, and refers to itself. In written English language, quotes resolve this distinction. In everyday communication, entities are often referred to by a *proper name* (e.g., "Bill Clinton") or by some definite description (e.g., "the previous president of the USA", or "the president named 'Bill Clinton'"). Proper names work if we know what the name refers to from the context of utterance. For example, in (10) you probably took "Australia" to refer to the country named "Australia". However, the sentence itself does not tell you this. Perhaps (10) was about a dog named "Australia" who has six moods (sleepy, playful, hungry, etc.).

Since humans may misinterpret, and information systems lack any creativity to add context, we play it safe by demanding that entities be clearly identified by definite descriptions. To begin with, the description must specify the kind of entity being referred to: the **entity type**. A *type* is the set of all possible *instances*. Each entity is an instance of a particular entity type (e.g., Person, Country). For a given UoD, the entity type Person is the set of all people we might want to talk about during the lifetime of the information system. Note that some authors use the word "entity" for "entity type". We sometimes expand "entity" to "entity instance" to avoid any confusion. Consider the sentence

12. Lee is located in 10B.

This could be talking about a horse located in a stable, or a computer in a room, and so on. By stating the entity types involved, (13) avoids this kind of referential ambiguity. Names of object types are highlighted here by starting them with a capital letter.

13. The Patient 'Lee' is located in the Ward '10B'.

Recall the old joke: "*Question:* Did you hear about the man with the wooden leg named 'Smith'? *Answer:* No—What was the name of his other leg?" Here the responder mistook 'Smith' to refer to an entity of type WoodenLeg rather than of type Man. Sometimes, even stating the entity type fails to fully clarify things. Consider the sentence:

14. The Patient 'Lee' has a Temperature of 37.

Now imagine that the UoD contains two patients named "Lee Jones" and "Mary Lee". There is more than one person to which the label "Lee" might apply. Worse still, there may be some confusion about the units being used to state the temperature: 37 degrees Celsius is normal bodily temperature, but 37 degrees Fahrenheit is close to freezing! We resolve this ambiguity by including the **reference mode** (i.e., the manner in which the value refers to the entity). Compare the following two sentences:

15. The Patient with surname 'Lee' has a Temperature of 37 Celsius.
16. The Patient with firstname 'Lee' has a Temperature of 37 Fahrenheit.

A common way to avoid potential confusion caused by overlap of first names and surnames is to demand that fuller names be used (e.g., "Lee Jones", "Mary Lee"). In some cases, however, even these names may not be unique, and another naming convention must be employed (e.g., PatientNr). To avoid confusing "No." with the word "No", we'll use "Nr" or "#" to abbreviate "Number". Most entity designators have three components:

- *Entity type* (e.g., Patient, Temperature)
- *Reference mode* (e.g., surname, Celsius)
- *Value* (e.g., 'Lee', 37)

This is the simplest kind of entity designation scheme. We'll restrict our discussion of reference schemes to this simple case for quite some time. Composite identification schemes are considered later.

Now that we know how to specify objects, how do we specify the roles they play? We use logical **predicates**. In logic, a predicate is basically a declarative *sentence with object holes in it*. To complete the sentence, the object holes or placeholders are filled in by *object terms*. Each object term refers to a single object in the UoD. Object terms are also called singular terms, or object designators. For us, values are designated by lexical constants (sometimes preceded by the value-type name), and entities are designated by definite descriptions that relate values to entities. Consider the following sentence:

17. The Person with firstname 'Ann' *smokes*.

Here the object term is "The Person with firstname 'Ann'". For discussion purposes, the predicate identifier is shown here in *italics*. The predicate may be shown by itself as "… smokes", using an ellipsis "…" as a placeholder or "object hole" for an object instance. This is a *unary predicate*, or *sentence with one object hole* in it. It may also be called a property, or a unary relationship type.

A *binary predicate* is a *sentence with two object holes*. Most predicates in information models are binary. Consider this example:

18. The Person with firstname 'Ann' *employs* the Person with firstname 'Bob'.

Here the predicate may be shown by itself as "… employs …", with two object holes. Notice that the *order* in which the objects are placed here is important. For example, even if it is true that Ann employs Bob, it may be false that Bob employs Ann.

A *ternary predicate* is a *sentence with three object holes*. For instance, the fact that Terry worked in the Computer Science Department for 10 years involves the predicate "… worked in … for …".

An *n-ary predicate* is a sentence with n object holes ($n > 0$). Since the order is significant, a filled-in n-ary predicate is associated with a *sequence* of n object terms, not necessarily distinct. The value of n is the **arity**, or degree, of the predicate. Predicates of arity ≥ 2 are polyadic. An elementary fact asserts a proposition of the form:

$Ro_1 ... o_n$

where R is a predicate of arity n, and $o_1 ... o_n$ are n object-terms, not necessarily distinct (as in *predicate logic*). Moreover, with respect to the UoD, the proposition must not be expressible as a conjunction of simpler propositions.

For naturalness, we write predicates in *mixfix* (or distfix) form, where the terms may be mixed in with the predicate. For example, the following ternary fact uses the predicate "… moved to … during …".

19. The Scientist with surname 'Einstein' *moved to* the Country with code 'USA' *during* the Year 1933 CE.

Step 1 of the CSDP involves translating relevant information examples into sentences like this. As a simple example, consider the output report of Table 3.1. Try now to express the information in the first row in the form of elementary facts. To help with this, use the *telephone heuristic*. Imagine you have to convey the information over the telephone to someone. In performing this visual to auditory transformation, you should fully specify each entity in terms of its entity type, reference mode, and value, and also include the predicate reading.

In reports like this, the column headings and table names or captions often give a clue as to the object types and predicates. The column entries provide the values. Here is one way of translating row 1 as an elementary fact:

20. The Person with surname 'Wirth' *designed* the Language named 'Pascal'.

Notice that the entity types and reference modes appear as nouns, and the predicate as a verb phrase. This is fairly typical. In translating row 1 into the elementary fact (20), we read the row from left to right. If instead we read it from right to left, we might say:

21. The Language named 'Pascal' *was designed by* the Person with surname 'Wirth'.

Table 3.1 Some languages and their designers.

Designer	*Language*
Wirth	Pascal
Kay	Smalltalk
Wirth	Modula-2

In reversing the order of the terms, we also reversed the predicate. We speak of "was designed by" as the *inverse* of the predicate "designed". Although semantically we might regard sentences (20) and (21) as expressing the same fact, syntactically they are different. Most logicians would describe this as a case of two different sentences expressing the same proposition. Linguists like to describe this situation by saying the two sentences have different *surface structures* but the same *deep structure*.

For example, one linguistic analysis might portray the deep structure sentence as comprising a verb phrase (Design) and various noun phrases (the object terms) each of which relates to the verb in a different case (e.g., agentive for Wirth and objective for Pascal), together with a modality (past tense). Different viewpoints exist as to the "correct" way to portray deep structures (e.g., what primitives to select), and the task of translation to deep structures is often complex. In practice, most information systems can be designed without delving further into such issues.

It is important not to treat sentences like (20) and (21) as different, unrelated facts. Our approach with binary fact types is to choose one primary way of stating the predicate, but optionally allow the inverse reading to be shown as well. For example:

22. The Person with surname 'Wirth' *designed* **/** *was designed by* the Language named 'Pascal'.

Here the predicate on the left of the slash "/" is used for the left-to-right reading (20). The predicate on the right of the slash is used for the inverse reading (21). The slash visually suggests jumping over the other predicate when reading left-to-right, and jumping under the other predicate when reading right-to-left. Having different ways to talk about a fact type can help communication, and can improve constraint specification. For example, the constraint verbalization "Each Language was designed by some Person" is preferable to the equivalent "For each Language, some Person designed that Language". *N*-ary fact types have many possible orderings, but typically only one is displayed.

Now consider Table 3.2. This is a bit harder to verbalize since the columns don't have separate names. Assume that Adam and Jim are males and that Eve and Mary are females. Try to verbalize the top row in terms of elementary facts before reading on.

Perhaps you verbalized this as in (23). For completeness, the inverse is included.

23. The Person with firstname 'Terry' *is married to* / *is married to* the Person with firstname 'Norma'.

Notice that the forward predicate is the same as the inverse. This is an example of a *symmetric* relationship. Such relationships create special problems (as discussed in a later chapter). To help avoid such problems, at the conceptual level *no asserted predicate should be the same as its inverse*.

Table 3.2 An output report about marriages.

Married couples	
Terry	Norma
Tony	Gwen

Table 3.3 Origin details about some programming languages.

Designer	*Language*	*Year*
Wirth	Pascal	1971
Kay	Smalltalk	1972
Wirth	Modula-2	1979

You can always rephrase the fact to ensure this. For example, (24) does this by highlighting the different roles played by each partner.

24. The Person with firstname 'Terry' *is husband of* / *is wife of* the Person with firstname 'Norma'.

Now consider Table 3.3. This is like Table 3.1 but with an extra column added. Try to express the information on the first row in terms of elementary facts before reading on.

We might at first consider expressing this information as sentence (25), using the ternary predicate "… designed … in …". Do you see any problems with this?

25. The Person with surname 'Wirth' *designed* the Language named 'Pascal' *in* the Year 1971 CE.

Recall that an elementary fact must be simple or irreducible. It cannot be split into two or more simpler facts in the context of the UoD. The appearance of the word "and" in a sentence usually indicates that the sentence may be split into simpler facts. Here there is no "and", but "common sense" tells us that the fact can be split into the following two elementary facts with no information loss:

26. The Person with surname 'Wirth' designed the Language named 'Pascal'.
27. The Language named 'Pascal' was designed in the Year 1971 CE.

Here "no information loss" means that if we know (26) and (27) then we also know (25). The phrase "common sense" hides some formal ideas. In order to split (25) into (26) and (27) we probably relied on our implicit understanding that each language was designed in only one year. This constraint holds if we interpret "was designed in" to mean "had its design completed in". Let us agree with this interpretation.

If instead we meant "had work done on its design in" then a language may be designed in many years. In this case, we could still justify the split if each language had only one designer, or at least the same set of designers for each year. However, this might not be true. For example, if we include UML as a language, it had different designers in different years. This illustrates the need to be clear about the meaning of our wording and to strive for sample data that illustrate just what is possible.

The ORM method encourages the use of informative terms for object types and predicates. However, sometimes it is difficult to think of a term that captures the precise meaning and is also conveniently short. In such cases, if the meaning of an object type name or predicate reading could be misinterpreted, it is important to add a *descriptive comment* to clarify its meaning. Although such comments are often suppressed on ORM diagrams, modern ORM tools allow them to be captured and displayed elsewhere.

Later CSDP steps add some formal checks to detect violations of elementarity, so if our "common sense" fails us in step 1, we will normally see this error at a later stage. For now though, let's work with our intuitions. Suppose we split the ternary into the two binaries: Person designed Language; Person completed design in Year. Would this be acceptable? As an exercise, use the table's population to show that this kind of split would actually lose information.

After plenty of practice at step 1, you may wish to write the elementary facts down in abbreviated form. To start with, you can drop words such as "the" and "with" where they introduce object types and reference modes. Reference modes are placed in parentheses after the object types. A dot before the reference mode name indicates that the object type name is prepended to it in the underlying value type name. For example, Language(.name) and Country(.name) have different underlying value types LanguageName and CountryName. A colon after the reference mode name, e.g., Height(cm:), indicates unit-based measurement. The next section discusses the various kinds of reference mode in more detail.

You may shorten some identifiers for object types, reference modes, and predicates so long as the shorter readings are still meaningful. In English, start the name of object types with a capital letter. Start the name of reference modes with a small letter, unless capitals have significance (e.g., "CE"). For example, facts (26) and (27) may be set out more concisely as (26a) and (27a).

26a. Person (surname) 'Wirth' designed Language (.name) 'Pascal'.
27a. Language (.name) 'Pascal' was designed in Year (CE) 1971.

If the reference schemes for entity types are declared earlier, they may be omitted in setting out the facts. For example, (26) and (27) may be specified as:

Reference schemes: Person(surname); Language(.name); Year(CE)

Facts: Person 'Wirth' designed Language 'Pascal'.
Language 'Pascal' was designed in Year 1971.

Even more conveniently, a fact type may be displayed in diagram form (see next section), and fact instances may be entered into the relevant fact table simply by entering values.

The task of defining a formal grammar sufficient to capture any sentence expressed in natural language is daunting, partly because of the many ways in which objects may be referenced. For example, consider the sentence: "The next person to step on my toe will cop it". Some artificial intelligence research is directed toward sorting out the semantics in sentences like this.

Fortunately for us, such sentences don't appear in database tables, where simple value-based schemes are used to reference objects. ORM is capable of formally capturing the relevant semantics of any fact that can be represented in a database table. Structured object-terms and predicates provide the logical deep structure, independent of the natural language (English, Japanese, etc.) used to express the fact. By supporting ordered, mixfix predicates ORM enables this deep structure to be expressed in a surface structure in harmony with the ordered, mixfix nature of natural language. For example, consider Figure 3.1.

EmployeeNr	*Department*
37	Sales

Jugyo in	*Ka*
37	Eigyo

Figure 3.1 The same fact in English and Japanese.

Here the two tables convey the same fact in different languages. The fact may be expressed in English as (28) and in Japanese as (29). For discussion purposes, the reference modes are underlined and the predicates are italicized.

28. The Employee with <u>employeeNr</u> '37' *works in* the Department with <u>name</u> 'Sales'.
29. Jugyo in <u>jugyo in bango</u> '37' *wa* 'Eigyo' to iu <u>namae</u> no Ka *ni shozoku suru*.

These may be parsed into the structures shown in 28a and 29a, with the predicates again italicized. The sentences have the same deep structure. The infix predicate "… works in …" corresponds to the mixfix predicate "… wa … ni shozoku suru".

28a. Employee (.nr) '37' *works in* Department (.name) 'Sales'.
29a. Jugyo in (.bango) '37' *wa* Ka (.namae) 'Eigyo' *ni shozoku suru*.

Now consider the two output reports in Figure 3.2. Try to verbalize their top row in terms of one or more elementary facts. The data in Figure 3.2(a) are real, while the data in Figure 3.2(b) are fictitious but possible. You may assume that the following reference schemes are predeclared: Politician(.name); Country(.code); and Year(CE).

Although there are in principle infinitely many possible interpretations of the data, you can probably guess the intended meaning. We may read off the information for Figure 3.2 (a) and Figure 3.2(b), respectively, as shown in the following two facts.

30. The Politician 'George W. Bush' *was born in* the Country 'US' *in* the Year 1946.
31. The Politician 'George W. Bush' *visited* the Country 'GB' *in* the Year 2003.

While these facts do provide a correct interpretation of the meaning of the data on the top rows, only one of these facts is elementary. Which is it?

Fact 30 is not elementary, since it may be split into the following facts without loss of information.

(a)

Politician	*Country*	*Year*
George W. Bush	US	1946
William J. Clinton	US	1946
John Howard	AU	1939
Kevin Rudd	AU	1957

(b)

Politician	*Country*	*Year*
George W. Bush	GB	2003
George W. Bush	IE	1999
William J. Clinton	IE	1999
William J. Clinton	IE	2003
John Howard	IE	2003
John Howard	US	2003

Figure 3.2 Two reports about politicians.

32. The Politician 'George W. Bush' *was born in* the Country 'US'.
33. The Politician 'George W. Bush' *was born in* the Year 1946.

Connecting two sentences by "and" forms a logical *conjunction* of the two sentences. Intuitively, fact 30 is equivalent to the conjunction of facts 32 and 33, so fact 30 is splittable (nonelementary). Formally, this is because of a uniqueness constraint on the politician column—each politician was born in only one country and in only one year. We discuss such constraints in a later chapter and provide rules and heuristics to help determine whether a given fact is elementary. For now, however, let's just use our intuitions.

Fact 31 is elementary, because you cannot treat it as a conjunction of simpler facts without losing information. For example, if we told you that George W. Bush visited the United Kingdom (the country with country code 'GB') and that he travelled overseas in 2003, it does not follow that he visited the United Kingdom in that year.

In some cases, a ternary that is not splittable into two facts may be split into three facts. Here we might try to split fact 31 into the above two facts as well as the third fact that the United Kingdom was visited in 2003. However even these three facts don't guarantee that George W. Bush visited the United Kingdom in 2003.

Formally, the structural difference between reports (a) and (b) is their uniqueness constraint pattern. In report (a) the first column values must be unique (not duplicated). In report (b) only the whole rows must be unique (entries for single columns or any column pair may be duplicated). The next chapter discusses such patterns in detail.

As another example, consider the report shown in Table 3.4. Try step 1 yourself on this table before reading on.

One tricky feature of this table is the final column. Entries in this column are sets of degrees. Phrase your sentences to include only one degree at a time. Step 1 applied to the first row results in four facts that may be set out as follows.

34. The Lecturer with name 'Adams JB' *was born in* the Year 1946 CE.
35. The Lecturer with name 'Adams JB' *has* the Age 42 years.
36. The Lecturer with name 'Adams JB' *holds* the Degree with code 'BSc'.
37. The Lecturer with name 'Adams JB' *holds* the Degree with code 'PhD'.

Here the entity types and reference modes are Lecturer (.name), Year (CE), Age (y:), and Degree (.code). There are three fact types: Lecturer *was born in* Year; Lecturer *has* Age; and Lecturer *holds* Degree. The entity types Year and Age are semantically different. Year involves a starting point in time, whereas Age is merely a duration of time (in this case, measured in years).

A more difficult example is shown next in Table 3.5. This is an extract of a report on tutorial groups. Perform step 1 yourself on this before reading the following discussion.

Table 3.4 A report storing facts about lecturers.

Lecturer:

name	*birthyear*	*age*	*degrees*
Adams JB	1946	46	BSc, PhD
Leung X	1960	32	BE, MSc
O'Reilly TA	1946	46	BA, BSc, PhD

Table 3.5 An output report listing tutorial allocations.

Tute Group	*Time*	*Room*	*Student Nr*	*Student Name*
A	Mon. 3 p.m.	CS-718	302156 180064 278155 334067 200140	Bloggs FB Fletcher JB Jackson M Jones EP Kawamoto T
B1	Tue. 2 p.m.	E-B18	266010 348112	Anderson AB Bloggs FB
…	…	…	…	…

Verbalization of an example involves interpretation, so it is important that the kind of example is *familiar* to us or the domain expert who is assisting us with step 1. Since domain experts often lack technical expertise in modeling, we should not expect them to do all of step 1 themselves. It is sufficient if they verbalize the information correctly in their own terms. This is **step 1a**: *verbalize the information* (perhaps informally).

As modelers, we often need to refine their verbalization by ensuring that the facts are elementary and that the objects are well identified. This is **step 1b**: *verbalize the information as elementary facts*. For example, a domain expert might perform step 1a by verbalizing the information on the top row of this report as:

Tute group A meets at 3 p.m. Monday in Room CS-718.
Student 302156 belongs to group A and is named 'Bloggs FB'.

We might then perform step 1b by refining these informal sentences into the following four elementary sentences:

38. The TuteGroup with code 'A' *meets at* the Time with dhcode 'Mon 3 p.m.'.
39. The TuteGroup with code 'A' *meets in* the Room with roomNr 'CS-718'.
40. The Student with studentNr 302156 *belongs to* the TuteGroup with code 'A'.
41. The Student with studentNr 302156 *has* the StudentName 'Bloggs FB'.

Many features in this example rely on interpretation. For instance, we assumed that StudentNr and StudentName refer to students and that a student number and student name on the same row refer to the same student. We also filled in the associations as "meets at", "meets in", "belongs to", and "has". The report itself does not tell us this. We use our background familiarity with the situation to make such assumptions.

Decisions were also made about entity types and reference schemes. For example, we treated Time as an entity type referenced by a day-hour-code rather than introducing Day and Hour entity types. A similar comment applies to Room and StudentName.

StudentNr was chosen rather than StudentName to identify Student. The report helps here since StudentNr appears first (on the left) and 'Bloggs FB' appears with two different student numbers, suggesting that more than one student may have the same name. However, we are still making assumptions (e.g., that students have only one student number, or that students belong to only one group).

Figure 3.3 Drawing connections for the verbalized relationships.

Another major assumption is that each tutorial group meets only once a week. We need to know this to justify using separate facts for the time and room (rather than verbalizing this as: TuteGroup meets at Time in Room). Of course the fact that group *A* is not repeated in the report helps us with this decision, but this still assumes the sample is representative in this regard.

Since interpretation is always involved in the initial step, if we are not familiar with the example we should resolve any doubts by asking a domain expert, using examples familiar to them. Although we as modelers might be expert in expressing ourselves at a formal, type level, not all domain experts can do this. By working with examples familiar to the subject matter experts, we can tap their implicit understanding of the UoD without forcing them to abstract and express, perhaps incorrectly, the structure we seek.

Sentence (41) expresses a relationship between an entity (a student) and a value (a name). When verbalizing facts, the **value type** is stated just before the value (e.g., StudentName 'Bloggs FB'). Unlike entity terms, value terms have no reference scheme.

With many reports it is sometimes useful to draw a connection between the relevant fields as we verbalize the corresponding fact. For example, we might add links between the columns of Table 3.5 as in Figure 3.3. This informal summary of the fact types helps us see if some connections have been missed.

In addition to tables, *forms* are a common source of information examples. Consider the input form shown in Figure 3.4. Forms are used more often for input than output, but may be used for both (e.g., the personnel forms considered in the previous chapter).

CS114 Tutorial preferences form

Please complete this form to assist in tutorial allocation.
Tutorials are of 1 hour duration, and are available at these times:

Monday	Tuesday	Thursday
		10 a.m.
		11 a.m.
		12 noon
	2 p.m.	2 p.m.
3 p.m.		3 p.m.

Student number: .....

Student surname: ..... *Initials:* .....

Tutorial preference 1: .....

Tutorial preference 2: .....

Tutorial preference 3: .....

Figure 3.4 An input form for collecting tutorial preferences.

Suppose students studying the subject CS114 use the form in Figure 3.4 to indicate up to three preferences regarding which tutorial time is most suitable for them. This form is used to help decide which students are assigned to which groups, and which groups are eventually used. If this information is to be taken into account in determining tutorial allocations, it must be stored.

Let's assume that the previous output report (Table 3.5) shows the tutorial allocations for CS114, made after all the student preferences are considered. The preference input form lacks some of the information needed for the allocation report. For example, it does not show how groups are assigned to rooms and times. This helps prevent students from entering wrong data (they enter the times they prefer directly rather than indirectly through associated group codes) and allows flexibility in offering many tutorials at the same time. To perform step 1 here, we first fill out the form with some examples, as shown in Figure 3.5.

These facts may now be verbalized as Student(.nr) 302156 has Surname 'Jones'; Student 302156 has Initials 'ES'; Student 302156 has first preference at Time(.dhCode) 'Mon 3 p.m.'; Student 302156 has second preference at Time 'Thurs 11 a.m.'.

Taken individually, the output report and the input form reveal only partially the kinds of information needed for the system. In combination however, they might be enough for us to arrive at the UoD structure. If so, the pair of examples is said to be significant.

In general, a set of examples is **significant** or adequate with respect to a specific UoD only if it illustrates all the relevant sorts of information and constraints. With complex UoDs, significant example sets are rare. With our current example, if a student can be allocated to only one group then Table 3.5 is significant in this respect.

However, if more than one group can be held at the same time, Table 3.5 is not significant in this other respect. A further row is needed to show this possibility (e.g., a row indicating that group B2 meets at Tuesday 2 p.m.).

CS114 Tutorial preferences form

Please complete this form to assist in tutorial allocation.
Tutorials are of 1 hour duration, and are available at these times:

Monday	Tuesday	Thursday
		10 a.m.
		11 a.m.
		12 noon
	2 p.m.	2 p.m.
3 p.m.		3 p.m.

Student number: 302156…

Student surname: *Jones*…… *Initials:* ES…..

Tutorial preference 1: Mon 3 p.m……

Tutorial preference 2: Thurs 11 a.m……

Tutorial preference 3: ……

Figure 3.5 The tutorial preference form populated with sample data.

When extracting facts from information examples, we need to decide *which aspects should be modeled*. This helps determine the scope of the UoD. A useful heuristic is to ask the question: *Which parts may take on different values?* Look again at the input form in Figure 3.5. The header section contains information that we may or may not wish to model. The first item we see is "CS114". Is it possible to have other values in place of it, within our overall application? If the only subject of interest is CS114, then the answer is "no". However if we wish to cater for other subjects as well, we might require another form with a different value here (e.g., "CS183"). In this case we need to introduce Subject (or some equivalent term) as an object type within our UoD.

Different universities may use different terms for a unit of study. Instead of our "Subject", terms such as "Course" or "Unit" might be used. If the domain experts all prefer the same term, use that. If different people use different terms for the same concept, get them to agree upon a *standard term*, and also note any *synonyms* in use.

Returning to the header of our preferences form in Figure 3.5, we see the word "Tutorial". Could this change (e.g., to "Lecture")? If we wish to capture preferences for lectures as well as tutorials, the answer is "yes", and we could model this as data. But let's assume that this is not the case. The rest of the form header contains other information (e.g., duration of tutorials) but let's assume this doesn't need to be modeled.

The middle section of the form contains information about the tutorial times. If our UoD has only one subject (CS114), we could model this information as unary facts (e.g., Time 'Mon 3 p.m.' is available'). If we need to cater for other subjects as well, then we need to treat the "CS114" at the top of the form as data, and hence verbalize the schedule as binary facts (e.g., Subject 'CS114' has a tutorial slot at Time 'Mon 3 p.m.'). A completed tutorial preference form for a different subject is shown in Figure 3.6.

Here the layout of the five weekdays into columns makes it more obvious that the times are to be treated as data. The first fact from this section reads: Subject 'CS183' has a tutorial slot at Time 'Tues 2 p.m.'.

CS183 Tutorial preferences form

Monday	Tuesday	Wednesday	Thursday	Friday
			9 a.m.	
			11. a.m.	
	2 p.m.		2 p.m.	
	3 p.m.	3 p.m.		

Student number: 211780…

Student surname: *Smith*…… *Initials:* JA…..

Tutorial preference 1: Tues 2 p.m……

Tutorial preference 2: Thurs 11 a.m…….

Tutorial preference 3: Tues 3 p.m……

Figure 3.6 A completed tutorial preference form for a different subject.

If you reformat the structure of the earlier CS114 example (Figure 3.5) to agree with this structure and place the two forms side by side, you can see what aspects are to be modeled as data by looking at what data changes between the two forms (subject code, times, student details).

In this larger UoD, assuming that students may enrol in many subjects, the facts about student preferences now need to take the subject into account. Instead of binaries, the preference facts are now verbalized as ternaries. For example: Student 302156 has first tutorial preference for Subject 'CS114' at Time 'Mon 3 p.m.'; Student 211780 has third tutorial preference for Subject 'CS183' at Time 'Tues 3 p.m.'. Tutorial allocations would also need to indicate the relevant subject (e.g., Table 3.5 would need a header showing the subject). Tutorial groups would then need a composite reference scheme including both the subject code and the group code. For example, the first fact from the CS114-headed version of Table 3.5 would now read as: TuteGroup 'A' of Subject 'CS114' meets at Time 'Mon 3 p.m.'. Composite reference schemes are discussed in detail later in the book.

It is seen later that no set of examples can be significant with respect to derivation rules or subtype definitions. In such cases the use of a domain expert is essential. With the current domain, we made no mention of the rules used to arrive at the tutorial allocations. If in addition to storing information about preferences and allocations, the information system has to compute the allocations in a nearly optimal way, respecting preferences and other practical constraints (e.g., size of groups), the design of the derivation rules becomes the challenging aspect of the schema. While this can be automated, an alternative is to divide the task between the human expert and the system. High level languages facilitate such cooperative solutions.

Often, information examples used for verbalization in step 1 apply to the way the business or application domain currently works. From these examples we can build the *as-is model* to reflect the current practice (as it is now). Some changes may also be needed to expand or improve the way the business operates. For example, we might start with separate applications to administer tutorials for just one subject and then realize it would be better to integrate these into a single application capable of handling all the subjects (as discussed earlier). By including examples of the new data requirements, we are then able to build the *to-be model*, which reflects the way we want the business to be in the future. A proper understanding of the as-is model is a great assistance in designing the to-be model. As you gain more experience as a modeler, you will be able to draw upon lessons learned from prior modeling projects to help spot ways to improve things on future projects. Modeling is not just a science. It's an art as well, and that makes it more fun.

A comprehensive set of output reports (covering intermediate stages) may include all the information on input forms. Output reports tend to be easier to interpret, especially if the input forms have been poorly designed. Care is needed in the design of the input forms to make them clear and simple for users.

Information can appear in many ways. Apart from various tables and forms, information may be expressed graphically in various *diagrams*, *charts*, *maps*, and *graphs*. Harris (1996) discusses several hundred different ways of presenting information graphically. Regardless of how it's presented, information can always be verbalized as facts. Because practice helps with verbalization skills, we've included many varied examples in the book to prepare you for performing step 1 in practical situations.

Figure 3.7 A graph showing nonstop flight connections between cities.

As a simple graphical example, Figure 3.7 might be used to display information about nonstop flight connections provided by a particular airline, with the arrowheads indicating the direction of the flights. As an exercise, perform step 1 for this graph before reading on.

Now let's see how well you performed. There is only one entity type: City (.name). There is also only one fact type: City has a flight to City. For instance, the arrow from Chicago to Seattle may be verbalized as: City 'Chicago' has a flight to City 'Seattle'. The "to" in the predicate is important, since it conveys direction and avoids the symmetry problem with the earlier marriage example.

If this was an as-is model, and we wanted also to talk about the flight connections (e.g., state their duration) or to include many airlines, we should add flight numbers to the arrows on the graph. This to-be model leads to a different verbalization, which you might like to try for yourself. A later exercise returns to this example.

In practice, it is usually possible for a city to have more than one airport, so that connections apply directly between airports rather than cities. Airports are identified by airport codes (e.g. 'JFK' and 'LGA' respectively denote John F. Kennedy and LaGuardia airports which service New York).

Airports can service a city without being located in that city. For example, Newark airport (EWR) located in New Jersey also services New York, and Sea-Tac airport (SEA) located between Seattle and Tacoma services Seattle. We need to determine which specific kinds of fact are of interest in the business domain and verbalize them precisely (e.g., Airport services City is a different fact type from Airport is located in City).

By now you may have some sense of the power of verbalizing examples in terms of elementary facts. No matter what kind of example you start with, if you or an assistant understands the example then you should be able to express the information in simple facts. This does require practice at the technique, but this is fun anyway—isn't it?

If you can't do step 1, there is little point in proceeding with the design—either you don't understand the UoD or you can't communicate it clearly.

Although it might sound hard to believe, if you have performed step 1 properly, you have completed the hardest part of the conceptual schema design procedure. The remaining steps consist of diagramming and constraining the fact types. Apart from the problem of detecting unusual constraints and derivation rules, once you learn the techniques you can often carry out those steps almost automatically. With step 1, however, you always need to draw upon your interpretation skills.

Exercise 3.3

1. Assuming suitable entity types and reference modes are understood, which of the following sentences fit our standard pattern for expressing exactly one elementary fact?
   (a) Adam likes Eve.
   (b) Bob does not like John.
   (c) Tom visited Los Angeles and New York.
   (d) Tom visited Los Angeles or New York.
   (e) If Tom visited Los Angeles then he visited New York.
   (f) Sue is funny.
   (g) All people are funny.
   (h) Some people in New York have toured Australia.
   (i) Brisbane and Sydney are in Australia.
   (j) Brisbane and Sydney are in the same country.
   (k) Who does Adam like?

2. Indicate at least two different meanings for each of the following sentences, by including names for object types and reference modes.
   (a) Pluto is owned by Mickey.
   (b) Dallas is smaller than Sydney.
   (c) Arnold can lift 300.

   Perform step 1 of the CSDP for the following output reports. In verbalizing the facts, you may restrict yourself to the top row of the table unless another row reveals a different kind of fact.

3.

Athlete	*Height (cm)*
Jones EM	166
Pie QT	166
Smith JA	175

4.

Athlete	*Height (cm)*
Jones EM	400
Pie QT	450
Smith JA	550

5.

Person	*Height (cm)*	*Year*
Jones EM	166	1955
Pie QT	160	1970
Smith JA	175	1955

6.

Person	*Height (cm)*	*Year*
Jones EM	160	1970
	166	1980
	166	1990

7.

Advisory panel	*Internal member*	*External member*
Databases	Codd	Ienshtein
	Kowalski	Spock
Logic programming	Colmerauer	Robinson
	Kowalski	
	Spock	

8.

Student Nr	*Course*	*Rating*
1001	CS112	7
1001	CS100	6
1002	CS112	4
1003	CS100	7
1003	CS112	4
1003	MP104	4

9.

Parents	*Children*
Ann, Bill	Colin, David, Eve
David, Fiona	Gus

10.

Country	*Friends*	*Enemies*
Disland	Oz	
Hades		Oz
		Wundrland
Wundrland	Oz	Hades
Oz	Disland	Hades
	Wundrland	

11.

Apple	*Australia*	June, July, August
	America	Oct, Dec, Jan
	Ireland	Oct, Dec
Mango	*Australia*	Nov, Dec, Jan, Feb
Pineapple	*America*	June, July
	Australia	Oct, Nov, Dec. Jan

3.4 CSDP Step 2: Draw Fact Types and Populate

Once we have translated the information examples into elementary facts, and performed quality checks, we are ready for the next step in the conceptual schema design procedure. Here we *draw* a conceptual schema diagram that shows all the *fact types*. This illustrates the relevant object types, predicates and reference schemes. Once the diagram is drawn, we check it with a *sample population*.

Table 3.6 A relational table listing who drives what cars.

Drives:	*person*	*car*
	Adams B	235PZN
	Jones E	235PZN
	Jones E	108AAQ

CSDP step 2: Draw the fact types and apply a population check.

Consider the sample output report of Table 3.6. Let us agree that the information in this report can be expressed by the following three elementary facts, using "regNr" to abbreviate "registration number":

The Person named 'Adams B' *drives* the Car with regNr '235PZN'.
The Person named 'Jones E' *drives* the Car with regNr '235PZN'.
The Person named 'Jones E' *drives* the Car with regNr '108AAQ'.

Before looking at the relevant conceptual schema diagram, it may help to explain things if we first view an *instance diagram* for this example (see Figure 3.8).

Instance diagrams illustrate particular instances of objects and relationships. Taking advantage of the concrete nature of the entities in this example, cartoon drawings denote the actual people and cars. The values are shown as character strings. A particular fact or relationship between a person and a car is shown as a solid line. A particular reference between a value and an entity is shown as a broken line.

Figure 3.9(a) shows a *conceptual schema diagram* for the same example, adding inverse readings for the two fact types. Instance diagrams and conceptual schema diagrams depict an *entity type* as a *named, solid, soft rectangle* (rectangle with rounded corners). A *value type* is shown as a *named, broken, soft rectangle*. The object type's name is written inside. Some versions of ORM allow an ellipse (Figure 3.9(b)) or a hard rectangle (Figure 3.9(c)) instead of a soft rectangle.

Figure 3.8 An instance diagram.

Figure 3.9 (a) A conceptual schema diagram (minus constraints), and (b and c) alternate shapes.

On an instance diagram, individual objects are portrayed explicitly (by icons or text). However on a conceptual schema diagram, individual objects are omitted (unless we reference them in associated tables). Recalling that a type is the set of permitted instances, we may imagine that objects are represented as points inside the type shape.

On a conceptual schema diagram, the *roles* (relationship parts) played by objects are shown explicitly as *boxes*. Optionally, roles may be given a name, in which case the *role name* is displayed in square brackets next to the role box, for example, "[driver]" in Figure 3.9(a). Each *predicate* is depicted as a *named, contiguous sequence of one or more role boxes* ("contiguous" means the boxes are adjacent, with no gaps in between).

Predicates are typically ordered from one end to the other, with their reading displayed next to the predicate shape. For binary relationships (2 roles), both forward and inverse predicate readings may be shown, separated by a slash "/", as in Figure 3.9(a).

Each predicate must have at least one reading displayed. If just one predicate reading is displayed, by default this is read left to right on a horizontal predicate and downward on a vertical predicate. *Arrow tips* may be added to a predicate reading to indicate the reading direction. For example, "◄" indicates the predicate reading is right to left, and "▲" indicates the reading direction is upward.

Each role is connected to exactly one object type by a *line*, indicating that the role is played only by objects of that type. A complete conceptual schema diagram includes the relevant constraints. We'll see how to add these later.

A relationship used within the preferred identification scheme for an entity is called a *reference* (e.g., The car registration number '235PZN' refers to some car) or *existential fact* (e.g., There exists a car that has the car registration number '235PZN'). All other conceptual schema relationships are called *elementary facts* (e.g., The person named 'Adams B' drives the car with car registration number '235PZN').

Typically, elementary facts are relationships between entities, and references are relationships between entities and values. References provide the bridge between the world of entities and the world of values.

This is clearly seen in Figure 3.10(a), where an instance has been added to populate each relationship. The relationship between the person and the car (depicted with icons) is an elementary fact. The relationship between the name 'Adams B' and the person is a reference or existential fact, as is the relationship between the registration number '235PZN' and the car.

Figure 3.10 Using reference modes for 1:1 reference: (a) model and (b) abbreviation.

Although both reference predicates have the reading "has", they are different predicates. Internally a CASE tool may identify the predicates by surrogates (e.g., "P2", "P3") or expanded names (e.g., "PersonHasPersonName", "CarHasRegNr"). Although the predicate reading "has" may be used with fact types, it is best avoided if there is a more descriptive, natural alternative. For example, "Person drives Car", if accurate, is better than "Person has Car", which could mean many things (e.g., Person owns Car).

For this example, each person has exactly one name, and each person name refers to at most one person. Each car has exactly one car registration number, and each car registration number refers to at most one car. This situation is seen clearly in the earlier instance diagram, Figure 3.8. Each of the two reference types is said to provide a *simple 1:1 reference scheme*. We read "1:1" as "one to one". Later we'll see how to specify this on a conceptual schema diagram using uniqueness and mandatory role constraints.

When a simple 1:1 naming convention exists, we may indicate the *reference mode* simply by placing its name in *parentheses* next to the name of the entity type, and use the values themselves to depict object instances in associated *fact tables*. Assuming appropriate constraints are added, the populated schema of Figure 3.10(a) may be displayed more concisely by Figure 3.10(b). Unless we want to illustrate the reference schemes explicitly, this concise form is preferred, because it's closer to the way we verbalize facts and it simplifies the diagram. The fact table is omitted if we wish to display just the schema.

Reference modes indicate the mode or manner in which objects (typically values) refer to entities. Using reference modes reduces the need to display value types explicitly. However to understand the abbreviation scheme, we need to know how to translate between reference modes and value types. Different versions of ORM have different approaches to this. The method used in the NORMA tool for ORM 2 is now outlined.

Let the notation "$E\ (r) \rightarrow V$" mean "Entity type E with reference mode r generates the Value type V". Reference modes may be popular, unit-based (measurement), or general. An initial list of *popular reference modes* is predefined, including *name*, *code*, *title*, *nr*, *#*, and *id*. When parenthesized, popular reference modes are preceded by a *dot* ".". To obtain the value type name, a popular reference mode typically has its first letter shifted to upper case, and is then appended to the name of its entity type.

Table 3.7 Driver details.

Person	*LicenseNr*	*Cars driven*
Adams B	A3050	235PZN
Jones E	A2245	235PZN, 108AAQ

For example, Product(.name) → ProductName; Country(.code) → CountryCode; Item(.code) → ItemCode; Song(.title) → SongTitle; Rating(.nr) → RatingNr; Room(.#) → Room#; Member(.id) → MemberId. Finer control is possible using format strings. A value type corresponding to a popular reference mode identifies only one entity type. For example, CountryCode identifies countries only, and GenderCode identifies genders only.

Unit-based (or *measurement*) *reference modes* include a built-in list of physical units (e.g., cm, kg, mile) and monetary units (e.g., USD, XEU), which may be extended by the user. When parenthesized, unit-based reference modes are appended by a *colon* ":". Value type names are generated from unit-based reference modes by replacing the colon by "Value" (in English). For example: (kg:) → kgValue; (USD:) → USDValue.

Each unit is based on a *unit type* (or *dimension*), which may optionally be displayed after the colon. Sometimes, different units are based on the same dimension. For example: (kg: Mass), (lb: Mass), (USD: Money), (XEU: Money).

General reference modes have no added punctuation, and their names are the same as their value type names. For example: (ISBN) → ISBN; (URL) → URL. These value types may identify many entity types (e.g., URL could be used to identify Link and Website).

As a check that we have drawn the diagram correctly, we should populate each fact type on the diagram with some of the original fact instances. We do this by adding a **fact table** for each fact type and entering the values in the relevant columns of this table.

In ORM, a fact table is simply a table for displaying instances of a fact type. The term "fact table" is used in a different sense in data warehousing (see Section 16.2). A diagram that includes both a schema and a sample database is called a *knowledge base diagram*.

Consider the output report of Table 3.7. LicenseNr entries identify the person's driver's license. Figure 3.11 shows the conceptual schema populated with the original data, in two fact tables. When the object types are simply identified (as here), the population of a fact role appears in a column of the fact table, with the column positioned next to the role.

At this stage, the diagram is incomplete because constraints are not shown. At least one fact from each fact table should be verbalized to ensure that the diagram makes sense. Populating the schema diagram is useful not only for detecting schema diagrams that are nonsensical, but also for clarifying constraints (as shown later).

Figure 3.11 A knowledge base diagram for Table 3.7 (constraints omitted).

Table 3.8 Smokers and nonsmokers.

Smokers	*Nonsmokers*
Arnie	Lee
Sherlock	Norma
	Terry

Nowadays most nonsmokers prefer a smoke-free environment in which to work, travel, eat, and so on. So for some applications, a report such as Table 3.8 is relevant. Please perform step 1 on this table before reading on.

One way to express the facts on row 1 is: Person (firstname) 'Arnie' smokes; Person (firstname) 'Lee' is a nonsmoker. Each of these facts is an instance of a different *unary fact type*. With a unary fact type, there is only one role. The knowledge base diagram is shown in Figure 3.12. Here the two roles belong to different fact types. This is shown by separating the role boxes.

If desired, the two unaries may be transformed into a single binary by introducing SmokingStatus as another entity type, with codes "S" for smoker and "N" for non-smoker. As a result, the first row of Table 3.8 could be rephrased as Person(.firstname) 'Pat' has SmokingStatus(.code) 'S'; Person 'Norma' has SmokingStatus 'N'. This approach is shown in Figure 3.13.

Each of the binary examples discussed had two different entity types. Fact types involving different entity types are said to be *heterogeneous fact types*. Most fact types are of this kind.

However, the fact type in Figure 3.14 has one entity type—Person. If each role in a fact type is played by the same object type, we have a *homogeneous fact type*. The binary case of this is called a *ring fact type* since the path from the object type through the predicate loops back to the same object type, forming a ring.

Figure 3.12 A knowledge base diagram for Table 3.8 (unary version).

Figure 3.13 A knowledge base diagram for Table 3.8 (binary version).

Figure 3.14 A ring fact type with sample population.

Here the forward predicate (is husband of) and inverse predicate (is wife of) are shown. Some versions of ORM set predicates out in gerundive form (e.g., husband of, wife of). Although useful for some styles of queries, it is generally better to use full predicate readings, since this leads to better verbalization of both facts and constraints.

To make diagrams more compact, you may abbreviate names for predicates, object types, and reference modes if their expanded versions are obvious to users. But this relaxed policy should be used with care. ORM tools often allow you to attach descriptions of model elements, which can compensate for such name shortening.

Apart from communication with humans, conceptual schemas provide a formal specification of the structure of the UoD, so that the model may be processed by a computer. Hence the schema diagrams we draw must conform to the formation rules for legal schemas. They are not informal cartoons.

Now consider the output report of Table 3.9. Here we have a *ternary fact type*. The reference schemes are Student(.nr), Course(.code), and Rating(.nr). Given this, we may express the fact on the first row as Student '1001' for Course 'CS100' scored Rating 4.

On a conceptual schema diagram, a ternary fact type appears as a sequence of three role boxes, each attached to an object type, as shown in Figure 3.15. When names for ternary and longer predicates are written on a diagram, the place-holders are included, each depicted by an *ellipsis* "…".

Table 3.9 A relational table storing student results.

Result:

studentNr	*course*	*rating*
1001	CS100	4
1002	CS100	4
1002	CS114	5

Figure 3.15 A populated ternary fact type for Table 3.9.

Figure 3.15 includes a sample population. No matter how high the *arity* (number of roles) of the fact type, we can easily populate it for checking purposes. Each column in the fact table is associated with one role in the predicate.

An earlier example transformed unaries into a binary. Another kind of schema transformation is **objectification** or **nesting**. This effectively *treats a relationship between objects as an object itself*. Consider once more the top row of Table 3.9. Instead of expressing this as a single sentence, we might convey the information in two sentences:

Student '1001' enrolled in Course 'CS100'.
This Enrollment resulted in Rating 4.

Here "This Enrollment" refers back to the enrollment relationship between the specific person and the specific subject mentioned in the first sentence. Any such enrollment may be treated as an object in its own right.

The act of making an object out of a relationship is called *objectification* or *reification* and corresponds to the linguistic act of nominalization (making a noun out of a verb phrase). An object formed by objectification is called an *objectified relationship*. The type of object so formed is called an *objectified association*, or an *objectification type*. Recall that "association" means "relationship type". Section 10.5 refines the notion of objectification to distinguish two varieties (propositional and situational).

An objectified association is depicted by a named, soft-rectangle around the predicate being objectified (see Figure 3.16). The name of the objectified association is placed in *double quotes*. An objectified association usually has two roles, but may have one or more.

Entries in fact columns for nested objects may be shown as bracketed pairs (triples etc.) of values. For example, the enrollment of student 1001 in CS100 appears as "(1001, CS100)" in the fact table for the resulted in predicate. Note that *nesting is not the same as splitting*. Figure 3.16 does not show two independent binaries. The resulted in predicate cannot be shown without the enrolled in predicate. The ternary in Figure 3.15 is still elementary. Figure 3.15 is said to be the *flattened*, or *unnested*, version.

Chapter 14 deals with schema equivalence in detail. The nested and flattened versions are not equivalent unless the role played by the objectified association is mandatory. With our current example, this means that a rating must be known for each enrollment. In this case the flattened version is preferred, since it is simpler to diagram and populate. As discussed later, nesting is often preferred if the objectified association has an optional role, or more than one role to play.

Figure 3.16 Knowledge base for Table 3.9 (nested version).

Figure 3.17 One way of verbalizing the first row.

For example, suppose we widen the UoD to include enrolment dates. The flat solution needs another ternary: Student enrolled in Course on Date. With the nested solution, we simply add the binary: Enrollment occurred on Date. As shown later, the nested solution also simplifies constraint specification in this case, and hence would now be preferred.

Now consider the travel record example depicted in Figure 3.17. Using the telephone heuristic, the modeler verbalizes the first row of data as an instance of the ternary fact type: Politician visited Country in Year.

The sample data indicates that many politicians may visit many countries in many years. Because of this symmetry, there are many options for nesting. We could objectify Politician visited Country as Visit, and then add Visit occurred in Year. We might instead objectify Politician traveled in Year as Travel and then add Travel visited Country, or objectify Country was visited in Year as Visit, and add Visit was by Politician. With no strong reason for one nesting choice over the other, it is simpler to leave it as a ternary.

Although we usually display only one reading for ternary and longer fact types on the diagram, there are many possible readings depending on the order in which we traverse the roles. An n-ary predicate has $n!$ (factorial n, i.e., $n \times n\text{-}1 \times \ldots \times 1$) reading orders. Hence a ternary has 6 possible reading orders, a quaternary has 24, and so on.

For example, the ternary fact type given earlier could be specified by any of the following six readings: Politician visited Country in Year; Politician in Year visited Country; Country was visited by Politician in Year; Country in Year was visited by Politician; Year included visit by Politician to Country; Year included visit to Country by Politician. Some ORM tools also support alias readings for any given reading order.

In practice, one reading is usually enough. However, if we wish to query the schema directly, it is handy to be able to navigate from any given role. To cater for this we would need to supply for each role a reading that starts at that role (the order of the later roles doesn't matter). So we never need any more than n readings for any n-ary fact type. This is a lot fewer than factorial n. Supplying predicate readings for more than one reading order of a fact type can also help with better verbalization of constraints and derivation rules. In practice, one often provides forward and inverse readings for binary fact types, but provides alternate readings for n-ary fact types only on an as-needed basis.

Now suppose we need to design a database for storing sales data that can be displayed graphically as shown in Figure 3.18. This three-dimensional bar chart shows the sales figures for two computer-aided drafting products codenamed "ACAD" and "BCAD". As an exercise in steps 1 and 2, try to verbalize the sales information and then schematize it (on a conceptual schema diagram) before reading on.

Consider the first bar on the left of the chart. A person familiar with the domain might verbalize the fact as: "BCAD in the first quarter had sales of one million dollars". This completes step 1a. As modelers, we complete step 1b by refining this into one or more elementary facts. In this case, we may verbalize it as a single ternary: "The Product with code 'BCAD' in the Quarter numbered 1 had sales of MoneyAmount 1000000 USD".

We chose to identify quarters using numbers (e.g., 1) but you can use codes (e.g., 'Q1') if you like. We used the object type "MoneyAmount" instead of "Sales" to make the underlying domain explicit. This makes it clear that we can compare sales figures with other monetary figures (e.g., costs and profits).

Assuming the chart applies to the United States, we chose USD (United States Dollar) for the monetary unit. This distinguishes it from other dollars (e.g., AUD for Australian Dollar). This is good practice, but if there is no danger of confusion you could simply show the unit as "$".

As an alternative to directly using MoneyAmount as the object type, we could do this indirectly by using Sales with the expanded reference mode (USD: Money), which may be abbreviated to (USD:). Each other bar may be verbalized similarly. This completes step 1. In preparation for step 2, we could set the fact type out with reference modes in parentheses as follows Product(.code) in Quarter(.nr) had sales of MoneyAmount(USD:).

It is now a simple task to draw the conceptual schema diagram. As a check, we populate it with some sample fact instances. Figure 3.19 shows both direct and indirect ways of typing sales to money amounts. Choose whichever suits you. As a display option, (USD:) may be expanded to (USD: Money).

Figure 3.18 Can you verbalize this bar chart?

Figure 3.19 A conceptual schema for the sales data, with a sample population.

Another common way for presenting numeric data is the pie chart. A legend is often provided next to the chart to indicate the items denoted by each slice. Each slice of the pie indicates the portion of the whole taken up by that particular item. An example is given in Figure 3.20. Try to schematize this yourself before reading on.

Applying step 1a to the defense slice of the first pie, we could verbalize the fact as "In 1965 defense consumed 43% of the budget". To complete step1, we refine this to "In Year 1965 CE the BudgetItem named 'Defense' consumed Portion 43% of the budget". Because the other slices denote the same kind of fact. we may generalize to the ternary fact type: in Year (CE) BudgetItem(.name) consumed Portion (%) of the budget. Here, the predicate "in … … consumed … of the budget" has front text before the first placeholder and two placeholders adjacent to one another. Although fairly rare, this is a legal mixfix predicate.

This kind of flexibility makes verbalization easier than it would have been otherwise. To complete step 2, the resulting schema and sample data are shown in Figure 3.21.

To conclude this section, let's review some terminology. Three terms for objects have now been introduced. Entities are the objects in the UoD that we reference by means of descriptions. Values (e.g., characters strings or numbers) appear as entries in database tables and are often used to refer to entities. Finally, relationships between objects may be treated as objects themselves: these are objectified relationships (or nested objects).

Figure 3.20 Can you schematize this pie chart?

Figure 3.21 A conceptual schema for the budget data, with a sample population.

There are two common notations for the arity or "length" of a predicate. In Table 3.10, the preferred Latin-based notation, shown as the main descriptor, is set out for the first nine cases. The alternate Greek-based descriptor tends to be restricted to the first five cases, as shown. In practice it is rare for any elementary predicate to exceed five roles.

Although we should populate conceptual schema diagrams for checking purposes, fact populations do not form part of the conceptual schema diagram itself. In the following exercise, population checks are not requested. However, we strongly suggest that you populate each fact type with at least one row as a check on your work.

Table 3.10 Classification of predicates according to number of roles.

Nr roles	*Main descriptor*	*Alternate descriptor*
1	unary	monadic
2	binary	dyadic
3	ternary	triadic
4	quaternary	tetradic
5	quinary	pentadic
6	senary	…
7	septenary	
8	octanary	
9	nonary	

Exercise 3.4

1. The names and gender of various people are indicated below:
   Male: Fred, Tom
   Female: Ann, Mary, Sue
   (a) Express the information about Fred and Ann in unary facts.
   (b) Draw a conceptual schema diagram based on this choice
   (c) Express the same information in terms of binary elementary facts.
   (d) Draw a conceptual schema diagram based on this choice.

   Note: For the rest of this exercise, avoid using unary facts.

2. Draw a conceptual schema for the fact types in the following questions of Exercise 3.3: (a) Q. 3; (b) Q. 4; (c) Q. 5; (d) Q. 6; (e) Q. 7; (f) Q. 8; (g) Q. 9; (h) Q. 10.

Perform steps 1 and 2 of the CSDP for the following output reports.

3.

Retailer	*Item*	*Quantity sold*
InfoWare	SQL+	330
	Zappo Pascal	330
	WordLight	200
SoftwareLand	SQL+	330
	Zappo Pascal	251

4.

Item	*Retailer*	*Quantity sold*
SQL+	InfoWare	330
	SoftwareLand	330
Zappo Pascal	InfoWare	330
	SoftwareLand	251
WordLight	InfoWare	200

5.

Tute group	*Day*	*Hour*	*Room*
A	Mon	3 p.m.	69-718
B	Tue	2 p.m.	42-B18
C1	Thu	10 a.m.	69-718
C2	Thu	10 a.m.	67-103

6. (*Hint:* Make use of nesting)

Subject	*CreditPts*	*Semester*	*Enrollment*	*Lecturer*
CS100	8	1	500	DBJ
CS102	8	2	500	EJS
CS114	8	1	300	TAH
CS115	8	2	270	TAH
CS383	16	1	50	RMC
CS383	16	2	45	PNC

7. The following interactive voting form is used to input votes by cruise club members on various motions (proposals moved by a club member). This example shows one completed form after a member has selected his/her voting choices. Although passwords are not displayed, they are captured by the information system. Perform CSDP steps 1 and 2 to schematize this UoD.

8. Assuming that appropriate names are supplied for entity types, reference modes, and predicates and that appropriate constraints are added, which of the following conceptual schema diagrams are legal? Where illegal, briefly explain the error.

9. The following chart indicates changes in the value of the Euro and the Australian dollar relative to the U.S. dollar over a given period. Figures may be recorded at the start of each month (only some figures are shown here). Schematize this domain.

3.5 CSDP Step 3: Trim Schema; Note Basic Derivations

Having drawn the fact types and performed a population check, we now move on to step 3 of our design procedure. Here we check to see if there are some entity types that should be combined. We also check to see if some fact types can be derived from the others by arithmetic computation.

CSDP step 3: Check for entity types that should be combined; note any arithmetic derivations

To understand the first part of this step, we need to know how the objects in the UoD are classified into types. Figure 3.22 shows the basic division of objects into entities (non-lexical objects) and values (lexical objects). *Entities* are identified by definite descriptions and may typically change their state, whereas *values* are constants (typically character strings or numbers). Values might include other objects directly representable on a medium (e.g., sounds), but such possibilities are ignored in this book.

These subdivisions are **mutually exclusive** (i.e., they have no instance in common). For example, no character string can be an entity. The division of a whole into exclusive parts is called a **partition**. You may think of it as cutting a pie up into slices. The slices are *exclusive* (they don't overlap) and *exhaustive* (together they make up the whole pie).

Figure 3.22 gives an example of how the object types might be divided into entity types and value types for a particular business domain. Which kinds of objects exist depends on the UoD. Basically, objects are grouped into the same type if we want to record similar information about them.

For any UoD, there is always a top-level partitioning of its entities into exclusive types: these are called **primitive entity types**. We may introduce *subtypes* of these primitive types, especially if they have some specific roles to play. In ORM, subtypes are shown connected by an arrow to their supertype. Although shown separately, it is possible that subtypes of a given entity type may overlap.

However, *primitive entity types never overlap*. For example, no person can be a city. On a conceptual schema diagram, the visual separation of primitive entity types indicates that these types are mutually exclusive. The same is not true of subtypes. In Figure 3.23, for example, Person and City are mutually excusive but Manager and Woman need not be. Subtypes are discussed in detail in Chapter 6.

Figure 3.22 Partitioning objects into types of entities and values.

Figure 3.23 Person and City are primitive entity types and hence mutually exclusive.

Figure 3.24(a) shows two value types: Surname and Cityname. Even if these appear as top level types in an ORM schema diagram, it is possible that they overlap, as shown in the Euler diagram of Figure 3.24(b). Here the character strings 'Denver' and 'Columbus' are both surnames and city names. Although subtype relationships between value types are rarely displayed, they may exist implicitly. In rare cases where the supertype plays a role of interest, we might display the connections explicitly, as in Figure 3.24(c).

Step 3 of the design procedure begins with a check to see if some entity types should be combined. At this stage we are concerned only with primitive entity types, not entity subtypes. So if you spot some entity types that do overlap, you should combine them into a single entity type, or at least introduce a common supertype (see Chapter 6).

For example, consider Table 3.11 which is based on the movie application discussed in Chapter 1. Suppose that as a result of applying steps 1 and 2 we arrived at the diagram shown in Figure 3.25. Do you see what's wrong with this diagram?

Figure 3.25 displays MovieStar and Director as separate, primitive entity types. This implies that these types are mutually exclusive (i.e., no movie star can be a director). But is this the case? Our sample population lists the value "Stephen Chow" in both the Director column and the Stars column. Does this refer to the same person?

If in doubt you can ask a domain expert. In actual fact, it is the same person. As a result, we must combine the Movie Star and Director entity types into a single entity type as shown in Figure 3.26.

Figure 3.24 Overlapping value types have an implicit or explicit supertype.

Table 3.11 Some motion pictures.

Movie#	*Movie Title*	*Director*	*Stars*
1	Cosmology	Lee Lafferty	
2	Kung Fu Hustle	Stephen Chow	Stephen Chow
3	The Secret Garden	Alan Grint	Gennie James, Barret Oliver
…	…	…	…

Figure 3.25 A faulty conceptual schema.

Figure 3.26 The result of applying step 3 to Figure 3.25.

This shows that it is possible to be both a star and a director. Of course, it does not imply that every movie star is a director. A later chapter discusses how to add subtypes later if necessary. For example, if some other facts are to be recorded only for directors, we form a Director subtype of Person for those additional facts.

One case where entity types may always be combined, if desired, is when they have the *same unit-based reference mode*. Here the entity is envisaged as a quantity of so many units (e.g., kilograms or U.S. dollars). Let's look at a few examples.

Consider the output report of Table 3.12, and assume that "$" means U.S. dollar. Here $50 is a wholesale price and a markup. In both cases the $50 denotes the same amount of money, so wholesale prices and markups overlap. If the table population is significant, the set of retail prices does not overlap the set of markups. Nevertheless, it is *meaningful to compare* retail prices and markups since they have the same unit (U.S. dollars). For instance, article A1 has a retail price that is three times its markup.

In modeling this report, we must reveal that the entries in the final three columns are compatible because they all represent amounts of money. One way to schematize this UoD is shown in Figure 3.27(a). By assigning WholesalePrice, RetailPrice, and Markup the same unit-based reference mode (USD), we implicitly declare that they are all based on the same unit dimension.

Table 3.12 Monetary details about articles on sale.

Article	*Wholesale Price ($)*	*Retail price ($)*	*Markup*
A1	50	75	25
A2	80	130	50
A3	50	70	20
A4	100	130	30

Figure 3.27 Unit-based types are implicitly subtypes of their unit dimension.

The colon in "(USD:)" signifies that the reference mode is unit-based. When entering this in an ORM 2 tool, we declare the unit dimension (by selecting from a predefined list or adding our own). In this case, the unit dimension is MoneyAmount (or simply Money). As a display option, the unit dimension may be shown explicitly, as in Figure 3.27(b).

WholesalePrice, RetailPrice, and Markup are not primitive types. They are implicitly subtypes of MoneyAmount. Instead, we may collapse these entity types into MoneyAmount and move their distinguishing semantics into the predicate readings, as in Figure 3.28. We may retain the names of the former entity types as role names, as shown.

The output report satisfies the following mathematical relationship between the values in the last three columns: markup = retail price – wholesale price. Assuming this is significant, the markup value may be **derived** from the wholesale and retail values by means of this rule. To minimize the chance of human error, we have the system derive this value rather than have humans compute it.

*To indicate that a fact type is derived, an **asterisk** is appended to its predicate reading*. Whether or not the derived fact type is displayed on the diagram, a *derivation rule* for it should be supplied. In this book, derivation rules are displayed as textual annotations.

A derivation rule may be specified in either attribute style or relational style. *Attribute style* uses role names to refer to attributes of an object type, typically cited in a for-each clause, as in Figure 3.28. For binary associations, role names may be treated as naming attributes of the object type at the opposite end of the association.

If the *context* of a rule is declared in a for-each clause, it may be assumed thereafter. If the context is not predeclared, it may be provided *in situ* by qualified names using either the *dot notation* familiar from programming and UML (e.g., Article.markup) or the *of-notation* (e.g., markup **of** Article). When validating rules with nontechnical domain experts, the of-notation is usually preferable to the dot notation.

Figure 3.28 An alternative to Figure 3.27, with derivation rule added.

Relational style instead uses predicate readings. For example, using "**iff**" for "if and only if", the markup rule may be declared in relational style as shown below. Subscripts distinguish variables introduced in the rule body (after "iff") from variables of the same type (here MoneyAmount) that occur in the rule head (before "iff").

Article has markup of MoneyAmount **iff**
Article retails for MoneyAmount_1 **and** Article wholesales for MoneyAmount_2 **and**
MoneyAmount = $\text{MoneyAmount}_1 - \text{MoneyAmount}_2$.

For brevity, relational style assumes that variables in the rule head are universally quantified and that variables introduced in the body are existentially quantified. So the rule may be stated explicitly thus: **For each** Article **and** MoneyAmount, Article has markup of MoneyAmount **iff** Article retails for **some** MoneyAmount_1 **and** Article wholesales for **some** MoneyAmount_2 **and** MoneyAmount = $\text{MoneyAmount}_1 - \text{MoneyAmount}_2$.

Strictly speaking, the equation markup = retailPrice – wholesalePrice is a *constraint* between the markup, retail price, and wholesale price fact types, with one degree of freedom. Any one of the three could be derived from the other two. Sometimes we might wish to enter retail and wholesale prices and have the system derive the markup. At other times, we might wish to enter the wholesale price and markup to derive the retail price. We might also want to enter the retail price and markup to derive the wholesale price.

If we wish the system to support all these choices, then each of the three fact types is *semiderived* (instances may be asserted or derived) and its predicate is marked with a "$^+$" (intuitively, half an asterisk) instead of "*". This is sometimes useful in practice (e.g., exploring loan options based on interest rates and repayment periods).

In most derivation cases however, we decide beforehand that one specific fact type will always be the derived one. In this case, the derivation rule may be clearly specified as a *definition* (rather than merely an equation or biconditional) where the derived fact type is defined in terms of the others using "define … as …". For example:

For each Article, **define** markup **as** retailPrice – wholesalePrice.

In such a definition, the derived fact type is said to be the *definiendum* (what is required to be defined). A fact type that is *primitive* (i.e., not defined in terms of others) is said to be an *asserted* fact type or *base* fact type. *Derived* fact types are defined in terms of other fact types (asserted or derived). For arithmetic derivation rules, such as our current example, attribute style is typically more compact and readable and hence is preferred. For logical derivation rules (see Section 5.5), relational style is often preferable.

Now consider the output report of Table 3.13, which shows details about windows on a computer screen. Try to schematize this before reading on.

Table 3.13 Window sizes.

Window	*Height (cm)*	*Width (cm)*	*Area (cm^2)*
1	4	5	20
2	6	20	120
3	10	15	150
4	5	5	25

The values in the last three columns are all numbers. Can we collapse Height, Width, and Area into one entity type? You might argue that Height and Width overlap since the value 5 is common, and that Width and Area overlap since each includes the value 20. However, Area is measured in square centimeters, quite a different unit from centimeters.

Heights and widths may be meaningfully compared since both are lengths: a length of 5 cm may be an instance of a height or a width. However, a length of 20 cm is not the same thing as an area of 20 cm^2. If our final column was headed "Perimeter(cm)", we could collapse three headings into one entity type as previously. But since Area is fundamentally a different type of quantity, we must keep it separate, as in Figure 3.29(a).

As an alternative, the height and width fact types could be modeled using the fact types Window(.nr) has Height(cm:) and Window(.nr) has Width(cm:), where the centimeter unit is based on the unit dimension Length.

The derivation rule for area in Figure 3.29(a) is prepended by a single asterisk to indicate it is derived and not stored. The rule body includes another asterisk, which in this context means multiplication. In principle, any one of area, height, and width could be derived from the other two. In many cases, however, there simply is no choice as to which fact type is derived. For example, facts about sums and averages are derivable from facts about individual cases, but except for trivial cases we cannot derive the individual facts from such summaries.

A derived fact type that is not stored is *derived-on-query* (*lazy evaluation*), so the derived information is computed only when the information is requested. For example, if our Window schema is mapped to a relational database, no column for area is included in the base table for Window. The rule for computing area may be included in a view definition or stored query and is invoked only when the view is queried or the stored query is executed. In most cases, lazy evaluation is preferred (e.g., computing a person's age from their birth date and current date).

A fully derived fact type that is also stored is *derived-on-update* (*eager evaluation*). Sometimes eager evaluation is chosen because it offers better performance (e.g., computing account balances). In this case, the information is stored as soon as the defining facts are entered and is updated whenever they are updated. Fact types that are fully derived and stored are marked with a *double-asterisk* "**", as in Figure 3.29(b). When the schema is mapped to a relational database, a column is created for the derived fact type, and the computation rule is specified either as a trigger or as a computed column rule that is fired whenever the defining columns are updated (including inserts or deletes).

Figure 3.29 Schema for Table 3.13, where area is simply derived (a) and derived and stored (b).

Figure 3.30 Should Doctor, Dentist, and Pharmacist be combined?

Now consider Figure 3.30. This might describe part of a UoD concerning practitioners in a medical clinic. Here the entity types Doctor, Dentist, and Pharmacist have a similar reference mode (name), but because this is not unit based, this is no reason to combine the types. If somebody could hold more than one of these three jobs then the overlap of the entity types would normally force a combination.

However, suppose that the entity types are mutually exclusive (i.e., nobody can hold more than one of these jobs). In this case we still need to consider whether to combine the entity types, since the *same kind of information* (their gender) is recorded for each.

In such cases, ask this question. *Do we ever want to list the same kind of information for the different entity types in the same query?* For example, do we want to make the request "List all the practitioners and their gender"? If we do, then we should normally combine the entity types as shown in Figure 3.31. If we don't, then there may be grounds for leaving the schema unchanged. Section 6.6 examines this issue in more detail.

Even if no doctor can be a dentist, the schema of Figure 3.30 permits a doctor and a dentist to have the same name (e.g., "Jones E"). In Figure 3.31, the use of "(.name)" with Practitioner implies that each instance of PractitionerName refers to only one Practitioner.

Suppose we add the constraint: **Each** Practitioner holds **at most one** Job. A graphical notation for this kind of constraint is discussed in the next chapter. With this constraint added, Figure 3.31 would forbid any doctor from having the same name as a dentist. If the original names did overlap, we would now need to rename some practitioners to ensure that their new names are distinct (e.g., "Jones, E1" and "Jones, E2").

Alternatively, we might choose a new simple identification scheme (e.g., PractitionerNr), or identify practitioners by the combination of their original name and job. Reference schemes are discussed in detail in Section 5.3.

To preserve the distinction between the different kinds of practitioner, we introduced the entity type Job and constrained job names to the set { 'doctor', 'dentist', 'pharmacist' }. Such "value constraints" are discussed in detail in Section 6.3.

Figure 3.31 An alternative schema for the UoD of Figure 3.30.

The new schema is simpler since it replaced three binary fact types with two binaries. If we had even more kinds of practitioner (e.g., acupuncturist, herbalist) the saving would be even more worthwhile. If we have only two kinds of practitioner (e.g., doctor and pharmacist) both schemas would have the same number of fact types. But even in this case, the new version is generally favored. If additional information is required for specific kinds of practitioners, subtyping should be added, as discussed in Chapter 6.

In rare cases, entity types might overlap but we are not interested in knowing this, and collapsing the types is awkward. We may then leave the types separate so long as we declare that our model differs from the real world in this respect.

In performing CSDP step 3, the relevant questions to ask are as follows. Here, the derivation rules that concern us are of an arithmetic nature. These are usually fairly obvious. Logical derivations can be harder to spot, and are considered in a later step.

1. *Can the same entity belong to two entity types?* If so, combine the entity types into one (unless such identities are not of interest).
2. *Can entities of two different types be meaningfully compared (e.g., to compute ratios)? Do they have the same unit or dimension?* If so, either combine the entity types into one, or assign the units the same unit dimension.
3. *Is the same kind of information recorded for different entity types, and will you ever need to list the entities together for this information?* If so, combine the entity types, and if needed add a fact type to preserve the original distinction.
4. *Is a fact type arithmetically derivable from others?* If so, add a derivation rule. If you include the fact type on the diagram, mark it with "*" if fully derived, or "+" if semiderived. If the derived facts are also stored, append a final asterisk.

In addition to verbalization and population, a third way to validate a model is to see whether it enables *sample queries* to be answered, either directly from, or by derivation on, the fact populations. If you know what kinds of questions the system must be able to answer, navigate around the ORM model to see if you can answer them. If you can't, your schema is incomplete and you should add the fact types and/or derivation rules needed to answer the queries.

Exercise 3.5

Perform steps 1–3 of the CSDP for the following output reports. In setting out derivation rules, you may use any convenient notation.

1.

Software	*Distributor*	*Retailer*
Blossom 1234	InfoWare	PCland
		SoftKing
SQL++	TechSource	PCland
WordLight	TechSource	InfoWare
		SoftKing

2.

Project	*Manager*	*Budget*	*Salary*	*Birth year*
P1	Smith J	38000	50000	1946
P2	Jones	42000	55000	1935
P3	Brown	20000	38000	1946
P4	Smith T	36000	42000	1950
P5	Collins	36000	38000	1956

3.

Dept	*Budget*	*NrStaff*	*EmpNr*	*Salary*	*Salary total*
Admin	800000	2	E01	50000	
			E02	35000	85000
Sales	900000	3	E03	40000	
			E04	35000	
			E05	40000	115000
Service	900000	2	E06	55000	
			E07	35000	90000

4.

Employee	*Project*	*Hours*	*Expenses*
E4	P8	24	400
E4	P9	26	300
E5	P8	14	200
E5	P9	16	220
E6	P8	16	240
E6	P9	14	220

5.

Female staff		*Male staff*	
Name	*Dept*	*Name*	*Dept*
Sue Bright	Admin	Sue Bright	Admin
Eve Jones	Admin	Eve Jones	Admin
Ann Smith	Sales	Ann Smith	Sales

6. The following excerpt is from the final medal tally for the 1996 Olympics. Only the top five countries are listed here.

	G	*S*	*B*	*Total*
United States	44	32	25	101
Germany	20	18	27	65
Russia	26	21	16	63
China	16	22	12	50
Australia	9	9	23	41
…	…	…	…	…

(a) Schematize this using binary fact types only.
(b) Schematize this using a ternary fact type.

7. Perform CSDP steps 1–3 for this report (13 cm = 5.25 inch; 9 cm = 3.5 inch).

Medium	*Capacity*	*Year introduced*	*Disk price (USD)*	*Cost per MB (USD)*
13 cm floppy	160 KB	1981	2.60	16.25
9 cm floppy	720 KB	1985	3.50	4.86
Zip	100 MB	1995	16.65	0.17
CD-R	650 MB	1996	1.79	0.003
DVD-R	9.4 GB	2002	7.89	0.0008

3.6 Summary

The following criteria are desirable characteristics for any language to be used for conceptual modeling: expressibility, clarity, simplicity, orthogonality, semantic stability, semantic relevance, validation mechanisms, abstraction mechanisms, and formal foundation.

Object-Role Modeling was designed with these criteria in mind. In particular, its omission of the attribute concept from base models leads to greater stability and simplicity, while facilitating validation through verbalization and population.

With large-scale business domains, the UoD is divided into convenient modules, the *conceptual schema design procedure* is applied to each, and the resulting subschemas are integrated into the global conceptual schema. The CSDP itself has seven steps, of which the first three were discussed in this chapter.

Step

1a. Verbalize familiar information examples as facts.
1b. Refine these into formal, elementary facts, and apply quality checks.
2. Draw the fact types, and apply a population check.
3. Check for entity types that should be combined, and note any arithmetic derivations.

Step 1 is the most important. This is seeded by *data use cases*, which are relevant information *examples* (e.g., tables, forms, diagrams) that are familiar to the domain expert. These are *verbalized in terms of elementary facts*. The domain expert verbalizes the examples informally as facts in natural language sentences (step 1a).

The modeler completes step 1 by refining these into elementary facts (step 1b). An elementary fact is a simple assertion that an object has some property or that one or more objects participate together in some relationship. With respect to the UoD, an elementary fact cannot be split into smaller facts without information loss.

Elementary facts are expressed as instantiated logical predicates. A logical *predicate* is a declarative sentence with object holes in it. To complete the sentence, the placeholders are filled in by object terms. With simple reference schemes, an entity term is a definite description that includes the name of the *entity type*, *reference mode*, and *value* (e.g., "the Scientist with surname 'Einstein'"). A value term includes the name of the value type and a literal value (e.g., "the Surname 'Einstein'").

Each object hole corresponds to a *role* (part in a relationship). A predicate with one role is *unary*, with two roles is *binary*, with three roles is *ternary*, with four roles is *quaternary*, and with *n* roles is *n*-ary. The value *n* is the *arity* of the predicate.

In CSDP *step 2* we *draw the fact types and apply a population check*. Object types are depicted as named, soft rectangles (solid lines for entity types and dashed lines for value types). An *n*-ary predicate is shown as a named, contiguous sequence of *n* role-boxes. Predicates are ordered. The predicate reading is placed next to the predicate shape. Each role is played by exactly one object type, as shown by a connecting line.

A *simple 1:1 reference scheme* involves a reference predicate between an entity type and a value type, where each entity is associated with exactly one value, and each value is associated with only one entity. This kind of scheme may be abbreviated by enclosing the *reference mode in parentheses* next to the entity type name.

Popular reference modes are preceded by a dot, e.g., Country(.code). *Unit-based reference modes* are appended by a colon, e.g., Height(cm:), possibly followed by the unit's dimension, e.g., (cm: Length). *General reference modes* may share their value types with other entity types, e.g., (URL).

Once a fact type is drawn, it should be checked by populating it with at least one fact and reading it back in natural language. Each column in the associated *fact table* corresponds to a specific role. To talk about an object corresponding to a relationship, an *objectified association* is formed from the fact type. This nominalization process is also known as *nesting*.

In *step 3* of the CSDP, we check for entity types that should be combined and note any arithmetic derivations. For any given UoD, each entity belongs to exactly one of the *primitive entity types* that have been selected (e.g., Person, Car). Hence, if we have drawn two entity types that may have a common instance, we should normally combine them.

Figure 3.32 Some basic symbols used in conceptual schema diagrams.

Even if they don't overlap, entity types that can be meaningfully compared (e.g., to compute ratios) may often be combined. If unit based, either collapse the entity types into one or ensure their units have the same unit dimension.

If the same kind of information is to be recorded for different entity types, these should often be combined, and if needed a fact type should be added to preserve the original distinction.

If a fact type is arithmetically derivable from others, an appropriate *derivation rule* should be declared in relational style (using predicates) or in attribute style (using attributes derived from role names). The derived fact type may be omitted from the diagram, but if included its derivation status should be marked "*" (simply derived) or "+" (semi-derived). If derived facts are also stored, append a final asterisk. Figure 3.32 summarizes the graphic notations met so far.

Chapter Notes

Bentley (1998) suggests the following alternative yardsticks for language design: orthogonality, generality, parsimony, completeness, similarity, extensibility, and openness. Some of these criteria (e.g., completeness, generality, extensibility) may be subsumed under expressibility. Parsimony may be treated as one aspect of simplicity. Another criterion sometimes mentioned is convenience (how convenient, suitable, or appropriate a language feature is to the user). We treat convenience as another aspect of simplicity.

The parsimony–convenience trade-off is nicely illustrated by two-valued propositional calculus, which allows for 4 monadic and 16 dyadic logical operators. All 20 of these operators can be expressed in terms of a single logical operator (e.g., nand, nor), but while this might be useful in building electronic components, it is too inconvenient for direct human communication. For example, "not *p*" is far more convenient than "*p* nand *p*". In practice, we use several operators (e.g., not, and, or, if-then) since their convenience far outweighs the parsimonious benefits of having to learn only one operator such as nand.

When it comes to proving metatheorems about a given logic, it is often convenient to adopt a somewhat parsimonious stance regarding the base constructs (e.g., treat "not" and "or" as the only primitive logical operators), while introducing other constructs as derived (e.g., define "if *p* then *q*" as "not *p* or *q*"). Similar considerations apply to modeling languages.

A classic paper on linguistics that influenced the early development of a number of modeling methods is Fillmore (1968). Use case specification was central to the former Objectory approach of Ivar Jacobson, one of the main contributors to UML. For an overview of logicians' approaches to proper names and descriptions, see Chapter 5 of Haack (1978). Some heuristics to help with step 1 in interpreting common forms used in business are discussed in Choobineh et al. (1992).

Although ordered, mixfix predicates are preferred for naturalness, you could also treat a fact as a named set of (object, role) pairs: $F\{(o_1,r_1), \ldots ,(o_n,r_n)\}$. Here each object o_i is paired with the role r_i that it plays in the fact F. For example, the fact that Wirth designed Pascal might be specified as Design{ (The Person with surname 'Wirth', agentive), (The Language named 'Pascal', objective) }. Instead of the case-adjectives "agentive" and "objective", other role names could be used (e.g., "designer" and "language", or "designing" and "being designed by"). By pairing objects with their roles, the order in which the pairs are listed is irrelevant. This approach is used in RIDL (Reference and Idea Language) (Meersman 1982).

In this book, the term "line" is used informally to mean "line segment" or "edge" (in geometry, lines have no beginning or end). The following notes provide a brief history of ORM.

In the 1970s, especially in Europe, substantial research was carried out to provide high level semantics for modeling information systems. Jean Abrial (1974), Mike Senko (1975) and others modeled binary relationships. In 1973, Eckhard Falkenberg generalized their work to *n*-ary relationships and decided that attributes should not be used at the conceptual level because they involved "fuzzy" distinctions and they also complicated schema evolution. Later, Falkenberg (1976) proposed the fundamental ORM framework, which he called the "object-role model". This framework allowed *n*-ary and nested relationships, but depicted roles with arrowed lines.

Sjir Nijssen (1976) adapted this framework by introducing the circle-box notation for objects and roles and adding a linguistic orientation and design procedure to provide a modeling method called ENALIM (Evolving NAtural Language Information Model) (Nijssen 1977). A major reason for the role-box notation was to facilitate validation using sample populations. Nijssen led a group of researchers at Control Data in Belgium who developed the method further, including Franz van Assche who classified object types into lexical object types (LOTs) and non-lexical object types (NOLOTs). Today, LOTs are often called "value types" and NOLOTs are called "entity types". Bill Kent (1977, 1978) provided several semantic insights and clarified many conceptual issues.

Robert Meersman added subtypes and made major contributions to the RIDL query language with Falkenberg and Nijssen. The method was renamed "aN Information Analysis Method" (NIAM) and was summarized in a paper by Verheijen and van Bekkum (1982). In later years the acronym "NIAM" was given different expansions, and is now known as the "Natural language Information Analysis Method". Two matrix methods for subtypes were developed: one (the role–role matrix) by Dirk Vermeir (1983) and another by Falkenberg and others.

In the 1980s, Falkenberg and Nijssen worked jointly on the design procedure and moved to the University of Queensland, where the method was enhanced further. It was there that Terry Halpin provided the first full formalization of the method (Halpin 1989), including schema equivalence proofs, and made several refinements and extensions. In 1989, Nijssen and Halpin co-authored a book on the method. Another early book on the method was written by Wintraecken (1990).

In the early 1990s, Halpin developed an extended version of NIAM called Formal ORM (FORM), initially supported in the InfoDesigner modeling tool from ServerWare. This product later evolved to InfoModeler (at Asymetrix Corp. and then InfoModelers Inc.) and then VisioModeler (at Visio Corp.). This ORM technology was then recoded to work on the Visio engine, first appearing in Visio Enterprise. Anthony Bloesch and Terry Halpin designed an associated query language called ConQuer (Bloesch and Halpin 1997), supported in the ActiveQuery tool. Microsoft acquired Visio in 2000 and modified its ORM solution for use in Visual Studio.

Many researchers have contributed to the ORM method, and a full history would include many not listed here. Today various versions of the method exist, but all adhere to the fundamental object-role framework. Although most ORM proponents favor *n*-ary relationships, some prefer Binary-Relationship Modeling (BRM), e.g., Peretz Shoval (1993). Henri Habrias (1993) developed an object-oriented version called the Normalized Object-Oriented Method (MOON). The Predicator Set Model (PSM) was developed mainly by Arthur ter Hofstede, Erik Proper, and Theo van der Weide (ter Hofstede et al. 1993), and includes complex object constructors. Olga De Troyer and Robert Meersman (1995) developed another version with constructors called Natural Object-Relationship Model (NORM). Harm van der Lek and others (Bakema et al. 1994) allowed entity types to be treated as objectified predicates, to produce Fully Communication Oriented Information Modeling (FCO-IM).

Independently of the main ORM movement, Dave Embley and others decided that using attributes in conceptual modeling was a bad idea, so they developed Object-oriented Systems Analysis (OSA) which includes an attribute-free "Object-Relationship Model" component that has much in common with ORM (Embley et al. 1992; Embley 1998). The Semantics of Business Vocabulary and Business Rules (SBVR) approach adopted in 2007 by the Object Management Group (OMG

2007) is a recent addition to the family of fact-orientation, and has much in common with ORM 2, with its support of deontic as well as alethic rules.

Various *software tools* have been developed to support different flavors of the fact-oriented approach. The earliest tools developed by Nijssen, Meersman, and others at Control Data (e.g., IAST, RIDL*) are no longer available. Bloesch, Halpin, and others developed VisioModeler (discontinued but freely available), ActiveQuery (currently unavailable), and the ORM modeling solution in Microsoft® Visio for Enterprise Architects (Halpin et al. 2003). The FCO-IM version is supported by the commercial tool CaseTalk (*www.casetalk.com*) and by the freeware tool Infagon (*www.mattic.com*). Nijssen's group at PNA Active Media has developed the Doctool product for NIAM2007, and is currently developing the CogNIAM tool for Cognition enhanced NIAM (*http://cogniam.com/*). Dogma Modeler, Dogma Studio (www.starlab.vub.ac.be), and T-Lex (Trog et. al. 2006) are academic ORM-based tools for specifying ontologies. NORMA (Curland and Halpin 2007), an open-source plug-in to Microsoft® Visual Studio, is under development to provide deep support for ORM 2, the next generation of ORM, and may be downloaded from *www.ormfoundation.org/files/* and from *http://sourceforge.net/projects/orm*. Clifford Heath is currently developing ActiveFacts (*http://dataconstellation.com/ActiveFacts/*), another suite of tools designed to support ORM 2. Victor Morgante, author of the Niam+ tool, is currently working on another ORM tool (*www.viev.com*), but no details are available at the time of writing.

4 Uniqueness Constraints

4.1 Introduction to CSDP Step 4
- Static and dynamic constraints
- Uniqueness constraints and arity checks

4.2 Constraining Unaries and Binaries
- Uniqueness constraints on unary fact types
- Uniqueness constraints on binary fact types

4.3 Constraining Longer Fact Types
- Uniqueness constraints on ternaries
- Uniqueness constraints on quaternaries, etc.

4.4 External Uniqueness Constraints
- Conceptual joins
- External uniqueness constraints

4.5 Arity Checks
- Keys and functional dependencies
- The $n - 1$ rule
- Rephrasing as conjunctions
- Objectified associations

4.6 Projections and Joins
- The projection operation
- The projection–join check

4.7 Summary

4.1 Introduction to CSDP Step 4

So far, the conceptual schema design procedure has focused on specifying the elementary fact types, both asserted and derived. The rest of the CSDP is concerned mostly with specifying **constraints**. Constraints apply to the database and are either static or dynamic. *Static constraints* apply to each individual state of the database and may usually be specified on a schema diagram, as discussed in CSDP steps 4–7. Examples include uniqueness, mandatory role, set comparison, value, subtyping, frequency, and ring constraints. *Dynamic constraints* restrict transitions between states, are often expressed in other ways (e.g., state charts), and are considered in step 7, along with other constraints. In practice, we usually capture all constraints relevant to the fact types being discussed, before moving on to another part of the model.

This chapter discusses **uniqueness constraints** (UCs). Once uniqueness constraints are added to a fact type, checks are made to see whether the fact type is of the right arity or length. In particular, there is a simple uniqueness check that shows that certain kinds of fact types are not elementary and hence should be split.

CSDP step 4: Add uniqueness constraints and check the arity of fact types.

Static constraints are often described as being "constraints on the fact types". More accurately, *static constraints apply to every possible state of the database*. Here "every state" means "*each and every state, taken one at a time*". Hence, static constraints apply to all possible *populations* of the fact types. For example, given the fact type Person(.surname) has Weight(kg:), let's add the constraint **Each** Person has **at most one** Weight. A sample population for two different states is shown in Figure 4.1.

During the lifetime of the information system, the database goes through a sequence of states. In one of these states, Smith is recorded to weigh 90 kg, and Jones is recorded to weigh 100 kg (Figure 4.1(a)). In a later state, Smith is still the same weight but Jones has lost weight by exercising and dieting (Figure 4.1(b)). Over time, Jones has two different weights. However, in each database state, at most one weight is recorded as the current weight for Jones, for Smith, or for any other person.

Figure 4.1 For each state taken individually, each person has at most one weight.

This is how to interpret the static constraint **Each** Person has **at most one** Weight. Fact types such as Person has Weight are "*snapshot fact types*"—each instance applies to a single state, or snapshot, of the database. One way to record history is to use a "*historical fact type*" with explicit reference to time (e.g., Person had Weight on Date).

This chapter discusses how to specify uniqueness constraints on a conceptual schema diagram. Each base fact type must be assigned at least one UC. Given the constraints on the base fact types, constraints on derived fact types are often implied by the derivation rules. For example, if a window has at most one height and at most one width, then the derivation rule "area = height * width" implies that each window has at most one area (which is the uniqueness constraint on the derived fact type Window has Area).

The rest of the chapter discusses CSDP step 4 in the following order. First we discuss how to mark uniqueness constraints on unary and binary predicates. Then we consider uniqueness constraints on longer predicates. After that, we examine external uniqueness constraints (these apply between different predicates). Finally we discuss ways of checking that our fact types have the right arity (number of roles).

4.2 Uniqueness Constraints on Unaries and Binaries

Unary fact types are the easiest, so let's look at them first. Suppose that, as part of a fitness application, we record which people are joggers. This can be handled with a unary fact type, as shown in Figure 4.2.

For simplicity, let's again assume we can identify people just by their surname. A sample population is included. From the conceptual viewpoint, the fact population is the set containing the following facts:

The Person with surname 'Adams' jogs.
The Person with surname 'Brown' jogs.
The Person with surname 'Collins' jogs.

Suppose that after recording these facts, we see Adams run by again. Forgetting that we already recorded his jogging, we now try the following update:

add: The Person with surname 'Adams' jogs.

Figure 4.2 A unary fact type with sample population.

Figure 4.3 Is this update allowed?

If accepted, this update may be pictured as shown in Figure 4.3. The database is a variable whose population at any state is a *set* of facts. Since the fact that Adams jogs was already present, if this fact were added again the population would remain unaltered because sets are insensitive to repetition. For example, the set {Adams jogs, Adams jogs} is identical to the set {Adams jogs}. Sets are defined simply by their membership, and don't change if we display a member more than once (even if this is unusual). So the two fact tables shown in Figure 4.3 are equivalent if we look at each as a set of instances. From this viewpoint, there is no problem with accepting the update.

From the internal viewpoint however, when a fact is added to a database it is typically stored in a previously unallocated space. So accepting this update would cause the fact that Adams jogs to be stored twice, in separate locations. This is an example of **redundancy**. If redundancy occurs, the database is a *bag* of facts rather than a *set* of facts. A bag or *multiset* is just like a set, except repetition of its members is significant. For example, although the sets {1} and {1, 1} are equal, the bags [1] and [1, 1] are different.

Avoiding redundancy in a database helps maintain its integrity by simplifying the correct handling of updates. For instance, suppose the fact that Adams jogs is stored twice in the database. If we later wanted to delete or change this fact, then we would need to take care to do this for both its recorded instances. Controlling redundancy in this way can be a headache, and uncontrolled redundancy can lead to inconsistency. A second reason for avoiding redundancy is to save space (computers have finite storage capabilities).

In practice, to retrieve information more efficiently we sometimes allow controlled redundancy at the *logical* schema level. However, for *conceptual* schema design, *no redundancy may occur in the conceptual database*. Redundancy is simply the repetition of an elementary or existential fact. Since each row of a conceptual fact table corresponds to one fact, *no row of a conceptual fact table may be repeated*. Hence the fact table in Figure 4.3(b) is illegal, since the Adams row is repeated.

Here, for any given state of the database, each person can be recorded as being a jogger at most once. As a result, each entry in the column of the fact table is *unique* (no duplicates may appear in this column). We may represent this *uniqueness constraint* explicitly by placing a *bar* above or below the fact role as shown in Figure 4.4.

Since the fact role corresponds to the fact column, a UC on the role implies that entries in the fact column are unique. We can show the constraint on the conceptual schema diagram alone by omitting the fact table.

Figure 4.4 The uniqueness constraint ensures that entries are unique (no duplicates).

Every ORM asserted fact type has an implied uniqueness constraint spanning it, since for each state of the conceptual database, each asserted fact instance is unique. Because each unary predicate has just one role, only one internal UC can be applied to it. So the uniqueness constraint bar on an asserted unary is implied, and hence may be omitted as a display option, as shown in Figure 4.5(b). However, because a binary or longer predicate has more than one role, there is more than one uniqueness constraint pattern that could apply to the predicate. Hence, for non-unary predicates, the uniqueness constraints need to be stated explicitly.

For *binary* predicates, the most common case has a uniqueness constraint over just one role, as in Figure 4.6. This indicates that *each entry in that role's fact column must be unique*. Duplicate entries are allowed in the other column. Here the politicians of interest are identified by their name, and countries are identified by their two-letter ISO codes (e.g., 'AU' denotes Australia).

In the sample fact table, each value in the politician column occurs only once, but the 'US' value in the country column is repeated. The constraint indicates that for all possible populations, the values in the politician column are unique. Since each politician is referenced at most once in that column, he or she is associated with at most one country in this fact type.

Figure 4.5 Implied uniqueness constraint on a unary base predicate may be omitted.

Figure 4.6 Uniqueness constraint on first role: entries for that column must be unique

Figure 4.7 The forward predicate is n:1; the inverse predicate is 1:n.

In other words: **Each** Politician was born in **at most one** Country. This is how the constraint is formally *verbalized* in *positive form*. The same constraint may be verbalized in *negative form* thus: **It is impossible that the same** Politician was born in **more than one** Country. The absence of a uniqueness constraint on the other role may be verbalized thus: **It is possible that the same** Country is birthplace of **more than one** Politician.

Figure 4.7 shows an instance diagram for the same example. Here entities are depicted as dots inside the entity types, and fact instances are shown as connecting lines. Many politicians may be born in the one country, but many countries may not be the birthplace of the one politician. Hence, the predicate "… was born in …" is said to be *many to one* (*n:1*). The inverse predicate "... is birthplace of ..." is *one to many* (*1:n*).

Instance diagrams such as Figure 4.7 are often used in logic or mathematics to discuss the theory of relations. Although instance diagrams and fact tables are useful for understanding and validating constraints, they are not part of the schema itself, which is concerned only with the structure, not the population. On a conceptual schema diagram, a UC over a single role is shown by placing it next to the role (on either side), as shown in Figure 4.8.

The role-based notation of ORM facilitates validation with sample populations, since each role corresponds to a column in the associated fact table. The uniqueness constraint on the first role means that its entries must be unique. The absence of a UC on the second role indicates that its entries may be duplicated.

Although this notation is easy to understand, and schema diagrams reveal how things are connected, the domain expert assisting you in the modeling task should not be required to master the diagram notation. To validate a constraint, it is more important to use verbalization and sample populations.

For a binary fact type, three rows are enough for a *significant population*, so long as you pick the data carefully. If a column can have duplicates, give it two values that are the same and one different. If a column is unique, make each value different.

Figure 4.8 Each person was born in at most one country.

Looking back at Figure 4.6, note that its population is significant. Using an ORM tool, you can enter sample populations like this and have them printed out along with automatic verbalizations of the constraints for the domain expert to validate.

Now consider the example shown in Figure 4.9, which concerns heads of government (e.g., Presidents or Prime Ministers). At any point in time, a politician can head the government of at most one country, and each country can have at most one head of government. Hence the entries in the politician column must be unique, and the entries in the country column must be unique also. To indicate this, uniqueness constraints are added above each of these columns. These constraints may be verbalized as shown. The conceptual schema for this fact type can be obtained by removing the fact table.

Note the use of the *hyphen* in the inverse predicate. This binds the word with the hyphen to the object type when the constraint is verbalized. So the right-hand uniqueness constraint is verbalized as "**Each** Country has **at most one** head Politician", not "**Each** Country has head **at most one** Politician". Some ORM tools interpret hyphens this way when automatically generating the constraint verbalization.

If you don't include an inverse reading, the right-hand uniqueness constraint verbalizes as "**At most one** Politician governs **each** Country". Usually, a constraint on a role verbalizes better if you include a predicate reading that starts at that role. We generally recommend that you include both predicate readings for binaries, but this is especially advisable if each role has a constraint that applies just to it.

For any state of the database, any one politician can be recorded as government head of only one country, and vice versa. Hence each population of this fact type is said to be a *one-to-one* (or *1:1*) relation. The instance diagram of Figure 4.10 best illustrates the idea.

Figure 4.9 Entries in each column must be unique.

Figure 4.10 A one-to-one relation.

It is possible that the same Politician visited **more than one** Country
and that the same Country was visited by **more than one** Politician.

Figure 4.11 A politician may visit many countries, and vice versa.

Now consider the populated binary fact type shown in Figure 4.11. Note that no whole row of the fact table is repeated. This must be the case, since each row corresponds to an elementary fact, and we agreed not to repeat facts in our database.

Although each row is unique, we cannot say the same for each politician. For example, "George W. Bush" appears twice in the first column. Nor is the country unique. For example, "MX" (Mexico) occurs twice in the second column. For this fact type then, the only UC that applies to every one of its possible populations is that the *combination* of politician and country is unique. This uniqueness constraint is depicted by a uniqueness bar that spans both columns of the fact table. The constraint verbalizes as shown.

The sample data shows that a politician may visit many (i.e., at least two) countries, and a country may be visited by many politicians. The fact population is an example of a *many-to-many* (or *m:n*) relation. Figure 4.12 shows the instance diagram.

A uniqueness constraint that spans all roles in a fact type is called a *spanning uniqueness constraint*. In addition to expressing the *m:n* nature of the binary fact type, the spanning uniqueness constraint in Figure 4.11 ensures that the fact type is populated by a set of facts (no duplicate rows). This may be verbalized explicitly as "**Each** Politician, Country **combination occurs at most once in the population of** Politician visited Country".

Whenever you have a spanning UC, you should check with the domain expert that a bag (multiset) of facts is to be excluded. For example, a politician may visit the same country many times. If we wish to distinguish between visiting a country at all and visiting it many times, we need to remodel the situation (e.g., by expanding the binary to a ternary that includes a visit date). We'll discuss such temporal aspects of modeling in detail in Section 10.3.

Figure 4.12 A many-to-many relation.

Figure 4.13 A simple 1:1 naming convention shown explicitly.

Simple 1:1 reference schemes are usually abbreviated by placing the reference mode name in parentheses. However, we may set them out fully as in Figure 4.13. This diagram is still incomplete; the next chapter shows how to add the constraint that each country has a country code. Here we have an *existential fact type* (also called a *reference type*): instances are used to simply assert the existence of an entity.

If there is only one means of referring to an entity type, the uniqueness constraints on the reference type are the responsibility of the user to enforce rather than the system. For example, we must ensure that there really is only one country with the code "US". If two 1:1 naming conventions are used (e.g., code or name) we choose one for our preferred reference scheme, and treat the other like any other elementary fact type.

The four possible patterns of uniqueness constraints for a binary predicate are: many to many; many to one; one to many; and one to one. Each of our examples so far has involved a heterogeneous predicate (different object types). Let's look now at some ring predicates (both roles played by same object type). To clarify the different roles being played, inverse predicate names are included. Our first example concerns parenthood (see Figure 4.14). For simplicity, assume that we can identify people by their first names.

As the sample fact table in Figure 4.14 reveals, a person may have many children. Moreover a person may have many parents. The column entries need not be unique, but the whole row must be. So the UC spans the whole row. Contrast this with motherhood (see Figure 4.15).

Although a mother may have many children, each person has only one mother. So we mark the constraint as shown. Entries in the second column of the fact table must be unique within that column (not necessarily unique within the table; e.g., after she becomes an adult, Mary also becomes a mother).

Figure 4.14 Parenthood is many-to-many.

Figure 4.15 Each person has only one mother.

The next example shown in Figure 4.16 is consistent with monogamy: at any time each man has at most one wife, and each woman has at most one husband. So entries in each column must be unique.

As an exercise, draw a model for the following marriage conventions: polyandry (a woman may have many husbands but not vice versa); polygyny (a man may have many wives but not vice versa); and polygamy (a man may have many wives, and a woman may have many husbands).

Uniqueness constraints are best understood in the context of a populated schema diagram, as discussed in this section. With this understanding, for any given fact type *each role is associated with a corresponding column of the fact table*.

In terms of logical predicates, the role boxes are the "object holes". For a set of relationship instances of this type the holes expand to columns. Marking a *single role with a uniqueness bar* means that entries in the associated column must be unique in that column (i.e., *no duplicates are allowed in that column*).

A spanning UC requires that each fact row be unique in the table. This whole row constraint applies to any asserted fact type. If we have a stronger uniqueness constraint, then the whole row constraint is implied by this and hence is redundant. For this reason, we never mark the whole row constraint across a fact type unless that is the only uniqueness constraint that applies.

With this understanding, we must choose exactly one of the four constraint patterns shown in Figure 4.17 for any binary predicate. For brevity, *a* and *b* denote object variables, and "*a R*'s *b*" means "*a* bears the relation *R* to *b*".

Figure 4.16 Monogamy is one to one.

Figure 4.17 The four uniqueness constraint patterns for a binary.

Provided a significant example is supplied, the uniqueness constraints on a predicate can be determined simply by looking for duplicates in its fact table. If we are not familiar with the application, we may be unsure as to whether the population is significant. Consider for example the output report of Table 4.1.

Table 4.1 An output report of doubtful significance.

Reviewer	*Paper Nr*
Jones E	1
Smith JB	2

Suppose this is a sample output report for a conference management system. One task involves getting qualified people to review (assess the suitability of) papers submitted by people who hope to present a paper at the conference. Let us agree that the information in the table may be modeled using the fact type Person(.name) reviewed Paper(.nr). Which of the four binary uniqueness constraint patterns should be specified? If the population of Table 4.1 is significant then we have a 1:1 situation, and constraints should be marked on each role, as shown in Figure 4.18.

This means that each person reviews at most one paper (good news for the reviewers), and each paper is reviewed by at most one person (bad news for the people submitting the papers).

Figure 4.18 Is the population significant?

Figure 4.19 Adding counterexamples to test the constraints.

While this arrangement is possible, we would probably doubt whether this constraint pattern is really intended. To resolve our doubts, we should *question each constraint in English, backed up by counterexamples*.

Consider the UC on the first column. Its positive verbalization is: **Each** Person reviewed **at most one** Paper. Its equivalent, negative verbalization is: **It is impossible that the same** Person reviewed **more than one** Paper. To test this constraint, we ask "Is it possible for a person to review more than one paper? For instance, could Smith review papers 2 and 3?" This counterexample appears as the second and third rows of the expanded fact table in Figure 4.19.

To test the UC on the second column, ask "Is it possible for a paper to have more than one reviewer? For instance, could paper 1 be reviewed by Jones and Adams?" This counterexample appears as the first and fourth rows of the expanded fact table in Figure 4.19. If the domain expert replies yes to both these tests, the fact type is many to many, and we modify the constraint as shown in Figure 4.20.

If desired, a more compact population may be provided for the *m:n* case, by using the tuple ('Smith JB', 1) instead of the last two rows. This tests both constraints at once. For testing purposes however, it's normally better to test just one constraint at a time, as discussed. If just one of the two test cases were accepted, the final constraint pattern would be *n:1* or *1:n*. If both test cases were rejected, we would stay with the 1:1 pattern.

Remember that all constraints on fact types are to be interpreted as applying to the database, not necessarily to the real world. An information system can only enforce constraints on its formal model of the domain. The constraint schema for the "real world" need not be the same as for the model (or "recorded world").

For example, suppose the model includes the schema of Figure 4.21(a). The uniqueness constraints assert that for any particular state of the database, each patient is recorded as having at most one gender and at most one phone. Now it may be the case in the real world that some patients have more than one phone (e.g., a home phone and a business phone), as shown in the "real world schema" of Figure 4.21(b).

Figure 4.20 If the test examples are accepted, the predicate is many to many.

Figure 4.21 A UC in the application model (a) may be stronger than in the real world (b).

For some reason, the designer of the information system decided that no patient will be recorded to have more than one phone. Hence the constraints in the first schema (Figure 4.21(a)) fit the intended model that is to be implemented in the information system. Unless otherwise indicated, *when we speak of a conceptual schema, we mean the structure of the conceptual model (not necessarily the real world).*

Nevertheless, we often use our background knowledge of the "real world schema" when designing the conceptual schema. In any case, *the uniqueness constraints should be at least as strong as those that apply in the real world*. For the case being discussed we know that in the real world each person has at most one gender. So we should enforce this in our schema. With the phone fact type, we need to consciously decide whether a stronger constraint than the real world constraint is required.

Exercise 4.2

1. For a given fact type, a sample population is significant with respect to uniqueness constraints if all the relevant uniqueness constraints can be deduced from the sample. A template is shown for a binary fact type. The names of the predicate and the reference modes are omitted for simplicity. For each of the fact tables shown, add the uniqueness constraints to the template assuming the population provided is significant.

2. In a given department, employees are identified by employee numbers "e1", "e2", and so on, and projects are identified by project numbers "p1", "p2", and so on. Draw a schema diagram for the fact type Employee works on Project, and provide populations that are significant with respect to the following constraint patterns:

   (a) 1:many (b) many:1 (c) many:many (d) 1:1

3. Add the relevant uniqueness constraints to the conceptual schema diagrams for the Exercise questions listed below. For some of these questions, the output report provided in the question might not be significant with respect to uniqueness constraints. Using common sense, you should be able to avoid adding uniqueness constraints that are likely to be violated by a larger population.

(a) Exercise 3.4 Question 1b (b) Exercise 3.4 Question 1d (c) Exercise 3.4 Question 2
(d) Exercise 3.4 Question 5 (e) Exercise 3.5 Question 2 (f) Exercise 3.5 Question3

4. By now you must be feeling like a challenge. The following report is an extract from an actual Computer Science Department Handbook. Verbalize this report in terms of *binaries*, draw the fact types and add uniqueness constraints.

   The first column headed "Subject title" actually lists the codes and titles of postgraduate topics (don't expect all the names in real-life examples to be well chosen!). A topic is not the same kind of thing as a subject. A postgraduate student first enrolls in a subject (identified by subject code, e.g., CS451) and then later chooses a topic to study for this subject. Subject enrollments are not part of this question. A topic may be offered in semester 1 only, in semester 2 only, or over the whole year (at half the pace).

   The fact that different students might be enrolled in the same subject (as identified by subject code) and yet be studying different topics is a source of possible confusion. As information modelers, one of the most significant contributions we can make is to suggest improvements to the UoD itself. In this example, we might argue that it would simplify things if topics were treated as subjects. The department involved eventually accepted this change. For this exercise however, you are to model things as they were.

Postgraduate coursework topics in Computer Science

Subject title	*Staff*	*When*	*Prerequisites*
AIT: Advanced Information Technology	MEO, PNC, TAH	1st sem	CS315
CLVS: Computational Logic and Verification Systems	JS, PJR	Year	CS260 preferred
DBMS: Advanced topics in Database Management	MEO, PNC, RC, TAH	2nd	CS315
FP: Functional Programming	EJS, PAB	Year	CS225 CS220 preferred
GA: Geometric Algorithms	PDE	1st	CS340

N.B. Topics are offered subject to the availability of staff and to there being sufficient demand for a topic. In each case the first-mentioned member of staff is lecturer-in-charge of the subject. Preferred prerequisites may become mandatory in later years.

4.3 Uniqueness Constraints on Longer Fact Types

This section shows how to specify uniqueness constraints on fact types of arity 3 and beyond, including nested versions. Let's begin with an example. Figure 4.22 shows a populated ternary fact type of the form: Person scored Rating for Subject. For simplicity, assume that persons are identified just by their surname. Even if we are not familiar with the example, sample populations supplied by the domain expert can reveal that certain constraints don't apply.

Looking at the fact table, first consider each column individually. Each column has at least one value that is repeated. So no column by itself has a uniqueness constraint. As shown later, this must be true of any ternary fact type that is elementary.

Next let's look at pairs of columns. For ease of reference here, the columns are numbered 1, 2, 3 in the order shown in the diagram. Beginning with columns 1 and 2, note that the pair (Adams, 7) is repeated. With columns 1 and 3, each pair is unique. That is, each (Person, Subject) combination occurs on only one row of the table. With columns 2 and 3 the pair (7, PD102) is repeated.

There are only three ways to pair columns in a ternary. If the population of the fact table is significant, the only pair-unique constraint is that each (Person, Subject) pair must be unique to the table. If familiar with the domain, we can usually decide whether the population is significant in this regard simply by using common sense. With this example, this constraint means that each (Person, Subject) combination, has at most one rating. This agrees with our background knowledge about the UoD and so the UC suggested by the table is accepted.

Sample fact tables may be obtained from output reports (which often represent a combination of separate fact tables) or from simple factual knowledge about the UoD. Unless the population is large, or well chosen, it is unlikely to be completely significant with respect to all uniqueness constraints. For instance, suppose row 3 was deleted from the table in Figure 4.22. For the smaller sample, both the column 1-2 pair and the column 2-3 pair would then show no duplicates. The table would then suggest a UC for all three column pairs. Apart from *insufficient data*, a population may fail to be significant because of *bad data* (i.e., some incorrect values have mistakenly been entered as data in the example).

So we need to be wary of relying on just a sample fact table to determine uniqueness constraints. We should *ask whether any suggested constraint really makes sense*. A uniqueness constraint spanning columns 1 and 2 would mean that a (Person, Rating) combination could occur for at most one subject. This would forbid somebody obtaining the same rating for two subjects—an unrealistic restriction! Similarly, the suggested constraint on the column 2-3 pair must be rejected as unrealistic (it would prevent two people from scoring the same rating for a particular subject).

Adams	7	CS112
Adams	6	CS100
Adams	7	PD102
Brown	6	CS112
Brown	7	PD102

Figure 4.22 What are the uniqueness constraints?

Figure 4.23 Each (Person, Subject) combination is unique.

Uniqueness constraints may often be determined simply by background knowledge of the UoD. As seen earlier, real world uniqueness constraints usually determine the weakest uniqueness constraints to be considered for the actual database. If we know the fact table is significant then we can generate the constraints from it.

Usually however, we won't know in advance that the table is significant, and we will have to use some of our background knowledge. If we lack such knowledge, we should consult the domain expert about any doubtful constraints, posing the relevant questions while adding concrete examples of facts to violate the constraints and asking whether such updates may be accepted.

With a ternary fact type, the constraint spanning all three columns is always implied, since no whole row may be repeated. This constraint should be specified if and only if no shorter UC holds. So for our current example, we mark just the constraint for the column 1-3 pair, indicating that each (Person, Subject) pair is assigned at most one rating.

Since the two role boxes involved are not adjacent, a *divided constraint bar* is used as shown in Figure 4.23. A *dotted line* connects the two active parts of the constraint bar, indicating that the constraint excludes the role under the dotted line. If we omitted the dotted line, the bars would instead denote two separate constraints, one for each end role. If we reordered the fact type instead as "Person for Subject scored Rating", there would be two contiguous roles spanned by an undivided constraint bar.

The term *n-role constraint* denotes a constraint that spans *n* roles. A uniqueness constraint that spans just one role is a **simple** uniqueness constraint. The constraint in Figure 4.23 is a two-role constraint since it spans two of the three roles involved in the fact type. A binary fact type has three possible basic constraints (two one-role and one two-role), which give rise to four possible cases. A later section demonstrates that *no (elementary) ternary fact type can have a simple uniqueness constraint*. So a ternary fact type has four basic uniqueness constraints to be considered: three two-role constraints and one three-role constraint, as shown in Figure 4.24.

In practice, ternaries usually have just a single UC either over one role pair, or over all three roles. As an example of the latter, a politician travel example is modeled in Figure 4.25.

Figure 4.24 Allowed basic uniqueness constraints for a ternary.

George W. Bush	CA	2005
George W. Bush	CA	2006
George W. Bush	MX	2006
Kevin Rudd	MX	2006

Figure 4.25 A ternary with a spanning uniqueness constraint.

Here many politicians may visit many countries in many years. Such a fact type is said to be "many to many to many". Since no whole row can be repeated, we cannot use this fact type to record multiple visits to the same country by the same politician in the same year. If you want to record this, use a different model (e.g., replace Year by Date, or add Quantity).

With a ternary fact type, we should systematically test each of the three 2-role constraints to see which ones hold. Only if none of these hold, should the 3-role constraint be specified. As an exercise, check for yourself that the sample data is significant for this constraint.

For a ternary association, there is only one way to have a three-role uniqueness constraint. There are three ways of having precisely one two2-role constraint, and three ways of having exactly two two-role constraints. Finally there is one way of having three two-role constraints. Thus there are eight different uniqueness constraint patterns that may arise with a ternary fact type. Four of these involve just one constraint (Figure 4.24) and four involve combinations (see Figure 4.26).

All other constraint patterns for a ternary are illegal (i.e., disallowed). Some examples of these are shown in Figure 4.27. The first two of these are fundamentally wrong, since a ternary fact type with a simple UC cannot be elementary. The third example is wrong because the three-role constraint is implied by the two-role constraint and hence should not be displayed.

Figure 4.26 Allowed uniqueness constraint combinations for a ternary.

Figure 4.27 Some illegal constraint patterns.

Figure 4.28 Each *acd* combination occurs on at most one row.

This general approach for specifying uniqueness constraints applies to fact types of any arity. Figure 4.28 shows a template for a quaternary predicate with a UC spanning roles 1, 3, and 4. This indicates that each *acd* combination must be unique to the fact table, where *a*, *c*, and *d* occur in the first, third, and fourth columns on the same row of this table.

A small but significant population of the fact table is shown. Verify for yourself that this is consistent with the indicated uniqueness constraint.

As a concrete example of a quaternary, consider Figure 4.29. The uniqueness constraint spans the first, second and fourth roles, and verbalizes as: **For each** Person$_1$, Person$_2$ **and** Language, **that** Person$_1$ judged **that** Person$_2$ to have **at most one** SkillLevel in **that** Language. Note the use of subscripts to distinguish the different roles played by Person.

It is shown later that any uniqueness constraints on a quaternary must span at least three roles. As an exercise, you might like to explore all the possible cases. Or maybe not! Since elementary fact types are rarely longer than quaternaries, no examples of quinaries or longer fact types are discussed here.

To visualize the connection between roles and columns of the fact table, think of the role boxes as the holes in which the objects get placed to complete the fact. No matter what the arity of the fact type, a UC across a combination of role boxes means that any instance of that column combination that does appear in the table must occur on one row only, and hence can be used to identify that row.

Since a role combination governed by a UC thus provides a "key to unlock" any row of the fact table, this combination is sometimes referred to as a *key* for that table. This notion will be elaborated later in the context of the (often non-elementary) tables used in relational databases.

Figure 4.29 A uniqueness constraint over roles 1, 2, and 4.

Figure 4.30 The enrollment association is objectified as a nested object type.

Now let's consider *nesting*. In Figure 4.23 the subject ratings example was set out in flattened form. Suppose that instead we use a nested approach as shown in Figure 4.30. Here each (Person, Subject) enrollment is treated as an object that may result in a rating. For example, the enrollment of Adams in CS112 results in a 7.

Figure 4.23 is not actually equivalent to Figure 4.30 unless a further constraint is added that a rating must be known for each enrollment. Such mandatory role constraints are discussed in the next chapter.

The Enrollment entity type in Figure 4.30 may be referred to as an objectified association, an objectified relationship type, an objectified predicate, a nested entity type or a nested object type. In most cases, objectified associations are fully spanned by a uniqueness constraint (as in Figure 4.30). This spanning uniqueness constraint indicates that each person may enroll in many subjects, and the same subject may be enrolled in by many people. The simple uniqueness constraint on the scored predicate indicates that each (Person, Subject) enrollment resulted in at most one rating.

Later we discuss nesting of associations that have non-spanning uniqueness constraints (e.g., 1:1 associations and *n*:1 associations). On the surface, such cases violate elementarity, but they can be transformed internally to avoid any problems.

Exercise 4.3

1. A template for a ternary fact type is shown. For each of the fact tables provided, add the uniqueness constraints, assuming that the population is significant in this regard.

(a)

a1	b1	c1
a2	b1	c1
a1	b2	c2

(b)

a1	b1	c1
a1	b2	c2
a2	b1	c2

(c)

a1	b1	c1
a2	b2	c1
a1	b1	c2
a1	b2	c1

(d)

a1	b1	c1
a2	b2	c2
a1	b2	c2
a2	b2	c1

2. Verbalize and then schematize the following report, marking uniqueness constraints.

Department	*Staff category*	*Number*
CS	Professor	3
CS	Senior Lecturer	6
CS	Lecturer	10
EE	Professor	3
EE	Senior Lecturer	7
EE	Lecturer	7

3.

Year	*Branch*	*Profit ($)*	*Total profit ($)*
2007	New York	200 000	
	Paris	90 000	290 000
2008	New York	300 000	
	Paris	150 000	450 000

(a) Verbalize and then schematize this report without nesting. Include UCs.
(b) As for (a) but instead use a nested approach.

4.4 External Uniqueness Constraints

The uniqueness constraints discussed so far are called *internal* (intrapredicate) uniqueness constraints, since each applies to one or more roles inside a single predicate. We now discuss **external** (interpredicate) **uniqueness constraints.** These *apply to roles from different predicates*. To help understand these constraints, and prepare for later discussion on queries, the *conceptual join* operation is first discussed.

Consider the schema and sample population shown in Figure 4.31. The left-hand binary fact table indicates that employee 15 drives two cars, with registration numbers PKJ123 and ABC000. The right-hand binary fact table tells us that the car PKJ123 is imported from Italy. The ternary fact table indicates that employee 15 drives the car PKJ123 imported from Italy. As the definition reveals, this ternary is derived by combining the binaries, with Car as the connecting or join object type.

Figure 4.31 A compound ternary derived from a conceptual (inner) join of the binaries.

The defining expression "Employee drives Car **that** was imported from Country" is a formal verbalization of the conceptual path from Employee, through the drives predicate, Car type, and import predicate, to Country. The keyword "**that**" declares that the driven car must be *the same* as the imported car. In other words, if an employee drives a car, we must use the very same car to continue the path to Country.

The instance diagram in Figure 4.32 may help clarify things. The three binary facts appear as lines connecting two dots. The ternary fact corresponds to the continuous path from Employee to Country. Starting at the Car type, this path may also be verbalized as the conjunction: Car is driven by Employee **and** was imported from Country.

A fact type resulting from a join is a *compound fact type*, not an elementary fact type, since it is essentially the conjunction of at least two simpler fact types. For this reason, *join fact types are normally excluded from ORM schema diagrams*. If included for discussion purposes, they may be *shaded* as shown in Figure 4.31 to indicate that they are just views. Unlike elementary derived fact types, their compound nature can lead to redundancy unless each join role is covered by a simple UC.

For example, if we add the row (16, PKJ123) to the drives table in Figure 4.31, this causes the row (16, PKJ123, IT) to be added to the ternary. The fact that car PKJ123 is imported from Italy is now contained twice in the ternary. Although join fact types are never used as asserted fact types, join paths are often involved in constraints and queries, as discussed later.

In an ORM schema, to navigate from one predicate to another, we must pass through an object type, performing a *conceptual join* (or *object join*) on that object type. By default, the join condition is that the object instance remains the same as we pass through. This is called a *conceptual inner join*. This is similar to a relational natural inner join (see Chapter 12), except that conceptual joins require the conceptual objects to be the same, instead of matching attribute names and values. ORM object types are conceptual domains, not attributes. Conceptual joins still apply if we declare different role names (cf. attribute names) for the join roles.

In terms of the fact tables however, the column value of the role entering the join object type must equal the column value of the role that exits the object type. Figure 4.33 depicts two predicates, *R* and *S*, sharing a common object type, *B*. From a fact table perspective, the conceptual inner join of *R* and *S* is obtained by pairing rows of *R* with rows of *S* where the *B* column values match, and arranging for the final *B* column to appear just once in the result. The dotted lines show how the rows are paired together. For example, row (a1, b1) is paired with rows (b1, c1) and (b1, c3) to give rows (a1, b1, c1) and rows (a1, b1, c3).

Figure 4.32 An instance diagram for the model shown in Figure 4.31.

Figure 4.33 Joining on *B* requires the values in the two *B* columns to match.

Figure 4.34 Table resulting from the natural inner join of the tables in Figure 4.33.

Figure 4.34 shows the result of this join operation. The final *B* column is the intersection of the original *B* columns. The natural inner join of any tables *R* and *S* is denoted by "$R \bowtie S$" or "*R* natural join *S*". The "inner" is often omitted and assumed by default.

Other kinds of joins are sometimes discussed (e.g., joins based on operators other than equality, and outer joins). A left (right, full) *outer join* is obtained by adding to the inner join those rows, if any, where the join column value occurs in just the left (just the right, just one of) the tables, and padding the missing entries of such rows with nulls. For example, the left outer join of the binary fact tables in Figure 4.31 adds the row (15, ABC000, ?) where "?" denotes a null.

The full outer join of the tables in Figure 4.33 includes rows (a2, b2, ?) and (?, b4, c4). Outer joins are often used in queries at both conceptual and relational levels. Since asserted fact types in ORM are elementary, their fact tables cannot have nulls. We postpone further discussion of outer joins until Chapter 10.

With this background, let's consider an example with an external uniqueness constraint. An output report concerning high school students is shown in Table 4.2. As an exercise, perform step 1 for the top row of this report before reading on.

The information for row 1 may be verbalized as two elementary facts:

The Student with studentNr '001' has the StudentName 'Adams J'.
The Student with studentNr '001' is in the Class with code '11A'.

Table 4.2 An output report about high school students.

Student Nr	*Name*	*Class*
001	Adams J	11A
002	Brown C	12B
003	Brown C	11A

Figure 4.35 A populated schema for Table 4.2 (draft version).

The first fact relates an entity and a value. The second fact is a relationship between two entities. In this UoD, students are identified by their student numbers. As rows 2 and 3 show, it is possible for two different students to have the same student name ("Brown C"). A populated schema diagram is shown in Figure 4.35.

Let us agree that the population supplied is significant. It follows that each student has at most one name, and that each student is in at most one class. These uniqueness constraints have been captured on the schema diagram. However, there is another uniqueness constraint that is missing. What is it?

The output report is reproduced in Table 4.3, with uniqueness constraints marked above the relevant columns. Notice that the combination of name and class is unique. Let's suppose that this is significant. Although a student's name need not be unique in the high school, it is unique in the student's class. While there are two students named "Brown C", there can be only one student with this name in class 12B, and only one student with this name in class 11A.

In the rare case where another student named "Brown C" joined one of these classes, at least one of the names would be modified to keep them distinct within that class. For instance, we could add extra initials or numbers (e.g., "Brown CT", "Brown C2").

Each row of the report splits into two elementary facts, one for the name and one for the class. So to specify the name–class UC on the schema we need to involve two fact types. The role boxes to which the uniqueness constraint applies are joined by dotted lines to a *circled uniqueness bar* "⊖" as shown in Figure 4.36.

Because this uniqueness constraint involves roles from different predicates, it is an example of an *external constraint* or interpredicate constraint. A constraint on a single predicate is an *internal* or intrapredicate constraint. So there are two kinds of uniqueness constraint: internal uniqueness and external uniqueness.

Table 4.3 Uniqueness constraints added to original table.

Student Nr	*Name*	*Class*
001	Adams J	11A
002	Brown C	12B
003	Brown C	11A

Figure 4.36 An external uniqueness constraint has been added.

In this example, the external uniqueness constraint indicates that for each student the *combination* of student name and class is unique. The constraint verbalizes as: "**Given any** StudentName **and** Class, **at most one** Student has **that** StudentName **and** is in **that** Class". For instance, given 'Adams J' and '11A' there is only one studentNr sharing rows with both ('001'). Given 'Adams J' and '12B' there is no studentNr paired with both.

Perhaps the easiest way to understand this constraint is to say that if we perform the conceptual (inner) *join* operation on the two fact tables, then the resulting table has a uniqueness constraint across the (name, class) column pair. Note that when we join the two fact tables, we obtain the table in the original report (Table 4.3)

The external uniqueness constraint is equivalent to an internal uniqueness constraint on the first and last roles of the derived, ternary fact type formed from the join path: StudentName is of Student **who** is in Class. See Figure 4.37. Because the object type Student is personal, we prefer the personal pronoun "who" instead of "that".

The middle role of the join fact type has a simple UC. This is equivalent to the two simple uniqueness constraints on the binaries. Although join fact types may be included on a schema diagram for discussion purposes, they are illegal in the base ORM model because they are compound, not elementary, fact types.

Figure 4.37 Equivalent constraints on derived, join fact type (illegal in base model).

Figure 4.38 This update would violate the external uniqueness constraint.

As discussed shortly, the presence of a simple uniqueness constraint on a ternary disqualifies it from being elementary. So don't feel that it's okay to do this in your normal model.

Suppose we added the following facts to the current populations: Student '004' has StudentName 'Adams J'; Student '004' is in Class '11A'. If this update were accepted, the new populations would be shown in Figure 4.38(a). To test your understanding, explain why this would violate the external uniqueness constraint before reading on.

The extra row added to each table provides a counterexample to the constraint, since for 'Adams J' and '11A' there are two studentNr entries paired with these entries (001 and 004). This breaks the rule that student names are unique within a given class. In class 11A, two students (001, 004) have the same name ('Adams J').

If we join the tables on studentNr (the dotted lines indicate the matches for the join), we obtain the ternary table in Figure 4.38(b), which clearly violates the equivalent compound uniqueness constraint over the StudentName and Class columns.

The general case is summarized in Figure 4.39, where *A*, *B* and *C* are any object types, and *R* and *S* are predicates. External uniqueness constraints may apply to roles from reference types, not just fact types. This is the basis for composite reference schemes, as discussed in a later chapter.

As a more complex case, suppose persons enroll in subjects but are given subject positions (1st, 2nd, 3rd etc.) rather than subject ratings. Each person achieves at most one position in any given subject. Moreover, no ties may occur (i.e., for each position in a subject there is only one student).

Figure 4.39 External uniqueness constraint.

Table 4.4 Student ranking within subjects (no ties allowed).

Person	*Subject*	*Position*
Adams J	CS114	3
Adams J	CS100	10
Adams J	PD102	3
Brown C	CS114	10
Brown C	PD102	5

Table 4.4 shows a sample report for this UoD. Performing step 1 on the first row, we may express the information as the ternary: Person (name) 'Adams J' is placed in Subject (code) 'CS114' at Position (nr) 3. This leads to the schema of Figure 4.40. Notice the overlapping uniqueness constraints. Check these with the population for yourself to ensure that you understand them.

Now suppose we adopt a nested approach instead. For example we might express the information on row 1 as follows: Person (name) 'Adams J' enrolled in Subject (code) 'CS114'; this Enrollment achieved Position (nr) 3. This leads to the nested version shown in Figure 4.41. Actually, for this to be equivalent to the ternary, the role played by the objectified association must be mandatory (see next chapter).

The Enrollment predicate is many:many. With this in mind, the simple uniqueness constraint on the achievement predicate captures the UC spanning the first two roles in the flat (unnested) version (Figure 4.40). The external uniqueness constraint corresponds to the constraint spanning the last two roles in the flattened version—each (Subject, Position) combination is unique. In other words, if we *flatten* the nested version into a ternary then any (Subject, Position) pair occurs on at most one row of the ternary table. To help understand this, we suggest that you add the fact tables to the diagram. The fact table for the outer predicate effectively matches the output report.

Figure 4.40 A schema for Table 4.4.

Figure 4.41 Another schema for Table 4.4 (nested version).

Table 4.5 Students are assigned both ratings and positions.

Person	*Subject*	*Rating*	*Position*
Adams J	CS114	7	3
Adams J	CS100	6	10
Adams J	PD102	7	3
Brown C	CS114	6	10
Brown C	PD102	7	5

External uniqueness constraints sometimes connect more than two roles. For example, a point in three-dimensional space might be associated with a unique combination of x, y, and z coordinates.

As you may gather from these examples, nesting tends to produce a more complex constraint picture. For this example, the flattened version is preferred. However, if the Position information is optional (e.g., to be added later) then the nested approach is preferred. Such modeling choices are covered in more detail later.

Another reason for nesting is to avoid embedding the same association within more than one fact type. Suppose we have to record both a rating and a unique subject position for each student taking any given subject. A sample report for this situation is shown in Table 4.5.

We might describe this UoD using two ternaries, as in Figure 4.42. Note however that all (Person, Subject) combinations appearing in one fact table must also appear in the other. A later chapter shows how such a constraint may be added.

Alternatively, we may objectify the enrollment association between Person and Subject, and attach rating and position predicates to this (see Figure 4.43). The nested approach corresponds to reading the information on row 1 of the table as: Person 'Adams J' enrolled in Subject 'CS114'; this Enrollment scored Rating 7; this Enrollment achieved Position 3. To indicate that rating and position must be recorded, additional constraints are needed (see next chapter).

Note that if a role has a simple, internal uniqueness constraint then it should not be included in an external uniqueness constraint. Even if a join path exists to enable an external uniqueness constraint to be declared, the external constraint would be implied by the stronger, simple uniqueness constraint; it is generally preferable to omit implied constraints.

Figure 4.42 A schema for Table 4.5 (flattened version).

Figure 4.43 Another schema for Table 4.5 (nested version).

As an exercise to illustrate this point, you may wish to modify Figure 4.36 so that student names are actually unique, and then show that the external constraint is implied.

Before you start the section exercise, there's one point you should be clear about. When you examine an information sample, you are seeing it at the external level. *There are often many aspects of the presentation that may not be relevant to your application*. In a tabular report for example, the order of the columns or rows is not normally something that you need to model. If the information is presented in nontabular ways (e.g., diagrammatic) there are always topological and metric aspects, and possibly other aspects (e.g., color), that are represented along with basic data. Whether these additional aspects of the presentation need to be modeled is something that only you and the domain expert can decide. To fully appreciate this point, make sure that you attempt at least Question 2 from the following exercise.

Exercise 4.4

1. Add the uniqueness constraints to the conceptual schema diagrams for:
   (a) Exercise 3.4 Question 6 (b) Exercise 3.5 Question 4

2. Many manufactured products contain parts that may themselves be products of even smaller parts. The structure of one such product is shown.

(a) List at least three different kinds of fact that are captured in this diagram.
(b) Assume that we are interested in modeling only containment facts. These facts may also be displayed as a labeled tree (hierarchy) as shown. Draw a conceptual schema diagram for this UoD, including uniqueness constraints, but make no use of nesting.

(c) Draw an equivalent conceptual schema diagram that does make use of nesting.

3. A car dealer maintains a database on all the cars in stock. Each car is identified by the vehicle identification number (VIN) displayed on a plate attached to the car (e.g., on its dashboard). For each car the dealer records the model (e.g., Saturn SW2), the year of manufacture (e.g., 2006), the retail price (e.g., $18,000), and the color (e.g., dark green).

   Because of space limitations the dealer will never have in stock more than one car of the same model, year, and color at the same time. The dealer also keeps figures on the number of cars of a particular model and color that are sold in any given year. For example, in 2007, five dark green Saturn SW2s were sold. Draw the conceptual schema diagram, including all uniqueness constraints.

4.5 Key Length Check

In step 1 of the CSDP we try to express information examples in terms of elementary facts. At that stage we rely on familiarity with the UoD to determine whether a fact type is simple or compound (splittable). Once uniqueness constraints are added, a formal check can be conducted in this regard. This section discusses a check based on uniqueness constraints, and the next section discusses the notions of projection and join, which may sometimes be used to perform a projection–join check.

Until we are experienced at conceptual schema design, we might include some fact types that are too long or too short. "Too long" means that the arity of the fact type is higher than it should be—the predicate has too many roles. In this case we must split the compound fact type up into two or more simple fact types. For example, the fact type Scientist was born in Country during Year should be split into two fact types: Scientist was born in Country; Scientist was born during Year.

"Too short" means the arity of some fact types is too small, resulting in loss of information—a fact type has been split even though it was elementary. In this case, the fact types resulting from the split need to be recombined into a fact type of higher arity.

Figure 4.44 A fact type has one or more keys.

For example, suppose that scientists may lecture in many countries in the same year. It would then be wrong to split the fact type Scientist lectured in Country during Year into the fact types: Scientist lectured in Country; Scientist lectured during Year. This kind of error is rare, but serious nevertheless.

For a given fact type, a minimal combination of roles spanned by a UC is called a "*key*" for that fact type. "Minimal" means the uniqueness constraint has no smaller UC inside it; if it did, it is implied by the smaller (but stronger) constraint and hence should not be shown. Role combinations for such longer, implied uniqueness constraints are called "proper superkeys". However, as proper superkeys are not minimal, we do not count them as keys in our discussion.

Each role is associated with a column of the fact table for the fact type. Unless the fact type is unary, it is possible for it to have more than one key. Consider the cases shown in Figure 4.44. Here the binary has two one-role keys, the middle ternary has two two-role keys, and other ternary has one three-role key.

The first two predicates also have implied uniqueness constraints across their whole length, but these are not counted as keys for these predicates. The *length* of a role sequence is the number of roles it contains. A key of length 1 is a *simple key*. All other keys are *composite*.

We can imagine unaries or binaries that should be split, but in practice not even a novice designer is likely to verbalize one (e.g., "The Barrister with surname 'Rumpole' smokes and drinks"). So the question of splittability is really an issue only with ternaries and longer fact types.

To help decide when to split in such cases, the so-called "*n – 1 rule*" may be used: *each n-ary fact type has a key length of at least n – 1*. If this rule is violated, the fact type is not elementary, and hence should be split. Various cases of this rule are set out in Table 4.6. A proof is sketched later in the section.

If a fact type is elementary, all its keys must be of the same length. Either there is exactly one key, which spans the whole fact type, or there are one or more keys and each is one role shorter than the fact type. So, if two or more roles in a predicate are not part of a key, the fact type is compound and must be split.

Table 4.6 Key length check for ternaries and longer fact types.

Arity of fact type	*Minimum key length*	*Illegal key lengths*
3	2	1
4	3	1, 2
5	4	1, 2, 3
n	$n-1$	$1, \ldots, n-2$

Table 4.7 Country data.

Country	*Population*	*Area (sq. km)*
Australia	20 600 856	7 686 850
Germany	82 369 548	356 910
United States	303 824 646	9 372 610

To start with the simplest case, *a ternary fact type can be split if it has a simple key*. Consider the sample extract of geographic data shown in Table 4.7. The first row of data may be verbalized: "The Country with name 'Australia' has a Population of 20,600,856 people and an Area of 7,686,850 square kilometers". The presence of "and" suggests that the fact is splittable. Suppose, however, that we ignore this linguistic clue and model using the ternary: Country has Population and Area.

Our familiarity with the UoD tells us that, at any given time, each country has at most one population and at most one area. So there is a uniqueness constraint on the Country column of the fact table. Although the populations and areas in Table 4.7 are unique, our common sense indicates that this is not significant. For example, the population of India is projected to soon equal China's population, and it is certainly possible to have two countries of the same size. The combination of population and area need not be unique either. This leaves us with the simple UC on the Country role as shown in Figure 4.45.

Now let's apply the $n - 1$ rule. Here the key has length 1, which is 2 less than the length of the fact type. So the fact type is not elementary and must be split. But *how do we split it?* Examining our verbalization of row 1, we find a conjunction of two facts about the same country. Assuming that our interpretation of the output report is correct, we must split the fact type into two fact types as shown in Figure 4.46. Since the common entity type is Country, the original ternary fact type is "split on Country".

Figure 4.45 This ternary is a compound fact type, and hence splittable.

Figure 4.46 The previous ternary splits into these two elementary fact types

If we have captured the semantics of the output report in step 1 correctly, then it would be wrong to split this fact type in any other way. Before considering this point further, let's see why the splittability rule works in this case.

Look back at Table 4.7. A uniqueness constraint on the Country column means that, given any state of the database, no country can be listed more than once in this column. Each country has at most one population and each country has at most one area. So for any given country, if we are given as separate pieces of information that country's population and that country's area we can reconstruct that country's row in the output report. Since the ternary fact type of the output report can be split and recombined without information loss, it is not elementary.

If a fact type violates the $n - 1$ rule, it is essentially a conjunction of smaller fact types even if its predicate has no conjunctive operator such as "and". Suppose we verbalized the previous ternary as Country with Population has Area. This still means the same, so is really a conjunction in disguise. Similarly, the ternary "Invoice was issued to Customer on Date" has a uniqueness constraint on its first role and can be split into two fact types: Invoice was issued to Customer; Invoice was issued on Date. In general, *if a fact type can be rephrased as a conjunction, it should be split*, since it is not elementary.

Before considering other examples, let's review some terminology used in the normalization approach to database design. For a given fact table, let *X* denote a single column or combinations of columns, and let *Y* denote a single column. Then we say that *Y* is *functionally dependent on X* if and only if, for each value of *X* there is at most one value of *Y*. In other words, *Y* is a *function of X*. A function is just a many:1 relation (including 1:1 as a special case). If *Y* is a function of *X*, we say that *X* functionally determines *Y,* written $X \rightarrow Y$. This constraint is a **functional dependency (FD)**. Although FD arrows are not used directly in ORM schemas, the theory of FDs provides another way to understand uniqueness constraints.

Now consider the sample output report shown in Table 4.8. If this population is significant, then any functional dependencies should be exposed here. Note that Degree is functionally dependent on Person, since for any given person there is only one degree. Moreover, Gender is a function of Person since for any given person there is only one gender. But can you spot any other functional dependencies?

If the population is significant, then Gender is a function of Degree, and Degree is a function of Gender! Does this make sense? To answer this question we need to know more about the semantics of the UoD. Spotting a functional dependency within a population is a formal game. Knowing whether this dependency reflects an actual constraint in the UoD is not. To assume the population is significant begs the question. Only someone who understands the UoD, a domain expert, can resolve the issue.

Table 4.8 Is the population significant?

Person	*Degree*	*Gender*
Adams	BS	M
Brown	BA	F
Collins	BS	M

Table 4.9 A counterexample to any FD between degree and gender.

Person	*Degree*	*Gender*
Adams	BS	M
Brown	BA	F
Collins	BS	M
Davis	BS	F

In Section 14.6, we extend the traditional treatment of dependencies on individual table populations to dependencies representing actual constraints in the business domain (and hence applying to all possible populations of the table schemes). Some of the following discussion anticipates this extension.

If one column functionally determines another column, this reflects a many:1 association (or 1:many depending on the direction of reading) that can be given a meaningful name by the domain expert. Normally we would interpret the table in terms of degree and gender facts about specific persons. But suppose instead that the domain expert says row 1 of the output report should be read as "The Person with surname 'Adams' seeks the Degree with code 'BS', and the Degree with code 'BS' is sought by people of Gender with code 'M'". With this unusual interpretation, we have two facts about the same degree. Hence we split on Degree rather than Person (see Figure 4.47).

If Gender really is a function of Degree, then any given degree is restricted to one gender (e.g., only males can seek a BS). If Degree really is a function of Gender then all people of a given gender seek the same degree (e.g., all females seek just a BA). These two FDs are captured in Figure 4.47 by the two uniqueness constraints on the relationship type between Degree and Gender.

While such a UoD is possible, our knowledge about degree awarding institutions makes this interpretation highly unlikely. To resolve our doubts, we could add test rows to the population. Although this is best done on conceptual fact tables, we could also test this example using the original output report. For example, if the domain expert accepts the population of Table 4.9 as being consistent with the UoD, then no FDs occur between Degree and Gender.

This by itself does not guarantee that Gender should be related to Person rather than Degree. For example, the schema of Figure 4.47 with the 1:1 constraint changed to a many:many constraint is still a remote possibility; but it would be silly to have a report with so much redundancy (e.g., the fact that the BS is sought by males would be shown twice already). The only way to be sure is to perform step 1 properly, verbalizing the facts and having them confirmed by the domain expert.

Figure 4.47 A weird UoD where Degree and Gender are functions of each other.

Table 4.10 University data.

Lecturer	*Department*	*Building*
Halpin	CS	69
Okimura	JA	1
Orlowska	CS	69
Wang	CN	1

While the notion of FD is useful, it is impractical to treat fact tables in a purely formal way, hunting for FDs. The number of dependencies to check increases rapidly with the fact type length. Although our conceptual focus on elementary fact types constrains this length, the search for FDs can still be laborious. While an automated FD checker can be of use, this kind of exhaustive checking is not fit for humans. We can short-circuit this work by taking advantage of human knowledge of the UoD.

As another simple example, consider the output report of Table 4.10. Suppose that, with the help of the domain expert, we express row 1 thus: the Lecturer with surname "Halpin" works for the Department coded "CS" which is located in the Building numbered "69". Is this fact splittable, and if so, how?

Here the pronoun "which" introduces a non-restrictive clause about the department that may be stated separately. So this can be rephrased as two facts about the department: Halpin works for the CS department; and the CS department is located in building 69. So we should split on Department to get two fact types (Figure 4.48).

As a challenge question, would it also be acceptable to split on Lecturer (e.g., Lecturer works for Department; Lecturer works in Building)? The logical derivation theory discussed in the next chapter shows that it is unwise to split on Lecturer. If the population is significant, then each lecturer works for only one department, and each department is located in only one building. If these constraints are confirmed by the domain expert, they are added to the schema as shown.

An outline proof of the $n - 1$ rule is now sketched. An n-ary fact type with a key of length less than $n - 1$ has at least two columns in its fact table that are functionally dependent on this key. Split the table by pairing the key with exactly one of these columns in turn. Recombining by join on this key must generate the original table since only one combined row is possible (otherwise the common key portion would not be a key). Although the n-ary fact table thus formally splits in this way, in practice the splitting might need to be made on part of the key or even a nonkey column if an FD applies there (consider the previous examples).

Figure 4.48 Populated conceptual schema for Table 4.10.

Table 4.11 Student data.

Person	*Degree*	*Subject*	*Rating*
Adams	BS	CS112 CS110 PD102	7 6 7
Brown	BA	CS112 PD102	6 7
Collins	BS	CS112	7

As an example of a *quaternary* case, consider the report shown in Table 4.11. Suppose we did a bad job at step 1 and schematized this situation as in Figure 4.49. Notice the uniqueness constraint. Here we have a fact type of length 4, and a key of length 2. *Since two roles are excluded from this key, we must split the fact type*. How do we split it? Each person seeks at most one degree, but this constraint is not captured by the two-role uniqueness constraint. Hence we split on Person. In terms of FD theory, the Person role functionally determines the Degree role (shown in Figure 4.49 as a solid arrow).

If an FD like this is involved, splitting takes place on its source. So the quaternary splits on Person into a binary (Person seeks Degree) and a ternary (Person for Subject scored Rating). As an exercise, draw these, being sure to include all constraints. In any correct ORM schema, *all FDs should be captured by uniqueness constraints*. So FD arrows are not used on final conceptual schema diagrams. Any quaternary splits if it has a key of length 1 or 2.

Now consider the populated schema in Figure 4.50. Suppose this interpretation of the UoD is correct. For example, suppose the first row of the output report behind this schema really does express the information "The Person with surname 'Adams' seeking the Degree with code 'BS' studies the Subject with code 'CS112'". Should the ternary be split?

Unlike the quaternary example, this ternary satisfies the *n*–1 rule. *Violating the n–1 rule is a sufficient condition for splittability, but it is not a necessary condition*. The ternary in this example does actually split. One way to see this is to realize that there is an *embedded FD* (i.e., an FD not captured by the uniqueness constraint(s)). In this case there is an embedded FD from the Person role to the Degree role, shown as an arrow in Figure 4.50.

Figure 4.49 A quaternary fact type that is splittable (FD added).

Figure 4.50 Is this ternary splittable?

However, *every FD corresponds to a uniqueness constraint on some predicate*, and it is better to verbalize this. Here the fact type behind the embedded FD is: Person seeks Degree. The remaining information can then be expressed with another fact type. Here that is Person studies Subject.

We can achieve the same effect without talking about FDs, by simply answering the question: *Can you rephrase the information in terms of a conjunction?* This approach was used earlier with the Country example. With our current example, instead of using the present participle "seeking", we can rephrase the information on the first row as:

> The Person with surname 'Adams' seeks the Degree with code 'BS'
> **and** studies the Subject with code 'CS112'.

which is equivalent to:

> The Person with surname 'Adams' seeks the Degree with code 'BS'
> **and**
> the Person with surname 'Adams' studies the Subject with code 'CS112'.

This is obviously a conjunction with Person common to each conjunct. So the ternary should be split on Person into two binaries (see Figure 4.51).

Figure 4.51 The ternary in Figure 4.50 should be split like this.

In general, given a significant fact table, the fact type is splittable if a column is functionally dependent on only some of the other columns. The phrase "only some" means "some but not all". This result corresponds to a basic rule in normalization theory (discussed in Section 12.6). With our present example, because the Degree column is functionally dependent on the Person column only, it should not be combined in a fact type with Subject information.

Another way to test for splittability is to *perform a redundancy check* by determining whether some fact is (unnecessarily) duplicated in the fact table. For example, in Figure 4.50 the pair (Adams, BS) occurs three times. Using our semantic insight, we see that this pair corresponds to a fact of interest (Adams seeks a BS degree). Since this fact has been duplicated, the ternary is not elementary.

Redundancy is not a necessary requirement for splittability. For example, the ternary in Table 4.9 has no redundancy but is splittable.

Whichever method is used to spot splittability, we still need to phrase the information as a conjunction of simpler facts. Notice how the redundancy within the ternary of Figure 4.47 has been eliminated by splitting it into two binaries in Figure 4.50. Although we may allow redundancy in output reports, we should avoid redundancy in the actual tables stored in the database.

Our examples so far have avoided nested fact types. Recall that nesting is not the same as splitting. If nesting is used, the following *spanning rule* can be used to avoid splittable fact types: *the uniqueness constraint on an objectified association should normally span all its roles.*

This rule simply adapts earlier results to the nested case. Suppose the objectified predicate has a UC over only some of its roles. If an attached, outside role has a uniqueness constraint on it, when the outer predicate is flattened we have at least two roles not spanned by a uniqueness constraint.

If an attached role has no UC, then flattening generates a predicate with an embedded FD not expressed as a uniqueness constraint (consider the cases of Figure 4.49 and Figure 4.50). In either case, the flattened version must split because it is not elementary. Hence to be elementary in this sense, the nested version must obey the spanning rule.

In the previous version of ORM, the only exception to the spanning rule was to allow 1:1 binaries to be objectified (e.g., Person is husband of / is wife of Person may be objectified as "Marriage").

The second generation ORM discussed in this book allows any fact type to be objectified, regardless of its uniqueness constraint pattern, and is similar to UML in this aspect (e.g. *n*:1 associations may be objectified). Although allowed for pragmatic reasons, such exceptions violate elementarity and require special treatment when mapped to logical models. Moreover, objectification without spanning uniqueness often makes the model more complicated than needed.

Consider the output report of Table 4.12. Here the credit for a given subject is measured in points. The "?" on the second row is a null, indicating that a real value is not recorded (e.g., a lecturer for CS109 might not yet have been assigned).

Table 4.12 Subject details

Subject	*Credit*	*Lecturer*
CS100	8	DBJ
CS109	5	?
CS112	8	TAH
CS113	8	TAH

Recall that elementary fact tables can't contain nulls (because you can't record half a fact). However, since output reports can group facts of different kinds in the one table, they often do contain nulls (indicating absence of a fact instance).

Now let's play the part of an inexperienced, and not very clever, schema designer, so that we can learn by the mistakes made. Looking just at the first row, we might be tempted to treat the information as a ternary:

Subject (.code) 'CS100' worth Credit (.points) 8 is taught by Lecturer (.initials) 'DBJ'.

If we drew the schema diagram now and populated it we would discover an error, since the second row doesn't fit this pattern (since we don't allow nulls in our fact tables). The second row reveals the need to be able to record the credit points for a subject without indicating the lecturer. So we then rephrase the first row as follows:

Subject (.code) 'CS100' is worth Credit (.points) 8.
This SubjectCredit is taught by Lecturer (.initials) 'DBJ'.

This overcomes the problem with the second row, since we can now express the information there simply as "Subject 'CS109' is worth Credit 5". Using this approach, we can now develop the schema shown in Figure 4.52.

Assuming that the population of the output report is significant, we mark the uniqueness constraints as shown. Note that a simple UC has been added to the objectified association. This breaks the spanning rule: an objectified association has no key shorter than itself. So something may be wrong. Since the schema follows from our handling of step 1, this means that we may have made a mistake at step 1.

Have another look at the way we set out the information for row 1. The second sentence here is the problem. In this UoD each subject has exactly one credit point value, so this is independent of who teaches the subject. As a result, there is no need to mention the credit point value of a subject when we indicate the lecturer. In FD terminology, the Credit and Lecturer columns are each functionally dependent on the Subject column alone.

Figure 4.52 A faulty schema.

Figure 4.53 The corrected schema for Figure 4.52.

The information on the first row can be set out in terms of two simple binaries:

Subject (.code) 'CS100' is worth Credit (.points) 8.
Subject (.code) 'CS100' is taught by Lecturer (.initials) 'DBJ'.

This leads to the correct schema shown in Figure 4.53. Note that Lecturer depends only on Subject, not Credit. The nested schema suggested incorrectly that Lecturer was dependent on both, by relating Lecturer to the objectified association. With this example, Lecturer is a function of Subject, but this has no bearing on the splittability. We should use two binaries even if the same subject can have many lecturers. It is the functional dependency of Credit on Subject that disqualifies the nested approach.

In a case like this where it is difficult to come up with a natural name for the objectified association, such nesting errors are unlikely to occur in practice. When a natural name for the nested object type is available however, it is easier to make the error. For example, we might objectify the fact type Person was born in Country as "Birth", and then add the fact type: Birth occurred on Date. Although it sounds reasonable, this nesting violates the spanning rule, and should normally be replaced by the unnested binary: Person was born on Date.

In some cases, it is reasonable to objectify associations with no spanning uniqueness (e.g., if the uniqueness constraint is likely to later change to a spanning constraint or if many related roles are played by the objectified association). We discuss such cases in Chapter 10 and provide heuristics for choosing when to objectify.

One lesson to be learned from this section is to look out for uniqueness constraints even when performing step 1. Consider the output report of Table 4.13. If told the population is significant, how would you model this?

If you used either a ternary or a nested fact type, you made a mistake! Although similar to an earlier example, the UoD described by this table is more restricted. Given that the population is significant, the lack of duplicates in the Person column indicates that, at any given time, each person can be enrolled in only one subject.

Table 4.13 Student results.

Person	*Subject*	*Rating*
Adams	CS112	7
Brown	CS112	5
Collins	PD102	5

For example, these people might be your employees, and you are funding their studies and want to ensure that they don't take on so much study that it interferes with their work performance.

Since we store information only about the current enrollments, and each person is enrolled in only one subject, it follows that each person can get only one rating. So if we know the subject in which a person is enrolled and we know the rating obtained by the person, we do know the subject in which this rating is obtained by that person. To begin with, we might express the information on the first row of the table in terms of the ternary "Person (surname) 'Adams' scores Rating (.nr) 7 for Subject (.code) 'CS112'. However, because of the uniqueness constraint on Person we should rephrase this information as the conjunction:

Person (surname) 'Adams' studies Subject (.code) 'CS112'
and
Person (surname) 'Adams' scores Rating (.nr) 7.

This leads to a conceptual schema with two binary fact types. Because of this unusual UC, the ternary has the same truth value as the conjunction of binaries for all possible states of this UoD.

Exercise 4.5

1. The keys for certain fact types are as shown. On this basis, which of these fact types are definitely splittable?

2. The following map provides information on nonstop flights between cities. The population is significant with respect to uniqueness constraints.

(a) In step 1a of the design procedure, a domain expert verbalizes the information on the NW378 arrow as: Flight 'NW378' goes from Seattle to Chicago. Assuming this interpretation is correct, complete steps 1 and 2 to draw a schema diagram for facts of this type and populate it with rows of data for the flights NW378, NW102, UA123, and UA246.
(b) Now complete the first part of step 4 by adding any uniqueness constraints.
(c) Is the fact type elementary? How do you know?
(d) Split the fact type into elementary fact types, and populate them with the data for the same four flights mentioned earlier.

3. Executives may be contacted at work on one or more phone numbers and at home on one phone number. The following report provides sample data. A novice modeler notes that this ternary table lacks a simple key, and on this basis decides to schematize with a single ternary fact type. Evaluate this approach, and draw a correct conceptual schema.

Executive	*Work phone*	*Home phone*
Adams A	4235402	5837900
Adams A	4235444	
Adams S	4235444	5837900
Brown T	4235300	7578051

4. The following table contains data about some moons in our solar system.

Moon	*Planet*	*Orbital period (days)*
Callisto	Jupiter	16.7
Deimos	Mars	30.0
Phobos	Mars	0.3

(a) Schematize this as a ternary. Is this okay?
(b) Schematize this using nesting. Is this okay?
(c) Schematize this using only simple binaries. Is this okay?

5. The following table contains information about movie exports. Suppose we model this with the quaternary: Movie exported from Country to Country achieved SalesLevel. Draw the schema for this approach. Is this acceptable? If not, explain why.

Movie	*Origin*	*Export performance*	
		Country	*Sales level*
Backdraft	US	AU	High
		NZ	Low
		GB	Medium
Crocodile Dundee	AU	NZ	Low
		US	High
Terminator 2	US	?	?

4.6 Projections and Joins

The previous section showed that a *sufficient* condition for splitting an unnested fact type is that it has a uniqueness constraint spanning fewer than $n - 1$ roles. Since this is not a *necessary* condition for splitting, in some cases further analysis is required to make a definite decision. At the heart of such an analysis is the question: *can the fact type be rephrased as a conjunction of smaller fact types?*

With a bit of experience behind us, we can usually answer this question fairly quickly. This section discusses a formal procedure for addressing this question in a systematic way. The procedure makes use of two operations known as "projection" and "join", which are important in relational database work. The *join* operation has already been discussed, and involves combining tables by matching values referencing the same object to form a new table (review Section 4.4).

The **projection** operation also produces a table, but is performed on a single table. To project on one or more columns of a table, we *choose just the columns of interest* (removing all other columns) and then *ensure each row in the result appears just once* (removing any duplicates).

Notationally, the columns on which a projection is made are listed in *italic square brackets* after the table name, separated by commas. For example, *T [a, c]* is a projection on columns *a* and *c* of table *T*. Academics often write this instead as $\pi_{a,c}(T)$, where "π" (pi) is the Greek "p" (first letter of "projection"). We prefer the square bracket notation because it encourages a top-down view of the operation (first find the table, then choose the columns). Visually we can picture the operation as returning a vertical subset of the whole table (see Figure 4.54).

For example, a ternary fact type met earlier is reproduced in Figure 4.55(a). Its fact table T1 has six rows. Projecting on the Person and Degree roles, we obtain the fact type Person seeks Degree, with its fact table T2 as shown in Figure 4.55(b). Notice that T2 has only three rows, because projection eliminates duplicates. Projecting on the Person and Subject roles yields the fact type Person studies Subject, and its fact table T3. Since T3 is a projection on a key of T1, it must have the same number of rows as T1 (Why?).

Figure 4.54 The projection operation.

Since the roles on the ternary are played by different object types, by default the roles are named using lowercase versions of the object type names. Using these role names as attribute names, the fact tables may be set out in relational terms as T1 (person, degree, subject), T2 (person, degree), and T3 (person, subject). The two projections may now be expressed thus: T2 = T1*[*person, degree*]* and T3 = T1*[*person, subject*]*.

In an earlier section, the ternary Person seeking Degree studies Subject was reworded as the conjunction Person seeks Degree **and** studies Subject. Here, the logical "**and**" operator performs a conceptual join on Person of the two binary fact types: Person seeks Degree; Person studies Subject. Since the ternary is equivalent to the conjunction of the binaries, no information is lost if we transform the ternary into the two binaries or vice versa. So no matter what the fact populations are, the ternary fact table must contain exactly the same elementary facts as the two binary fact tables collectively contain. In relational terms, the ternary relation is equal to the (natural inner) join of the two binary projections. So, T1 = T2 ⋈ T3. Check this out for yourself using the populations shown in Figure 4.55.

Conceptually, the situation is summarized by the following equivalence:

Person seeking Degree studies Subject **iff** Person seeks Degree **and** studies Subject.

Here "iff" abbreviates "if and only if". The projection corresponds to the conditional: *if* the ternary is true *then* the conjunction is true. The join corresponds to the conditional: *if* the conjunction is true *then* the ternary is true. Any compound fact can always be formulated as a logical conjunction (using "and").

In principle (although of little use in practice, as discussed later), the following *projection–join check* can be applied to test whether a fact type is compound (and hence splittable).

Figure 4.55 Compound fact types can be split by projection and restored by join.

Provide a significant fact table for the fact type.
Split this table into two or more projections.
Recombine by natural (inner) join.
The fact type is splittable in this way **iff** the result is the same as the original.

An inner join is acceptable since the conceptual fact table (and hence its projections) cannot have nulls. We chose a significant population for the ternary example just discussed, so the test confirms that ternary's splittability.

Now consider our familiar ternary example about people scoring ratings for subjects. A significant fact table is provided for it in Figure 4.56(a). Suppose we suspect that this ternary is splittable on Person. For example, we might feel that in this UoD the following equivalence holds:

Person scored Rating for Subject **iff** Person scored Rating
and studied Subject.

To test this way to split, we could form the two binary projections shown in Figure 4.56(b), and then recombine by joining on Person. The result is shown in Figure 4.56(c). As an exercise, please confirm this result. In forming the join, several new rows (marked in bold italics) appeared that were not in the original table. Any one of these new rows is enough to prove that the fact type cannot be split this way (i.e., on Person).

Consider for instance the fact from the first row of our original table: The person with surname "Adams" scored a rating of 7 for the subject CS112. We attempted to split this fact into the two separate facts: Adams scored a rating of 7; Adams studied CS112 (see first rows of the two projections). If this splitting is legitimate, the two separate facts must, in combination, be equivalent to the original fact. But they are not. Adams can score more than one rating and study more than one subject. Knowing that Adams scored a 7 does not tell us the subject(s) for which Adams scored this rating. It could be any of Adams's subjects (CS112, CS110, or PD102).

Joining the projections causes each rating of Adams to be paired with each subject of Adams. Some of these rows were not present in the original, and the join does not tell us which ones apply to the original ternary. Since we are unable to construct the population of the original ternary from the new ternary population, information has been lost. Hence the new ternary fact type is not equivalent to the original fact type. So the original ternary fact type cannot be split in this way.

This projection–join check invalidated the proposed equivalence: Person scored Rating for Subject **iff** Person scored Rating **and** studied Subject. The forward conditional of the equivalence does hold (*if* ternary *then* conjunction). However, the backward conditional fails (it is not generally true that *if* conjunction *then* ternary). Simply understanding the English verbalization of the fact types should be enough for us to understand that "Person scored Rating for Subject" differs in meaning from "Person scored Rating and studied Subject", if we read "and" in a purely conjunctive sense. Another way to spot the difference is that the uniqueness constraint on the join fact type in Figure 4.56(c) is weaker than that of the original in Figure 4.56(a).

Figure 4.56 The original ternary cannot be split in this way.

If you use some insight it should be clear that trying to split on Subject, Rating, or all three would be pointless. In this case you may conclude that the fact type is unsplittable and leave it as a ternary. If this is not clear then you can use the algorithm to test all possible ways of splitting. You would find that none of them work.

In general, for a ternary A-B-C there are four ways in which it might split: A-B, A-C; B-A, B-C; C-A, C-B; and A-B, A-C, B-C. In the fourth case the schema diagram is triangular in shape: this is referred to as three-way splitting. As an exercise, draw these four possibilities. With three-way splitting the join should be done in two stages. For example, first join A-B and A-C (with A common); then join the A-B-C result of this join with B-C (with B and C common).

A classic example of testing for 3-way splitting involves the fact type Agent sells Car-type for Company. Even if the key spans the whole fact type, the fact type can be split into three binaries if a derivation rule allows the ternary to be deduced from the binaries. Such examples are very rare in practice. For an *n*-ary fact type, if the *n*-way split fails (by generating new rows in the final join) then all other ways of splitting can be ruled out too: the fact type is unsplittable.

Although useful for understanding the notion of splittability, the projection–join test is of little practical value for actually determining splittability. The problem is that it only works if the population is significant. But to know the population is significant implies that we know what all the constraints are, and have crafted a population to illustrate these constraints. If we already know what the constraints are, it is obvious whether the fact type is elementary, so there's no need to do any projection–join test.

In Figure 4.55(a) for example, if we already know that each person seeks at most one degree then we need to separate out the underlying binary. In Figure 4.56(a) there was no similar constraint, because a person can score many ratings. Formally this is the only difference between the two cases, but it's enough for us to determine splittability without resorting to the tedious projection–join test. The best test for splittability is to work linguistically with the domain expert to see if a fact can be phrased as a conjunction, and to test this with a couple of carefully chosen instances. We'll have more to say on the notion of semantic dependencies in Section 14.6.

In case you feel like you've just wasted a lot of time learning a projection–join check that is of theoretical interest only, please don't be upset. It's very useful to understand what the projection and join operations are, at both conceptual and logical levels. Apart from their modeling impact, these operations are used extensively in both conceptual and relational query languages, as we'll discuss later.

Exercise 4.6

1. A template for a ternary fact type is outlined here. The population shown is significant.

a1	b1	c1
a1	b1	c2
a2	b2	c1
a2	b2	c2

(a) Add the uniqueness constraints for this fact type.
(b) Use the projection–join test to show that this fact type cannot be split into two binaries with *C* as the common node.
(c) Use the projection–join test to show that this can be split on *B* into two binaries
(d) Draw a schema outline for this result.

4.7 Summary

In step 4 of the conceptual schema design procedure, we *add uniqueness constraints and check the arity of fact types*. Uniqueness constraints (UCs) come in two varieties: internal and external. An *internal* (or intrapredicate) UC applies to one or more roles of the same predicate. It is marked as a bar across the constrained role(s). If the roles are noncontiguous, the bar is divided, with a dotted line connecting its active parts.

Redundancy is repetition of an elementary fact. Since redundancy is not permitted in a conceptual fact table, each row must be unique. This is shown as a UC across the whole predicate, unless a stronger UC exists. If an internal UC spans just some of the roles spanned by another internal UC, the former UC implies the latter UC, and the implied UC should not be displayed.

Each unary predicate has a UC across its role, whether or not it is displayed. Binary predicates have four possibilities: UC on role 1 only; UC on role 2 only; UC on role 1 and another UC on role 2; and one UC spanning both roles. These cases are, respectively, described as many to one (*n*:1) , one to many (1:*n*), one to one (1:1), and many to many (*m*:*n*). For example, if each employee works for at most one department, but many employees may work for the same department, the predicate "works for" is many to one and has a UC on its first role. In Figure 4.57, the *R* predicates are *n*:1 and the S predicate is *m*:*n*.

Each role in a predicate is associated with a column of the predicate-table. A simple UC across just one role means no duplicates are allowed in its column. A UC spanning *n* roles means each sequence of *n* entries taken from the associated columns can occur in that column sequence on only one row.

Figure 4.57 Some uniqueness constraints.

Let *aRb* and *bSc* be two fact instances. The conjunction of these facts (*aRb* **and** *bSc*) is a compound fact instance (*aRb* **that** *Sc*) of the schema path *A-R-B-S-C* that is the conceptual (inner) *join* of *R* and *S* via the common object type *B*. The fact table for this join is obtained by pairing rows of *R* with rows of *S* where the *B* values match, and showing the *B* column(s) just once in the result. In relational terms this is a natural inner join, symbolized by "⋈". For example, $R \bowtie S$ in Figure 4.57(b) joins on *A* and has a table population of {(a1, b1, c1), (a1, b1, c2), (a2, b2, c2)}.

An *external* (or interpredicate) uniqueness constraint is shown by connecting two or more roles from different predicates to a circled bar "⊖". This indicates that when the join operation is applied to the predicates, an internal UC spans these roles in the result. As an example of the constraints shown in Figure 4.57(b), a program may appear on at most one channel, and may appear at different time slots, but for any given channel and time slot at most one program is shown.

Uniqueness constraints apply to the model, and must be at least as strong as those in the real world. A combination of roles spanned by an internal UC, with no smaller UC inside it, is a *key* of its predicate. A one-role key is a *simple key*.

Because conceptual predicates are elementary, a ternary fact type cannot have a simple key. It either has a three-role UC, or one or more two-role UCs. For a predicate with *n* roles, *each internal UC must span at least n – 1 roles*. This *n – 1 rule* may be applied as a *key-length check*. Violation of this rule is a sufficient but not necessary condition for splittability.

This rule also implies the following *spanning rule: a uniqueness constraint on an objectified association must normally span all its roles*. Exceptions to this rule are allowed for 1:1 cases and other cases discussed in Chapter 10.

For a given predicate, let *X* denote a combination of one or more roles (or columns) and *Y* denote a single role (or column). We say *X functionally determines Y*, written $X \rightarrow Y$, if and only if for each value of *X* there is at most one value of *Y*. In this case, *Y* is said to be *functionally dependent* on *X*. The term "*FD*" abbreviates "functional dependency". If the conceptual schema is correct, all FDs corresponding to actual constraints are implied by UCs. If a non-implied FD $X \rightarrow Y$ exists, the predicate should be split on the source *X*.

With experience, you can usually determine whether a fact type splits by using background knowledge of the UoD to answer the questions: "Can the fact type be rephrased as a conjunction? Is information lost by splitting the fact type?" The shortest key rule and the detection of a non-implied FD provide simple ways of checking that our intuitions are correct here.

The *projection* on columns *a*, *c*, … of table *T*, written *T[a, c, …]*, is obtained by choosing just those columns and removing any duplicate rows from the result. If we know that we have a significant population for a fact type, the *projection–join check* can be used to determine its elementarity. If we suspect that a predicate might split in a certain way, we can split its significant table in this way by projection and then recombine by natural join: if new instances appear in the result then the fact type cannot be split in this way. This test is of more theoretical than practical interest.

Chapter Notes

By depicting each role as a box associated with a table column, Object-Role Modeling enables uniqueness constraints to be portrayed in a natural way where the impact of the constraint on populations is immediately apparent. Entity Relationship modeling (ER) and UML depict internal uniqueness constraints on associations using a variety of notations for "cardinality constraints" or "multiplicity constraints", as discussed later. ER offers graphical support for some uniqueness constraints on attributes, but UML does not. Neither ER nor UML offers any significant support for external uniqueness constraints.

Object-Role Modeling makes minimal use of the notion of functional dependency, but this notion is central to many other methods. For a clear treatment of an ER approach as well as functional dependency theory, see Elmasri and Navathe (1994).

Although it is generally wise to objectify a predicate only if it has a spanning uniqueness constraint, rare cases may arise where it may be useful to remove this restriction, as discussed in Chapter 10.

This page intentionally left blank

5 Mandatory Roles

5.1 Introduction to CSDP Step 5
- CSDP step 5 overview
- Populations of roles, fact types, and object types

5.2 Mandatory and Optional Roles
- Optional and mandatory roles
- Inclusive-or constraints

5.3 Reference schemes
- Simple reference schemes
- Compound reference schemes
- Disjunctive reference

5.4 Case study
- Compact disc retailer

5.5 Logical derivation check
- Logical derivation rules
- Semiderived fact types

5.6 Summary

5.1 Introduction to CSDP Step 5

So far you've learned how to proceed from familiar information examples to a conceptual schema diagram in which the elementary fact types are clearly set out, with the relevant uniqueness constraints marked on each. You also learned how to perform some checks on the quality of the schemas. In practice, other kinds of constraints and checks need to be considered also.

Next in importance to uniqueness constraints are *mandatory role constraints*. Basically these indicate which roles *must* be played by the population of an object type and which are optional. Once mandatory role constraints are specified, a check is made to see if some fact types may be *logically derived* from others. This constitutes the next step in the design procedure.

CSDP step 5: Add mandatory role constraints, and check for logical derivations

The next two sections cover this step in detail. The rest of this section discusses some basic concepts used in our treatment of mandatory roles and later constraints. Once mandatory roles are understood, we are in a good position to examine reference schemes in depth, especially composite reference—we do this later in the chapter.

Recall that a *type* may be equated with the set of all its possible instances. This is true for both object types and relationship types. For a given schema, types are fixed or unchanging. For a *given state* of the database and a given type *T*, we define *pop(T)*, the **population** of *T*, to be *the set of all instances of T in that state*.

Let us use "{ }" to denote the *null set*, or *empty set* (i.e., the set with no members). When the database is empty, the population of each of its types = { }. As the database is updated, the population of a given type may change. For example, suppose the ternary in Figure 5.1 is used to store information about medals won by countries in the next Olympic Games. The fact type and roles are numbered for easy reference.

Initially the fact table for F1 is empty, since no sporting results are known, so pop(F1) = { }. Now suppose the database is to be updated after each sporting event, and in the first event the gold and bronze medals are won by the United States and the silver medal by Japan. The new state of the fact table is shown in Figure 5.2, using 'G' , 'S', and 'B' for gold, silver, and bronze. Now the population of the fact type contains three facts. The population grows each time the results of an event are entered.

Figure 5.1 A fact type for Olympic Games results.

Figure 5.2 The fact type populated with results of the first event.

We may also define the *population of a role*. Each role of a fact type is associated with a column in its fact table. Values entered in the column refer to instances of the object type that plays that role. Given any role *r* and any state of the database:

pop(r) = population of role *r*
= set of objects referenced in the column for *r*

Typically the objects referenced are entities, not values. For example, in Figure 5.2, pop(r1) = {The Country with code 'US', The Country with code 'JP'}. The *valuation* of a role *r*, written *val(r)*, is the set of values entered in its column. For example, in Figure 5.2, val(r1) = {'US', 'JP'}.

When there is no chance of confusion, object terms may be abbreviated to constants in listing populations. With this understanding, in Figure 5.2, pop(r1) = {US, JP}, pop(r2) = {G, S, B} and pop(r3) = {1}. Role populations are used to determine *object type populations*. With our current example, if each country referenced in the database must play r1, then after the first event pop(Country) changes from { } to {US, JP}, and pop(CountryCode) changes from { } to {'US', 'JP'}. Assuming the UoD is just about the Olympics, the entity type Country is the set of all countries that might possibly compete in the Olympic Games.

A predicate occurs within an elementary fact type or reference type (also known as an existential fact type). Reference types are usually abbreviated as a parenthesized reference mode. Roles in a reference type are called *referential roles*. Roles in an elementary fact type are called "*fact roles*". Each entity type in a completed conceptual schema plays at least one referential role and, unless declared independent (see section 6.3), at least one fact role. In general, the *population of an entity type equals the union of the population of its roles*. Unless the entity type is independent, its population is the union of the populations of its fact roles.

For example, the populated schema in Figure 5.3 indicates the size (in square kilometers) and carbon dioxide emissions (as a percentage of worldwide emissions) of some countries as measured in 2004. The Country entity *type* might include many countries (e.g. from Afghanistan to Zimbabwe), but for this state of the database, only three countries are listed in each fact type. Each instance in the database population of Country plays the role r1, r2, or both. So the current *population* of Country is the set of all the instances referenced in either the r1 column or the r2 column.

Figure 5.3 pop(Country) = pop(r1) ∪ pop(r2).

We could set this out as:

pop(Country) = pop(r1) ∪ pop(r2)
= {AU, FR, US} ∪ {AU, CN, US}
= {AU, CN, FR, US }

Here "∪" is the operator for set union. The union of two sets is the set of all the elements in either or both. For instance {1, 2} ∪ {2, 3, 4} = {1, 2, 3, 4}. In the case just given, Australia and the United States occur in both role populations, while China and France occur in only one.

5.2 Mandatory and Optional Roles

Consider the output report of Table 5.1. The question mark "?" denotes a *null*, indicating that an actual value is not recorded. For instance, patient 002 may have a phone but this information is not recorded, or he/she may simply have no phone.

Patients are identified by a patient number. Different patients may have the same name and even the same phone number, so the population of Table 5.1 is not significant. We must record each patient's name, but it is optional whether we record a phone number. Figure 5.4 shows a preliminary conceptual model for this situation. Conceptual facts are elementary, so they cannot contain nulls. The null in Table 5.1 is catered for by the absence of a fact for patient 002 in the fact type Patient has PhoneNr.

In step 5 of the design procedure, each role is classified as mandatory or optional. A **role is mandatory** if and only if, for all states of the database, the role must be played by every member of the population of its object type; otherwise the role is **optional**. A mandatory role is also called a *total role*, since it is played by the total population of its object type.

Table 5.1 Details about hospital patients.

Patient Nr	*Patient Name*	*Phone*
001	Adams C	2057642
002	Brown S	?
003	Collins T	8853020

Figure 5.4 A populated schema for Table 5.1.

Which of the four roles in Figure 5.4 are mandatory and which are optional? If the diagram includes all the fact types for the UoD, and its sample population is significant, we can easily determine which roles are mandatory.

In practice, however, we mostly work with subschemas rather than the complete, or global schema; and sample populations are rarely significant. In such cases, we should check with the domain expert whether the relevant information must be recorded for all instances of the object type (e.g., must we record the name of each patient?).

Consider the two fact roles played by Patient. For the database state shown, the populations of these roles are {001, 003} and {001, 002, 003}. If these are the only fact roles of Patient, then pop(Patient) = {001, 002, 003}. The role of having a name is played by all recorded patients, and the role of having a phone number is played by only some. Assuming the population is significant in this regard, the name role is mandatory and the phone number role is optional.

To indicate explicitly that a role is mandatory, we add a **mandatory role dot** to the line that connects the role to its object type. This dot may be placed at either end of the role line. In Figure 5.5(a), the dot is placed at the object type end. This reinforces the *global* nature of the constraint in applying to the object type's population. If we add a patient instance to the population of a mandatory Patient role, we must also include this instance in the population of each other role declared mandatory for Patient.

In this sense, a mandatory role constraint can have an impact beyond its predicate. In contrast, a uniqueness constraint is *local* in nature, constraining just the population of its role(s), with no impact on other predicates.

Figure 5.5 Mandatory role constraint shown as a dot at either end of the role line.

In Figure 5.5(b), the mandatory dot is placed at the role end. This choice is useful when an object type's role lines are so close together that if we added a dot at the object type end it would be unclear which role is intended. This situation can arise when an object type plays a large number of roles displayed on the same schema page.

In Figure 5.5 the first role of the top predicate is both mandatory and unique. The mandatory role constraint verbalizes as **Each** Patient has **at least one** PatientName, or equivalently, as **Each** Patient has **some** PatientName. The uniqueness constraint verbalizes as **Each** Patient has **at most one** PatientName. In combination, these constraints verbalize as **Each** Patient has **exactly one** PatientName (i.e., each recorded patient has one and only one name recorded). In general, *at least one + at most one = exactly one*.

In a completed schema, *if two or more fact roles are played by the same object type, then individually these roles are optional unless marked mandatory*. For example, in Figure 5.5 the first role of Patient has PhoneNr is optional. This means it is possible to add a patient to the database population without adding a phone number for that patient.

What about the roles played by PatientName and PhoneNr? If these are the only roles played by these object types in the global schema, then these roles are mandatory. Unless declared independent (see Section 6.3), each primitive object must play some role, and each primitive entity must play some fact role. Hence *by default, if a primitive object type plays only one role, or a primitive entity type plays only one fact role* (in the global schema), *this role is mandatory*. In such cases, the implied mandatory role dot is usually omitted. In Figure 5.5, if no other roles are played by PatientName and PhoneNr, their roles are mandatory by implication. In this case, although not recommended, these implied constraints could be marked explicitly as shown in Figure 5.6.

Explicit depiction of implied mandatory role constraints has several disadvantages. First, it complicates schema evolution. For example, suppose that tomorrow we decide to add the fact types: Patient had previous- PatientName; Patient has secondary- PhoneNr. With the implicit approach, this is simply an addition to the current schema. But with the explicit approach, we need to delete the formerly implied mandatory role constraints and replace them with weaker constraints (e.g., Each PatientName is of, or was of, at least one Patient; Each PhoneNr is used by, or is a secondary number for, at least one Patient—such inclusive-or constraints are discussed shortly).

Another problem with marking implied mandatory constraints is that this de-emphasizes the mandatory role constraints that are really important, that is, the ones we need to enforce (e.g., you must record a name for each patient). If you think about it, you should see that implied mandatory constraints are enforced automatically and hence have no implementation impact.

Figure 5.6 Implied mandatory constraints specified explicitly (not recommended).

If we demanded the explicit approach for simple mandatory constraints, we should do the same for disjunctive mandatory (inclusive-or) constraints. But if the roles involved in the disjunction occur on separate schema pages, there is no convenient way to mark the constraint between them.

Another problem with the explicit approach is that it complicates theorem specification. Later on we consider some schema equivalence theorems. In this context, the implicit approach allows us to discuss schema fragments independently of other roles played in the global schema, but the explicit approach would forbid this, leading to extra complexity.

For such reasons, the explicit specification should be avoided unless we have some special reason for drawing the attention of a human reader to the implied constraints. As discussed in the next chapter, the rule for implicit mandatory roles does not apply to subtypes: if they play just one fact role, this is optional unless marked mandatory.

If a role is mandatory, its population always equals the total population of its attached object type. Figure 5.7 depicts the general case for an object type *A* and an attached role *r*. The role *r* may occur in any position in a predicate of any arity, and *A* may participate in other predicates as well. Mandatory role constraints are enforced on populations rather than types. If the only fact stored in the database is that patient 001 is named "Adams C", the name role played by Patient is still mandatory even though many more instances of the Patient type have yet to be added to the population.

Recall that the UoD (typically part of the real world) is not the same thing as the recorded world or formal model of the UoD. Like other constraints, mandatory role constraints are assertions about our model, and do not necessarily apply to the world being modeled. With our current example, it is optional whether a patient has a phone. This simply means that we do not need to record a phone number for each patient. Maybe this is because in the real world not every patient has a phone or perhaps each patient does have a phone but some patients won't supply their phone number.

The real world schema of Figure 5.8(a) concerns applicants for an academic position. By nature, each person has exactly one gender. As a business decision, each applicant is required to have a degree. Applicants may or may not have a fax number. Figure 5.8(b) shows the model actually used. Here, applicants may choose whether to have their gender recorded, but must provide details about their degrees. For this model, a business decision has been made (perhaps to avoid gender bias) to remove a mandatory constraint that applies in the real world. This fairly common practice of relaxing some real world mandatory constraint should always be a conscious decision.

Role *r* is **mandatory** (for the population of *A*).

Each instance of type *A* that is recorded in the database is also recorded to play *r*.

pop(*r*) = pop(*A*)

Figure 5.7 The role *r* is mandatory.

Figure 5.8 A conscious decision to make recording of an Applicant's gender optional.

Do *not* read a mandatory role constraint to mean "if an object plays that role in real life then we must record it". The information system can work only with the model we give it of the world—it cannot enforce real world constraints not expressed in this model.

The following checking procedure helps ensure that our mandatory role constraints are correct. For each mandatory role: is it mandatory in the real world? If not, make it optional. For each optional role: is it optional in the real world? If not, what reasons are there for making it optional? We refine this further when we discuss subtypes.

In Figure 5.8(b) the mandatory constraint is on a *nonfunctional role* (i.e., a role not covered by a simple uniqueness constraint). In this case it is acceptable, since we wouldn't want to hire any academic without a degree, and we would be interested in all their degrees.

In general, however, you should *be wary of adding mandatory constraints to nonfunctional roles*. Do so only if you are sure the constraint is needed. Such cases can lead to complexities (e.g., referential cycles) in the implementation, which are best avoided if possible. Of course if your application really requires such a constraint then you should declare it, regardless of its implementation impact.

Novice modelers tend to be too heavy handed with mandatory role constraints, automatically making a role mandatory if it's mandatory in the real world. In practice, however, it is often best to make some of these roles optional in the model to allow for cases where for some reason we can't obtain all the information. As a general piece of advice, *make a role mandatory if and only if you need to*.

When an object type plays more than one role, special care is needed in updating the database to take account of mandatory roles. Suppose we want to add some facts from Table 5.1 into a database constrained by the conceptual schema of Figure 5.5 (recall that it is mandatory to record a patient's name but optional to record a phone number).

Assuming the database is initially empty, an untutored user might proceed as follows.

User		*CIP*
add: Patient 001 has PatientName ‘Adams C’.	→	accepted.
add: Patient 001 has PhoneNr ‘2057642’	→	accepted.
add: Patient 003 has PhoneNr ‘8853020’	→	rejected. Violates constraint: **Each** Patient has **some** PatientName.

To add the third fact into the database we must either first record the fact that patient 003 has the name ‘Collins T’ or include this with the phone fact in a compound transaction.

Now consider the two report extracts shown in Figure 5.9. These list sample details maintained by a sporting club. Membership of this club is restricted to players and coaches. For simplicity, assume members may be identified by name. The term “D.O.B.” means date of birth. On joining the club, each person is assigned to a team, in the capacity of player or coach (possibly both).

Teams are identified by semantic codes (e.g., ‘MR-A’, ‘BS-B’, ‘WS-A’ denote the men’s rugby A team, the boy’s soccer B team, and the women’s soccer A team), but the semantics of these codes is left implicit rather than being stored explicitly in the information system. A record is kept of the total number of points scored by each team. Each team is initially assigned zero points, even if it doesn’t yet have any members.

As an exercise, you might like to model this yourself before peeking at the solution in Figure 5.10. The uniqueness constraints assert that each member coaches at most one team, plays for at most one team, was born on at most one date, joined the club on at most one date, and each team has at most one coach and scored at most one total. The reference mode “mdy” for Date indicates that for verbalization purposes date instances are in month–day–year format—of course, this conceptual choice does not exclude other formats being specified for internal or external schemas.

Member	*D.O.B.*	*Joined*
Adams F	02-15-1975	01-02-1999
Anderson A	12-25-1986	01-02-1999
Brown C	11-02-1960	11-14-1992
Collins T	02-15-1946	05-05-1980
Crystal B	01-02-1986	11-14-1999
Downes S	11-02-1960	06-17-1995
...	...	...

Team	*Pts*	*Coach*	*Players*
MR-A	23	Downes S	Adams F Brown C Collins T ...
BS-B	0	Collins T	Anderson A Crystal B ...
WS-A	0	?	?
...	...	...	...

Figure 5.9 Extracts of two reports from a sporting club.

Figure 5.10 Inclusive-or constraint: each member coaches or plays (or both).

The mandatory role dot on Team indicates that each team has a total score (possibly 0). The other mandatory role dots on Member indicate that for each member we must record their birth date and the date they joined the club. The circled mandatory role dot is linked to two roles: this is an **inclusive-or** (**disjunctive mandatory role**) **constraint**, indicating that the disjunction of these two roles is mandatory for Member. That is, each member *either* coaches *or* plays (*or both*).

For example, Adams is a player only, Downes is a coach only, and Collins is both a player and a coach. A coach of one team may even play for the same team (although the population doesn't illustrate this). The lack of an inclusive-or constraint over the "is coached by" and "includes" roles of Team indicates that it is possible to know about a team without knowing its coach or any of its players.

If Figure 5.10 includes all the roles played by Date in the global schema, then the disjunction of Date's roles is also mandatory (each date is a birth date or a join date). This could be shown explicitly by linking these roles to an inclusive-or dot; however this constraint is implicitly understood, and the explicit constraint would need to be changed if we added another fact type for Date, so it is better to leave the figure as is.

This Date example illustrates the following generalization of the rule mentioned earlier for single mandatory roles: *by default, the disjunction of roles played by a primitive object type is mandatory* (this default can be over-ridden by declaring the object type independent—see next chapter). If the object type is an *entity type, the disjunction of its fact-roles is also mandatory by default*.

Apart from highlighting the important cases where disjunctive mandatory roles need to be enforced (e.g., the coach–player disjunction), and simplifying schema evolution and theorem specification, this rule facilitates the drawing of schemas. For example, object types such as Date and MoneyAmount often have several roles that are disjunctively mandatory, and to display this constraint explicitly would be messy (or even impossible if the object type occurs on separate schema pages).

To avoid confusion, a mandatory role constraint (simple or disjunctive) should be shown *explicitly* if the object type it constrains is a subtype (see Chapter 6).

In the case of implied (possibly disjunctive) mandatory role constraints, it is okay to *mark the constraint explicitly if it will still apply if more roles are added to the object type* (where these extra roles are excluded from the constraint). For object types such as Date, this would almost never happen.

An inclusive-or (disjunctive mandatory role) constraint indicates that each instance in the population of the object type must play at least one of the constrained roles (and possibly all of them).

Inclusive-or (Disjunctive mandatory role) constraint

The inclusive disjunction of roles r_1 .. r_n is mandatory for *A*

i.e., each member of pop(*A*) plays r_1 **or** ... r_n (or all)

i.e., each member of pop(*A*) plays *at least one of* r_1, ..., r_n

Figure 5.11 Inclusive-or constraint (the role disjunction is mandatory).

Figure 5.11 indicates in general how to specify explicitly that a disjunction of roles r_1, r_2, ..., r_n is mandatory by linking the *n* roles to a circled dot. The roles may occur at any position in a predicate, and the predicates need not be distinct.

The inclusive "or" is used in the *verbalization* of the constraint. For example, the inclusive-or constraint in Figure 5.10 is formally verbalized as **Each** Member coaches **some** Team **or** plays for **some** team. As in logic and computing, "or" is always interpreted in the inclusive sense unless we say otherwise.

Inclusive-or constraints sometimes apply to roles in the same predicate. Figure 5.12 relates to a small UoD where people are identified by their first name. A sample population is shown for the ring binary; here each person plays one (or both) of the two roles. For instance, Terry is a child of Alice and Bernie, and a parent of Selena. Since Person plays another role in the schema, the disjunctive mandatory role constraint must be depicted explicitly.

As an example involving derivation, consider Table 5.2. Here students sit a test and exam, and their total score is computed by adding these two scores. The schema for this, shown in Figure 5.13, includes a derived fact type. If included on the diagram, a derived fact type must be marked with an asterisk, and its constraints should be shown explicitly (even though these constraints are typically derived). The mandatory constraint on the derived fact type means the total score must be known—this does not mean that it must be stored.

is a parent of / is a child of

Alice	Terry
Bernie	Terry
Norma	Paul
Norma	Selena
Terry	Paul
Terry	Selena

Figure 5.12 Each person is a parent or child.

Table 5.2 Student scores.

Student Nr	*Student Name*	*Test*	*Exam*	*Total*
1001	Adams, A	15	60	75
1002	Brown, C	10	65	75
1003	Einstein, A	20	80	100

The roles played by Score are named "testScore", "examScore", and "totalScore" to enable the derivation rule to be specified in attribute style. The uniqueness constraint on the total score fact type is derivable from the derivation rule and the other uniqueness constraints (Each student has only one test score and only one exam score and the total score is the sum of these). The mandatory role on the total score fact type is derivable from the derivation rule and the other mandatory roles (Each student has a test score and an exam score, and the rule then provides the total score).

By default, all constraints shown on a derived fact type are derivable. This default rule is discussed in more detail later. In rare cases we may want to declare a derived fact type with a nonderivable constraint. We consider some examples much later.

In principle, a derived fact type may be drawn with no constraints if these can be derived. Typically, however, a derived fact type is included on the diagram for discussion purposes, and it is illuminating to show the constraints explicitly.

The term "mandatory role" is used in the sense of "must be *known*" (either by storing it or deriving it). If desired, we may talk about the population of derived fact types as well as asserted fact types. For a given object type A, the population of A, pop(A) includes all members of A that play any of its roles (asserted, derived, or semiderived). Derived roles may be optional. For example, if either the test or the exam role in Figure 5.13 is optional, the total score role is also optional (although it is still unique).

With large schemas, an object type may play so many roles that it becomes awkward to connect a single shape for the object type to all its roles. To solve this problem, *object types may be duplicated* on a schema. In this case, the rule for implicit mandatory disjunctive roles applies to the union of all the duplicate shapes for the object type. Large schemas are typically divided into pages, and the same object type might appear on many pages and might also be duplicated on the same page.

Figure 5.13 Here, constraints on the derived fact type are derivable.

Figure 5.14 Types may be duplicated, or imported from an external model.

An object type may also be *imported from an external model* where it is originally defined. Different ORM tools may use different notations for such cases.

To indicate that an object type or predicate is duplicated in the same model, either on the same or a different page, the NORMA tool adds a *shadow* to the shape. In Figure 5.14 the object type Date is duplicated on the same page of the model. The fact type Member has MemberName is duplicated on another page (not shown here).

At the time of writing, the NORMA tool does not yet allow an object type to be declared *external* (imported from another model), but this is planned for a later release. The tentative notation to depict an external object type is a circumflex "^" (pointing outside the model). Assuming that this notation is adopted, the Medication object type in Figure 5.14 is an external object type. Since it is defined in another model, its reference scheme may be suppressed.

Exercise 5.2

1. Schematize the UoD described by the following sample output report. At the time these figures were recorded, the former Commonwealth of Independent States (CIS) was treated as a single country. Include uniqueness and mandatory role constraints.

Country	*Reserves (Gt)*	
	Coal	*Oil*
CIS	233	8.6
USA	223	4.1
China	99	2.7
Germany	82	?
Australia	59	?
Saudi Arabia	0.1	23.0

2. Set out the conceptual schema for the following output report. Include uniqueness and mandatory role constraints. Identify any derived fact type(s).

Subject	*Year*	*Enrollment*	*Rating*	*NrStudents*	*%*	*Lecturer*
CS121	2007	200	7	5	2.50	P.L. Cook
			6	10	5.00	
			5	75	37.50	
			4	80	40.00	
			3	10	5.00	
			2	5	2.50	
CS123	2007	150	7	4	2.67	R.V. Green
			6	8	5.33	
			5	60	40.00	
			4	70	46.67	
			1	6	4.00	
CS121	2008	250	7	10	4.00	A.B. White
			6	30	12.00	
			5	100	40.00	
			4	80	32.00	
			3	15	6.00	

3. A cricket fan maintains a record of boundaries scored by Australia, India and New Zealand in their competition matches. In the game of cricket a *six* is scored if the ball is hit over the field boundary on the full. If the ball reaches the boundary after landing on the ground a *four* is scored. In either case, a *boundary* is said to have been scored.

   Although it is possible to score a 4 or 6 by running between the wickets, such cases do not count as boundaries and are not included in the database. A sample output report from this information system is shown. Here "4s" means "number of fours" and "6s" means "number of sixes" Schematize this UoD, including uniqueness constraints, mandatory roles and derived fact types. Use nesting.

Year	*Australia*			*India*			*New Zealand*		
	4s	*6s*	*total*	*4s*	*6s*	*total*	*4s*	*6s*	*total*
1984	120	30	150	135	23	158	115	35	150
1985	112	33	145	110	30	140	120	25	145
1986	140	29	169	135	30	165	123	35	158

4. Report extracts are shown from an information system used for the 1990 Australian federal election for the House of Representatives (the main body of political government in Australia). The table lists codes and titles of political parties that fielded candidates in this election or the previous election (some parties competed in only one of these elections). For this exercise, *treat Independent as a party*.

PartyCode	*Title*
ALP	Australian Labor Party
AD	Australian Democrats
GRN	Greens
GRY	Grey Power
IND	Independent
LIB	Liberal Party of Australia
NDP	Nuclear Disarmament Party
NP	National Party of Australia
…	…

A snapshot of voting details is shown about two seats (i.e., voting regions) in the 1990 election. This snapshot was taken during the later stage of the vote counting. The number of votes for each politician, as well as the informal vote, is initially set to 0. During the election the voting figures are updated continually. The percentage of votes counted is calculated assuming all those on roll actually vote. For simplicity, *assume that each politician and each seat has a unique name*. Figures are maintained for all seats in all states.

An asterisk (*) preceding a politician's name indicates a sitting member (e.g., Fife is the sitting member for the seat of Hume): express this as a binary. Sitting members are recorded only if they seek re-election. Some seats may be new (these have no results for the previous election). States are identified by codes (e.g., 'QLD' denotes Queensland). Each state has many seats (not shown here). Draw a conceptual schema diagram for this UoD. Include uniqueness constraints and mandatory roles. If a fact type is derived, omit it from the diagram but include a derivation rule for it (you may specify this rule informally). Do not attempt any nesting.

QLD:

FADDEN (On roll: 69110)	
CROSS (AD)	7555
FRECKLETON (NP)	2001
JULL (LIB)	24817
HEYMANN (IND)	641
WILKINSON (ALP)	21368
Informal	1404
% counted: 84	
Previous election: AD 4065; ALP 24481; LIB 21258; NP 9581	

NSW:

HUME (On roll: 70093)	
JONES (IND)	9007
* FIFE (LIB)	29553
KIRKWOOD (IND)	507
MARTIN (ALP)	20554
ROBERTS (AD)	3889
Informal	1309
% counted: 92	
Previous election: AD 2498; ALP 24516; IND (total): 2086; LIB 33687; NDP 1105	

5.3 Reference Schemes

With uniqueness and mandatory role constraints covered, it is time to consider reference schemes in more depth. With simple identification schemes, each entity is identified by associating it with a single value. For example, a country might be referenced using a code. For instance, Australia may be referenced by the definite description "The Country that has CountryCode 'AU'". We usually abbreviate simple reference schemes by parenthesizing the reference mode, or manner in which the value relates to the entity, for example, Country(.code). In practice, more complex reference schemes are often encountered, and we need to deal with them even as early as step 1 of the CSDP, when we verbalize facts.

Before examining these more difficult cases, let's have a closer look at simple reference schemes. Table 5.3 contains data about a small UoD where people can be identified by their surnames and each city has a unique name.

Table 5.3 Personal details.

Person	*City*	*Height (cm)*	*Chest (cm)*	*Mass (kg)*	*IQ*
Adams	Brisbane	175	100	77	100
Brisbane	Sydney	182	91	80	120
Collins	Sydney	173	80	67	100
Darwin	Darwin	175	95	90	90

Figure 5.15 shows one way to schematize this. The reference schemes are abbreviated as reference modes in parentheses. Some role names have been added for discussion purposes.

Figure 5.16 repeats the schema, this time with the reference schemes depicted explicitly—each appears as a *mandatory, 1:1 reference type*. For example, in this schema each person has exactly one surname, and each surname refers to at most one person. Technically, this is called an *injection* (or *1:1 into* mapping) from Person to Surname.

Figure 5.15 A conceptual schema for Table 5.3 (reference schemes abbreviated).

Figure 5.16 A conceptual schema for Table 5.3 (reference schemes explicated).

A *double uniqueness bar* on a role indicates the *preferred reference scheme* (also known as the *preferred identifier* or *primary identifier*) for the entity type playing the adjacent role. For example, in this UoD, surnames provide the preferred way to identify persons, and centimeter values provide the preferred way to reference lengths.

In this case, there is only one way to reference instances of a given entity type. In general however, ORM allows many reference schemes for the same entity type; in such cases, one of these eventually needs to be chosen as the preferred reference scheme.

The second generation ORM considered in this book appends a colon ":" when displaying unit-based reference modes (e.g., cm:) and requires a unit dimension (e.g., Length) for each unit. ORM tool display of unit dimensions may be toggled on/off at will. Since access to unit dimensions is always possible with unit-based reference modes, specific object types corresponding to roles played by the unit dimension may be used instead of the underlying unit dimension when setting out fact types. For example, our current example may be remodeled as shown in Figure 5.17.

Unlike top-level entity types, value types may overlap with one another. For example, the populations of Surname and CityName overlap here since both include the character string 'Brisbane'.

Figure 5.17 An alternative conceptual schema for Table 5.3.

Assuming the data in Table 5.3, and an appropriate query language, consider the following dialog between a user and the CIP.

User: Which person lives in a city with the same name as his/her surname?
CIP: Darwin.
User: Which person is a city?
CIP: None: Entity types Person and City are mutually exclusive.

It is meaningful to compare the names for Person and City, since they are both character strings. But it is not sensible to compare persons and cities—they are completely different types of entity. Although the value types for IQ, mass, and length are all numeric, mass and length have different units, while IQ is dimensionless (i.e., it has no units). Conceptually, two numerically referenced entities may be compared or added only if they have the same unit (or both have no unit). Hence the following queries do not make sense:

List persons whose mass equals their IQ.
List persons whose height is more than twice their mass.
List height + mass for each person.

Of course, the numbers themselves may always be compared. Also, height and chest size are both based on the same domain and unit, so they are compatible. So the following queries are acceptable.

List persons whose mass number equals their IQ number.
List persons whose height number is more than twice their mass number.
List height + chest size for each person.

Most current database systems provide little or no support for the distinction between entities and their values. Operations specified in queries are interpreted as applying directly to the values. For example, assuming the data of Table 5.3 is stored in a relational database as the table *Athlete* (person, city, height, chest, mass, iq) the following SQL query is quite legal:

```
select person from Athlete
where person = city
   and mass = iq and chest + mass > height
```

For the population in Table 5.3, this query returns 'Darwin', despite the fact that taken literally the query is nonsense. The comparisons evaluated in the query are actually between values, not entities. We could clarify this by a better choice of column names (e.g., use "surname" and "cityname" instead of "person" and "city").

Values are constants with a predefined interpretation, and hence require no explicit reference scheme. For example, 'Brisbane' denotes itself (i.e., the character string 'Brisbane'), and when written without quotes the numeral '77' denotes the number 77 (Hindu-Arabic decimal notation is assumed). It is sometimes claimed that when written without quotes, a constant such as 'Brisbane' is enough to identify a single entity. However, some context is always required (at least implicitly). Even in Table 5.3, 'Brisbane' is used to refer to both a person and a city.

Table 5.4 Details about computer files.

File		*Size*
Folder	*Filename*	*(kB)*
MyDocs	flag.vsd	35
MyDocs	orm1.doc	1324
OzWork	flag.vsd	40

Earlier in the book we saw that entities are referenced by means of definite descriptions (e.g. "The City with name 'Brisbane'"). Within the context of a reference type, a value may, somewhat loosely, be said to be an identifier for an entity if the value relates to only one entity of that type.

If only one value is used in a definite description, we have a simple 1:1 reference scheme. Sometimes however, two or more values may be required. In this case, we have a *compound reference scheme*. As an example, consider how computer files are identified in Table 5.4. Within a given folder, a file may be identified by its local filename, e.g., "flag.vsd". But when many folders are involved, we need to combine the folder name with the local filename to know which file we are talking about.

To reflect the two-part naming convention suggested by the two columns for File in Table 5.4, we could model the situation as shown in Figure 5.18. The mandatory roles and internal uniqueness constraints declare that each file is in exactly one folder, has exactly one (local) filename, and has exactly one size. The external uniqueness constraint can be used to provide a compound reference scheme and may be verbalized as shown.

If no other identification scheme exists for File, the external uniqueness constraint is the basis for the preferred way in which humans communicate about the files. To indicate this choice of *compound, preferred reference*, a *circled double uniqueness bar* is used, as shown in Figure 5.19.

Using icons for entities, the reference types File is in Folder and File has FileName are populated as shown. In the fact table for File has Size, the file entries appear as value pairs. Although not apparent in the diagram, these are ordered (Folder, FileName) in the same order as the reference types were added to the schema—an ORM tool can display this order on request.

Figure 5.18 File has a compound reference scheme.

Figure 5.19 The previous schema populated with references and elementary facts.

If the reference predicates are included in the definite descriptions for the files, any order can be used. For example, "the File that is in Folder 'MyDocs' and has FileName 'flag.vsd'" is equivalent to "the File that has FileName 'flag.vsd' and is in Folder 'MyDocs'". In general, compound reference schemes involve a mandatory 1:1 map of objects to a tuple of two or more values.

Textually, a compound reference scheme for an object type *A* may be declared by listing the reference types, with the leading *A* removed, in parentheses after *A*. The relevant uniqueness and mandatory constraints on reference predicates are assumed, and if the reference schemes are declared first, the facts may be stated in shortened form. For example, the sample model may be specified textually as follows:

Reference schemes:	Folder (.name); File (is in Folder, has FileName); Size (kB:)
Fact type:	File has Size
Constraints:	**Each** File has **at most one** Size.
	Each File has **at least one** Size.
Fact instances:	File ('MyDocs', 'flag.vsd') has Size 35.
	File ('MyDocs', 'orm1.doc') has Size 1324.
	File ('OzWork', 'flag.vsd') has Size 40.

In principle, we could model Files using a simple reference scheme by concatenating the folder name and local filename into one string with a separator (e.g. "\"). For example, the files may then be named simply "MyDocs\flag.vsd". "MyDocs\orm1.doc" and "OzWork\flag.vsd". This approach is normally undesirable because it requires extra derivation work to extract and group the components to produce various reports. We'll say more about this issue later. Indeed, if we wish to issue queries about the file extension (e.g., "vsd", "doc") as well, a three-part reference scheme is desirable (using folder name, simple name, and file extension).

Table 5.5 Physics department offerings.

Subject	*Semester*
Electronics	1
Mechanics	1
Optics	2

Identification schemes are *relative* to the particular UoD. Often, a simple scheme that works in a local context fails in a UoD of wider scope. This notion is important to bear in mind when merging schemas into a larger schema. For example, suppose that within a given university department each subject offered by that department can be identified by its title. For instance, consider the output report in Table 5.5, which indicates which subjects are offered by the Physics Department in which semesters.

If designing a schema for just the Physics Department, we have a 1:1 correspondence between subjects and their titles. So this table may be schematized as the fact type Subject (.title) is offered in Semester (.nr). A similar schematization could be used for just the Mathematics Department, a sample output report for which is shown in Table 5.6.

Table 5.6 Mathematics department offerings.

Subject	*Semester*
Algebra	2
Calculus	1
Mechanics	1

But now suppose we need to integrate departmental schemas into an overall schema for the whole university. Our simple identification scheme for subjects will no longer work because, in this wider UoD, different subjects may have the same title. For example, the subject Mechanics offered by the Physics Department is different from the subject Mechanics offered by the Mathematics Department. A combined output report would look like Table 5.7.

Table 5.7 University subject offerings.

Department	*Subject title*	*Semester*
Physics	Electronics	1
	Mechanics	1
	Optics	2
Mathematics	Algebra	2
	Calculus	1
	Mechanics	1
…	…	…

Figure 5.20 A schema for Table 5.7 using a compound reference scheme.

The first row may be verbalized using a compound reference scheme as follows: "The Subject that is offered by the Department named 'Physics' and has the SubjectTitle 'Electronics' is offered in the Semester numbered 1". This approach is schematized in Figure 5.20.

Although conceptually this picture is illuminating, in practice its implementation may be somewhat awkward, with two labels (one for the department and one for the title) needed to identify a subject. In such cases, a new identification scheme is often introduced to provide a simple 1:1 reference. For example, each subject may be assigned a unique subject code, as shown in Table 5.8.

Table 5.8 Subject codes provide a simple reference scheme.

Subject	*Title*	*Department*	*Semester*
PH101	Electronics	Physics	1
PH102	Mechanics	Physics	1
PH200	Optics	Physics	2
MP104	Algebra	Mathematics	2
MP210	Calculus	Mathematics	1
MA109	Mechanics	Mathematics	1

In this output report there are two *candidate identifiers* for Subject. We could identify a subject by its code (e.g., 'PH102') or by combining its department and title (e.g., ('Physics', 'Mechanics')). One of these is chosen as the *preferred identifier*, or standard means of referring to the entity. In this case, we would usually pick the subject code as the preferred identifier. We can indicate this choice by displaying the subject code identification scheme as reference mode in parentheses and displaying the external uniqueness constraint with a single uniqueness bar (see Figure 5.21).

Figure 5.21 A schema for Table 5.8 using subject code as Subject's primary identifier.

If a parenthesized reference mode is displayed, this is always taken to be the preferred reference scheme. Here, the Department and SubjectTitle predicates are treated just like any other elementary fact types; they are no longer considered to be reference types. Of course, the external uniqueness constraint between them must still be declared.

Choosing a preferred reference scheme from two or more existing schemes is perhaps not always a conceptual issue. At the conceptual level, its main uses are to note any business decisions on preferred identification schemes and to disambiguate abbreviated references that omit the reference predicate(s), e.g., "Patient 3452 has Temperature 37". This is especially useful when populating fact tables with instances. At the logical level however, the choice of preferred reference typically has a major impact (e.g., determining the primary key of a table).

Here are some *guidelines for selecting a preferred identifier.* First, *minimize the number of components*. A single subject code is easier to enter than two values for department and title. Moreover, a compound identifier adds extra overhead in the later database implementation (joins, indexes, and integrity checks over composite columns require more effort).

Second, favor an identifier that is *more stable*. In a university environment, the same subject may undergo some title changes throughout its history and sometimes the department that offers a given subject changes. It is even possible for the same department–title combination to refer to different subjects at different times. However, it is extremely rare for a subject to have its code changed, and the same subject code is typically never reused for a different subject.

If an historical record is needed for an object, and its identifier changes though time, extra work is required to keep track of these changes so that we know when we are talking about the same object. This extra effort can be minimized by making the identifier as stable as possible. Ideally, the object has the same identifier throughout the lifetime of the application; this is known as a *rigid identifier*. Most organizations choose rigid identifiers such as EmployeeNr, ClientNr, SerialNr for their employees, clients, equipment, and so on.

To explain some basic concepts in a friendly way, we've used identifiers such as PersonName, Surname, or even Firstname in some examples. However, except for trivial applications, this kind of identification is unrealistic. Often, two people may have the same name. Moreover, people may change their name—women usually change their surname when they marry, and anyone may legally change their name for other reasons. A philosophy lecturer we know once changed his surname to "What" and his wife changed her surname to "Who"!

As another example, the schema in Figure 5.22 expands the UoD considered in Figure 5.19 to enable computer files to be identified across many computers. Here, an artificial fileId is introduced as the preferred identifier to provide a simple, rigid identifier that remains stable even when the file name is changed (a fairly common occurrence). In addition, a human friendly path name is provided as a multipart secondary identification scheme via the external uniqueness constraints.

Figure 5.22 Identifying files in a wider environment.

Concatenating the components of this secondary identification scheme along with separators leads to file names such as "www.myServer.com\c:\MyDocs\flag.vsd". In practice, many systems would also assign simple internal identifiers for other elements (e.g., folders and drives).

A third criterion sometimes used for selecting an identifier is that it be easy for users to recognize. This criterion is mainly used to minimize errors when users enter or access artificial identifiers such as subject or stock-item codes. This effectively endows codes with semantics, at least implicitly. For example, the subject code "CS114" is used in an Australian university for a particular subject in informatics. The first two characters "CS" indicate the discipline area (Computer Science), the first digit indicates the level (first), and the last two digits merely distinguish it from other computer science subjects at the same level. Such a code is less likely to be misread than an arbitrary code (e.g., "09714").

Such *"information-bearing" identifiers* are sometimes called "*semantic names*". They should be avoided if the semantic relationships involved are unstable, since the names would then often need changing. In the current example, however, the semantics are fairly stable—the subject discipline is unlikely ever to change, and the subject level would normally be stable too. The linking of the two letters to discipline (e.g., "MP" for pure mathematics and "MA" for applied mathematics) rather than department is better, since disciplines are more stable than departments.

If semantic names are used, we need to decide whether their semantics are to be modeled in the system or simply assumed to be understood by users. The semantics of subject codes may be modeled explicitly by including the following three binaries in the schema: Subject is in Discipline; Subject is at YearLevel; Subject has SerialNr. An external uniqueness constraint spans the roles played here by the three components.

Figure 5.23(a) shows the case where subject code is the preferred identifier. In this case, these three binaries would normally be derived, and appropriate derivation rules specified (e.g., the discipline code may be derived using a substring operator to select the first two characters of the subject code). The external uniqueness constraint is then derivable.

An alternative approach is shown in Figure 5.23(b). Here the three-part identification scheme is used for preferred reference, and the subject code is derived by concatenation.

Figure 5.23 Different ways of exposing the semantics of subject codes.

The major advantage of storing the components separately like this is that it facilitates queries concerning the components. In particular, *if we wish to formulate queries* ***for each*** *instance of a reference component, then that component should be stored.*

The schema in Figure 5.23(b) makes it easy to query properties for each discipline or year level (e.g., "How many subjects are offered for each discipline?"). As discussed in Section 12.10, the conceptual "for-each" construct corresponds to "group-by" in SQL.

The model in Figure 5.23(a) is extremely awkward for such queries, since groups can normally only be formed from stored columns. However, a simple code is more compact and offers much better performance for relational joins than a three-part identification scheme.

Is there a way of having your cake and eating it too? Yes, at least to some extent. In Figure 5.24 the three components are both derived and stored (as shown by the "**"). This allows fast joins on the subject code and grouped queries on the components.

Semantic codes were often used in legacy systems, with no attempt to expose their semantics as separate components. If these legacy systems have to be retained in their current form, a separate system (e.g., a data warehouse) can be constructed to better support queries on the components. In copying the data from the legacy system to the new system, the codes are transformed into separate component fields.

Figure 5.24 The three components are derived and stored.

Table 5.9 Employee details.

EmpNr	*Employee Name*	*Room*
715	Adams A	69-301
139	Cantor G	67-301
503	Hagar TA	69-507
…	…	…

Just how much of the semantics underlying an information-bearing name should be exposed in a conceptual schema depends on what kinds of queries we wish to issue.

For example, consider the output report shown in Table 5.9. Here room numbers have the format *dd-ddd* (where *d* denotes a digit). The first two digits provide the building number for the room's building, the third digit indicates the room's floor (in this UoD no building has more than nine floors), and the last two digits comprise a serial number to distinguish rooms on the same floor.

If we never wish to query the information system about buildings or floors, it's fine to model rooms using the simple reference scheme Room(.nr). Now suppose instead that we also expect the system to output reports such as the extract shown in Table 5.10. Here we *are* interested in buildings, and even want to issue a query that lists the number of rooms *for each* building. In this case we must expose at least the building component of the reference scheme for buildings, as shown in Figure 5.25.

In the new model, the term "RoomNr" indicates a local room number (e.g., "301") that identifies a room within a given building. It corresponds to the last three digits in a campus-wide room number such as "67-301". If we are not interested in querying the system about floors, there's no need to expose the room number semantics any further.

Table 5.10 Building details.

BuildingNr	*Building Name*	*Nr rooms*
…	…	…
67	Priestly	100
68	Chemistry	100
69	Computer Science	150
…	…	…

Figure 5.25 The reference scheme for Room exposes the building component.

Figure 5.26 Two ways of exposing the three-part reference scheme for rooms.

Now suppose that we wish to query the system about floors; for example, "How many rooms are there on floor 3 of building 67?" To facilitate this, we could expand the reference scheme to three components to expose the floor semantics. One way to model this is shown in Figure 5.26(a). This divides the previous room number (e.g., "301") into a FloorNr (e.g., "3") and a serial number (e.g., "01").

If we want to think about floors themselves, rather than just floor numbers, it is better to model Floor as an entity type as shown in Figure 5.26(b) Here Floor plays in the composite reference scheme for Room, and also has its own composite reference scheme.

Although less convenient, it is possible to use derivation functions to formulate queries about single *instances* of a reference component, without actually storing the component. For example, to list the number of rooms on floor 3 of building 67, we could use the following SQL query: **select count**(*) **from** Room **where** roomNr **like** '67%' **and** roomNr **like** '%-3%'. Even though this approach is possible, we still recommend exposing the semantics. If ever in doubt as to whether to expose some semantics, it's generally better to do so.

You should always store a reference component if you wish to query about a *group* of its instances; for example, "*For each* floor of building 67, how many rooms are there?" Using the schema of Figure 5.26(b), this may be formulated in ORM by selecting the path from Room to Building (i.e., Room is on **a** Floor **that** is in Building), adding the condition "= 67" for Building (implicitly applied to BuildingNr), and then requesting Floor and **count**(Room) **for each** Floor.

At the relational level, the query may be formulated in SQL using a group-by clause; for example, **select** floorNr, count(*) **from** Room **where** buildingNr = 67 **group by** floorNr. The SQL language is discussed in detail in Chapters 12 and 13.

Choosing reference schemes for entity types is an important aspect of modeling. As a strict guideline, *an entity type's preferred identification scheme must 1:1 map each entity of that type to a tuple of one or more values* (either directly or indirectly). So, each preferred reference scheme provides an *injection* (mandatory, 1:1 mapping). For example, in Figure 5.26, each building maps directly to a unique building number, and each room maps (indirectly via the reference predicates) to a unique triple of values (building number, floor number, serial number).

Figure 5.27 This association is not 1:1, so should *not* be used for preferred reference.

In everyday life, weaker kinds of identification scheme are used in some contexts. For example, some people have identifying nicknames (e.g. "Tricky Dicky" denotes President Nixon, and "the great white shark" denotes the golfer Greg Norman). However in a typical application not everybody will have a nickname, so nicknames are not used for preferred reference (they are not mandatory).

Sometimes a 1:many reference scheme is used. For example, a plant or animal type may have more than one identifying, common name. For instance, the bird species with the scientific name of *Dacelo gigas* may also be identified by any of the following common names: "kookaburra", "laughing jackass", "great brown kingfisher", or "bushman's clock". As a botanical example, "gorse" and "furze" refer to the same plant.

In Figure 5.27, common names are used as the preferred identifier for bird species. This choice is signified by the double uniqueness constraint bar. A sample population is included to illustrate the 1:many nature of this reference type. Do you spot any problem with this reference scheme?

This association must not be used for preferred reference because it is not 1:1. You can imagine the problems that would arise if we used "kookaburra" to identify a bird species in one part of a database and "laughing jackass" in another part. In the absence of any standard identifier like a scientific name, how would we know that we were talking about the same bird species?

For preferred reference, we must choose a mandatory, 1:1 scheme. Such a scheme may already exist (e.g., scientific name). If not, we create one with the help of the domain expert. This might be partly artificial (e.g., standard common name) or completely artificial (e.g., birdkindNr). Whatever we choose, we need to get the users of the information system to agree upon it.

The scientific name for a bird species is actually a semantic name with two components. For example, in *Dacelo gigas*, the "Dacelo" is the name of the genus, and "gigas" is a species name. By itself, a species name such as "gigas" or "americana" does not identify a species. For instance, *Certhia americana* (the brown creeper) is a different bird species from *Parula americana* (the northern parula).

Figure 5.28 Many common names may be used for the same species.

If we wish to query about the genus or species name, we should unpack the naming semantics into a two-part reference scheme as shown in Figure 5.28; otherwise we can use the simple reference scheme BirdSpecies(.scientificName).

Notice that we can still include facts about common names. Once this schema is fully populated, we could let users interact with the system via the common names, still using the scientific naming convention to establish identity.

Simple 1:1 naming schemes relate entities directly to values (e.g., subjects to subject codes). In life we sometimes identify entities by 1:1 relating them to other entities (e.g., "the Olympics that was held in the Year 1992", "the Warehouse that is located in the City 'Brisbane'", "the Director who heads the Department 'Sales'"). Such identification schemes may be chosen for primary reference, and depicted as mandatory 1:1 binaries (e.g., Olympics was held in Year (CE)). These binaries are then regarded as reference types, not elementary fact types.

However, this practice is rare. Instead such binary associations are usually modeled as elementary fact types, and other primary reference schemes are chosen (e.g., Olympics(.sequenceNr), Warehouse(.nr), Director(.empNr)). Alternatively, the relevant entity types may be removed by using suitably descriptive predicates. For example, the assertion "The olympics of Year 1992 was located in City 'Barcelona'" may be portrayed as a binary association between the entity types Year and City. If the original referential semantics are not needed, another alternative is to use simple value reference, but this is usually a last resort. For example, "Warehouse(.name) 'Brisbane'" loses the semantics that this warehouse is located in the city Brisbane.

In compound reference schemes, each reference predicate is typically mandatory. In such cases, each entity of the same type is identified using the same number of values. In the real world however, we sometimes encounter identification schemes where some of the reference roles are optional, but their disjunction is still mandatory. This is called *disjunctive reference*. In these cases, different entities of the same type may be referenced by different numbers of values. For example, consider the botanical identification scheme depicted in Figure 5.29.

Some kinds of plants are identified simply by a genus (e.g., *Agrostis*). Many other plant varieties are referenced by a combination of genus and species names (e.g., *Acacia interior*). Still others are identified by combining genus, species, and infraspecies, where the infraspecies itself is identified by a rank and infraname (e.g., *Eucalyptus fibrosa* ssp. *nubila*). So depending on the kind of plant, there may be one, two, or four values required to identify it.

Figure 5.29 With botanical identification, some reference roles are optional.

The external uniqueness constraint over three roles indicates that each subtuple of (genus, species, infraspecies) refers to at most one plant kind. The other external uniqueness constraint declares that the (rank, infraname) pair identifies infraspecies. In the reference scheme for Plantkind, only the genus is mandatory. Moreover, for each genus there is at most one plant kind with only a genus name. And for each genus-species combination there is at most one plant kind with no infraspecies. In a relational database, each plant kind maps to a sequence of four values, three of which may be null, and each quadruple is unique (treating null just like any other value). Uniqueness constraints over optional roles are discussed in detail in Section 10.1.

Years ago when Peter Ritson and one of us formalized disjunctive reference within ORM, we displayed it with a percentage sign next to the external uniqueness marker, suggesting that "partial" sequences of components are allowed. Nowadays this notation is no longer used, and the basic external uniqueness symbol is used for both ordinary and disjunctive reference. You know the reference scheme is disjunctive if at least one reference role is optional. The minimum requirement for a legal reference scheme is that it provides a mandatory 1:1 mapping to a sequence of one or more values. So the disjunction of the reference roles must still be mandatory.

You may have noticed the arrow with a subset symbol running from the first role of Plant has Infraspecies to the first role of Plant has SpeciesName. This denotes the subset constraint that infraspecies is recorded only if species is. Subset constraints are discussed in detail in the next chapter.

Believe it or not, the reference scheme of Figure 5.29 is a simplified version of the actual identification scheme used in botany, where complications such as hybrids and cultivars also need to be catered for. Life can be messy!

In practice, *try to avoid using a disjunctive reference scheme for preferred reference*. With the current example, we should consider introducing an alternative simple identifier (e.g., plantkindNr). If the users agree to use this in their normal communication, this change can be made to the conceptual model. If not, we can still consider using it in the logical design. If we do this, we still need to model the disjunctive information as a "secondary reference" scheme. Although still messy to implement, moving the disjunction from a primary to a secondary reference simplifies and improves the performance for most implementations.

Another way to avoid disjunctive reference is to concatenate the reference components into a single name. Since this makes it difficult to issue queries about individual components, this is unlikely to be viable with our current example.

Yet another way to avoid disjunctive reference is to include special default values in the identification scheme. For example, if we use a symbol such as "--" or "#" for "does not exist" all the reference roles become mandatory, and this special value can be treated like any other value in enforcing uniqueness. Although this approach can work for implementing some cases, it is hardly conceptual and may be impractical if you require the user community to actually use the special values or if you need the same symbol to work with different data types (e.g., numbers as well as strings).

Sometimes reference chains can get lengthy. A value is identified by a constant. Each entity is referenced directly using a sequence of one or more objects, which in turn may be values or entities. In the latter case, the reference chain continues, recursively, until finally the referencing objects are all values. In this way, each object is ultimately referenced by one or more values that appear as entries in named columns of output reports or database tables.

Candidate identifiers for the same entity are said to be *synonyms*. In the information systems literature the term "*homonym*" is used for a label that refers to more than one entity. For example, the same surname "Jones" may refer to more than one person. Since "homonym" has different grammatical senses, another term such as "nonidentifying label" is preferable. At any rate, the "problem of homonyms" is solved either by augmenting the reference scheme until identification is achieved (e.g., combining surname and initials) or by using a completely different preferred identification scheme (e.g., social security number).

In cases considered so far, an object's reference scheme has been the same throughout the whole schema. The next chapter examines some cases involving subtypes where this assumption is removed; this leads to *context dependent reference*. To end this section, let's consider the case of variable reference where *different units are used for the same physical quantity* in the same application domain.

In Australia, pieces of lumber have their longest length specified in meters, but their breadth and depth are measured in millimeters. Because this is a standard in the Australian building industry, a report on lumber sizes might look like Table 5.11.

Suppose we want to compare the different linear measurements (e.g., Which sizes of lumber are 20 times as long as their breadth?) or compute volumes in standard units (e.g., What is the volume of lumber size C7 in cubic meters?). Figure 5.30(a) shows one way to model this situation.

Table 5.11 Lumber details.

Lumber size	*Length (m)*	*Breadth (mm)*	*Depth (mm)*
A4	3	100	100
B7	2	100	75
C7	5	200	100

Figure 5.30 Derived units are based on a standard unit.

Here the colon ":" indicates the unit-based nature of the reference modes. The asterisk "*" after "(mm:)" indicates that millimeter (mm) is a *derived unit*, unlike meter (m), which is a *standard unit*. Figure 5.30(b) displays the expanded form of the unit reference modes, revealing that each unit is based on the same *unit dimension* (Length). It also shows the *conversion rule* between the units. A derived unit is always placed on the left of the equals sign. We use the term "dimension" to include derived dimensions (e.g., Volume = Length^3). Both sides of a conversion rule equation must have the same dimension (e.g., Length) or dimension expression (e.g., Mass/Volume).

Since several units might be used for the same physical quantity, we reduce the number of conversion rules by choosing a standard unit and just supplying rules to convert to this standard (rather than separate conversions between each unit pair).

ORM 2 regards object types with the same unit-based reference mode to be compatible. So, if Length and Pressure occur in the same model, it would be incorrect to use the reference schemes Length(mm:) and Pressure(mm:) since this implies that Length and Pressure are based on the same semantic domain. To avoid this problem, we need to rename one of the reference modes, for example, Pressure (mmHg:).

As a more abstract approach, we could model the metadata (length, etc.) in Table 5.11 as data, as shown in the populated model in Figure 5.31. The first fact from Table 5.11 could then be expressed by the facts: LumberSize 'A4' has Attribute 'length' of NumericValue 3; Attribute 'length' has Unit 'm'. The unit conversion rule is still required. This technique is called *metadata demotion* since it demotes metadata (data about data) to domain data. While this approach is very powerful, it is often unnatural for most users. If used at all, it should normally be transformed into a simpler external schema for user interaction.

As a simpler example of treating different units as values, a financial application involving many currencies might model amounts of money using the compound reference scheme MoneyAmount(is in Currency(.code), has NumericValue()).

Demoting metadata (types) to data (instances) can help to minimize changes to the schema over time. An abstract model such as Figure 5.31 might be used to permit new attributes to be added without altering the database schema. For example, a medical application might use the ternary association Patient has Attribute of Value to anticipate new patient tests to be added later. In such cases, additional associations are often needed to control the use of units within their relevant contexts.

Figure 5.31 Demoting metadata to data.

Sometimes, we may need to work with multiple unit systems. For example, we may need both metric and nonmetric measures as users gradually move from one unit system to another, or merge separate applications. In addition to supplying conversion rules to transform between different units, we must take care to expose the units to humans interacting with the system (recall the fate of the Mars Climate Orbiter).

Exercise 5.3

1. It is desired to identify a warehouse by its physical location. Design an appropriate identification scheme for the each of the following contexts:
   (a) UoD restricted to one suburb. The street in which the warehouse is located is identified by its name.
   (b) UoD restricted to one city. Each suburb is identified by its name.
   (c) UoD restricted to one country. Each city is a major city, identified by name.
   (d) UoD restricted to planet Earth. Countries are identified by name.
   (e) UoD restricted to Milky Way galaxy. Planets are identified by name.

2. The UoD is restricted to Earth, and we wish to store facts of the form: Warehouse has Floorspace; Warehouse contains Item in Quantity. Is the identification scheme discussed in Question 1(d) practical from the implementation point of view? If not, suggest a better scheme and support your scheme by comparing a sample population for the two fact types, using both approaches. Comment on the relative work needed to perform a join (conceptual or relational) between the fact types.

3. (a) A triangle ABC is formed from points in a Cartesian coordinate system. The coordinates are pure numbers (no units). Assume that at any given time, only one shape can be displayed, which must be a triangle. Model the information displayed.

(b) The map shown here indicates the approximate location of parking stations near a famous cathedral. The squares on the grid represent city blocks. Model the parking information stored on this map.

4. Members of a small social club are identified by the combination of their given names and surname. Each member has at least one and at most three given names. For example, one member is Eve Jones, another is Johann Sebastian Bach, and another is Eve Mary Elizabeth Jones. It is required that each component of their names be individually accessible. Draw a conceptual schema diagram for this situation.

5. Members of a small American gymnasium have their weight recorded in pounds (lb). The weight that each member can bench press is also recorded, but in kilograms (kg). It is desired to compare these two weights but retain the separate units. Specify a conceptual schema for this situation. Note that 1 lb = 0.454 kg and 1 kg = 2.205 lb.

6. The following table lists the common names by which beer drinks of various volumes may be ordered in hotels in the states of Australia. Volumes are measured in fluid ounces (oz). The sample data is significant. A double hyphen "--" indicates that beer drinks of that volume are not on sale in that state. For instance, in Queensland exactly three different beer drinks may be ordered. Schematize this UoD.

	4 oz	5 oz	6 oz	7 oz	8 oz	10 oz	15 oz	20 oz
Qld	--	Small beer	--	--	Glass	Pot	--	--
NSW	--	Pony	--	Seven	--	Middy	Schooner	Pint
Vic	--	Pony	Small	Glass	--	Pot	Schooner	--
SA	--	Pony	--	Butcher	--	Schooner	Pint	--
WA	Shetland Pony	Pony	--	Glass	--	Middy	Schooner	Pot
Tas	Small beer	--	Beer or Six	--	Eight	Pot or Ten	--	--
NT	--	--	--	Seven	--	Handle	Schooner	--

7. The following report is an extract from an information system that records the results of driving tests for various employees. A person may take as many tests as he/she wishes, but cannot take more than one test on the same day. A minimum score of 80 is required to pass the test. Sometimes a person takes another test simply to try for a better score. Schematize this UoD.

Test	*Date*	*Driver*		*Result*	
		SSN	*Name*	*Score*	*P/F*
101	1/2/00	539-31-1234	Hagar, David Paul	75	F
102	1/2/00	438-12-3456	Jones, Selena Linda	85	P
103	1/9/00	539-31-1234	Hagar, David Paul	80	P

5.4 Case Study: A Compact Disc Retailer

The following small case study is designed to consolidate the main ideas in CSDP steps 1–5. To derive maximum benefit from it, you should attempt to solve the problem yourself before looking at the solution. The business domain concerns a compact disc (CD) retailer who uses an information system to help with account and stock control, and to provide a specialized service to customers seeking information about specific musical compositions and artists.

Figure 5.32 illustrates the kind of musical information required for compact discs (some of the data is fictitious). Each disc contains several individual musical items, referred to as "tracks". Although compact discs usually have about 20 tracks, for this example only a few tracks are listed.

cd#: 654321-2 **name**: Special Oldies

artist: **record company**: EMI

track#	*title*	*duration*	*singers*
1	Maya's Dance	225	Donovan
2	Wonderful Land	200	
3	King of the Trees	340	Cat Stevens
4	Sultans of Swing	240	Mark Knopfler
5	Seven Wonders	300	Stevie Nicks
…	…	…	…

cd#: 792542-2 **name**: The Other Side of the Mirror

artist: Stevie Nicks **record company**: EMI

track#	*title*	*duration*	*singers*
1	Rooms on Fire	300	Stevie Nicks
2	Two Kinds of Love	250	Bruce Hornsby, Stevie Nicks
…	…	…	…

cd#: 836419-2 **name**: Money for Nothing

artist: Dire Straits **record company**: Phonogram

track#	*title*	*duration*	*singers*
1	Sultans of Swing	346	Mark Knopfler
2	Walk of Life	247	Mark Knopfler
…	…	…	…

cd#: 925838-2 **name**: Fleetwood Mac Greatest Hits

artist: Fleetwood Mac **record company**: Warner Bros

track#	*title*	*duration*	*singers*
1	Say You Love Me	200	Chris McVie, Stevie Nicks
2	Seven Wonders	300	Stevie Nicks
…	…	…	…

Figure 5.32 Musical details about four compact discs.

Before reading the next two paragraphs, try to verbalize information from Figure 5.32 as elementary facts. Pay particular attention to the reference schemes involved.

Each compact disc has a CD number as its preferred identifier. Although not shown here, different discs may have the same name. Note that "CD" is used here in a generic sense, like a catalog stock item or car model. The retailer may have many copies of CD 654321-2 in stock, but for our purposes these are all treated as the same CD. In some applications (e.g., car sales) you do need to distinguish between different copies of the same model, but for our application this is not a requirement.

An artist is a person or a group of persons. For a given CD, a main artist is listed if and only if most of the tracks on the disc are by this artist (this constraint is enforced by a data entry operator, not by the system). Assuming that each artist has an identifying name, we use the fact type: CompactDisc(CDNr) has main- Artist(.name). If this were not the case, we must replace Artist by ArtistName thus: CompactDisc(CDNr) has ArtistName(). The record company that releases the disc must be recorded.

A challenging aspect of this application is understanding what is meant by "track" and determining its identification scheme. To clarify terms, you often need to consult the domain expert. The term "track" is commonly used in two senses: the *physical track* on a compact disc and the musical composition, or *work*, recorded on this track.

Within the context of a given CD, tracks are identified by their track number, or sequential position on the disc. But there are many CDs in this domain, so we need both the CD number and the track number to identify a track. Using "track" in the physical sense, the fact involving "Maya's Dance" in the first CD report may be verbalized thus:

> The Track that is on the CompactDisc with CDNr '654321-2' and has TrackNr 1
> stores a work with the Title 'Maya's Dance'.

This used the compound reference scheme: Track(is on CompactDisc, has TrackNr). Notice that the fact predicate says "*a* work" of that title, not "*the* work". Different musical works may have the same title. Although a physical track occurs on only one CD, the same work may appear on many discs. For example, track 5 of CD 654321-2 and track 2 of CD 925838-2 hold the same title, and probably the same work. There are also two tracks titled "Sultans of Swing", but these have different durations so are not the same work. In rare cases, two tracks with the same title can occur on the same CD (not shown here).

The previous verbalization did not include Work as an entity type. Should we include it? Do we need to know whether a work on one disc is exactly the same as a work on another disc? Only the domain expert can tell us whether he/she wants to know this. For our solution, let's assume the answer to this question is "no".

However, if the answer were "yes", we would need a 1:1 identification scheme for Work. A natural reference scheme for Work is hard to find (even the combination of title, artist, and duration might not be enough). We might introduce an artificial identifier (e.g., workNr), but this might be hard to enforce, since humans are responsible for enforcing the constraints in preferred reference schemes.

Table 5.12 Stock quantity and recommended retail price of compact discs.

CD#	*CD name*	*Stock qty*	*RRP*
654321-2	Special Oldies	5	17.95
792542-2	The Other Side of the Mirror	100	25.00
836419-2	Money for Nothing	10	20.00
925838-2	Fleetwood Mac Greatest Hits	50	23.95
…	…	…	…

The duration of a track is the time it takes to play. Each track has exactly one duration, measured in seconds. Most tracks have one or more singers. Some tracks may have no singers (these are instrumental rather than vocal tracks, although this dichotomy is left implicit).

Table 5.12 shows an extract from another report from the same application. This lists the quantity in stock and recommended retail price (in US dollars) of each compact disc. This table is updated when required, on a daily basis. The stock quantity of a disc may drop to zero. For simplicity, let's assume that only the current stock and price figures are recorded (no history).

An extract from a third report required of the information system is shown in Table 5.13. For simplicity, let's assume that sales records are kept only for the current calendar year of operation (otherwise the year needs to be included). For each month that has passed, figures are kept of the quantity of copies sold and the net revenue (profit) accruing from sales of the compact discs in that month.

For simplicity, these figures are restricted to the four compact discs met earlier. In reality this table would have thousands of rows and much larger totals. This table is updated on a monthly basis. Daily sales figures are kept manually but are not part of the application. For each month, if a disc was on the stock list and had no sales for that month then a figure of zero is recorded for it. If a disc is a new stock item in the month then it has no sales figures recorded for previous months (e.g., disc 792542-2 was not in stock in January).

Table 5.13 Monthly sales figures (small sample only)

Month	*CD#*	*CD name*	*Qty sold*	*Net revenue*
Jan	654321-2	Special Oldies	0	0.00
	836419-2	Money for Nothing	30	180.00
	925838-2	Fleetwood Mac Greatest Hits	50	400.00
			80	580.00
Feb	654321-2	Special Oldies	5	33.50
	792542-2	The Other Side of the Mirror	70	630.00
	836419-2	Money for Nothing	15	90.00
	925838-2	Fleetwood Mac Greatest Hits	50	350.00
			140	1103.5 0

This is the last report we need to model. Before reading on, try to verbalize this report and then schematize the whole application. Include all uniqueness and mandatory role constraints, and derivation rules.

How did you fare with the monthly sales report? The reference schemes here are obvious. The totals for quantity sold and net revenue are obviously derived. You may have schematized the stored fact types thus: CompactDisc has CDname(); CompactDisc in Month sold in Quantity; CompactDisc in Month earned Profit. This is acceptable (although as shown later, an equality constraint then needs to be added).

However it is better to nest the common part of the two ternaries, as shown in Figure 5.33. Here the name "Listing" is used for the objectified association that records which compact discs were listed for sale in what months. For each individual listing (of a given compact disc in a given month) the number sold (possibly zero) and profit earned (also possibly zero) are recorded. Apart from a more compact schema, the nested solution leads to a more efficient relational schema (as discussed later).

If desired, the Profit and RetailPrice entity types may be replaced by MoneyAmount (the USD unit is based on the unit dimension Money), and their specific semantics captured in predicate readings (e.g., Listing earned profit of MoneyAmount). The derivation rules are shown in attribute style, using role names. If desired the derived fact types defined by these rules may also be displayed. Other aspects of the schema are straightforward, and are not discussed here.

Figure 5.33 One way of schematizing the application.

Two main points arising from this case study are: care is often required in choosing and identifying object types, and sample populations are not always significant. Having examined the case study, you are invited to try your hand at a larger schema in the following exercise, which covers all aspects of the CSDP discussed so far.

Exercise 5.4

1. The following reports are extracted from an information system used by an Australian tourism company. The first table provides details about tours offered by the company. Each tour must provide some kind of meal (snack or main meal).

Tour	*Tour Description*	*Snack*	*Main Meal*	*Duration (h)*	*Adult Fare (AUD)*	*Child Fare (AUD)*
1	Gold Coast Grand Tour		✓	11	50	30
2	Brisbane City Tour	✓		3	20	10
3	Rainforest Tour	✓	✓	10	40	22
4	Quick Brisbane Tour	✓		1	5	2
…	…	…	…	…	…	…

All tours are conducted on a daily basis. A tour may depart at more than one time. A given tour departing at a given time is called a "tour shift". The next table shows an extract from the tour shift schedule. Details about a new tour, including its duration, may be recorded before shifts are assigned for that tour (e.g., no shifts have been decided yet for the Rainforest Tour). Each tour shift is offered in at least one language. A "?" mark denotes a null.

Tour Shift	*Language(s) supported*	*Tour*	*Departs*	*Returns*	*Driver*	
					Nr	*Name*
1A	English	1	8:00	19:00	4	Smith, John
2A	English	2	8:30	11:30	1	Smith, John
2B	Japanese	2	10:00	13:00	3	Kano, Fumie
2C	English, Japanese	2	14:00	17:00	1	Smith, John
4A	English, German	4	8:00	9:00	3	Kano, Fumie
4B	English, Japanese, Spanish	4	10:00	11:00	?	?
4C	French	4	12:00	13:00	2	Bell, Michelle
…	…	…	…	…	…	…

Details about tour drivers are maintained as shown in the third table. Some drivers (e.g., driver 5) might not yet have been assigned to any tour shifts. A "?" mark denotes a null.

Driver	*Driver Name*	*Driver's Licence*	*Sex*	*Birthdate*	*Language Fluency*
1	Smith, John	17062925	M	1946-02-15	English, Japanese
2	Bell, Michelle	15534110	M	?	French
3	Kano, Fumie	20130456	F	?	English, German, Japanese
4	Smith, John	10033021	M	1965-03-26	English, Japanese, Spanish
5	Jones, Eve	17765310	F	1946-02-15	English, Japanese, Spanish

Some of the tours have a fixed sequence of sites that they visit, as shown in the figure. Here the circled numbers are stop numbers, indicating the order in which the tour bus stops at a site. It is not possible for a tour to visit the same site more than once. Sites are primarily identified by a site code (e.g., 'MW'), but also have a unique name (e.g., 'Movie World'). Although site codes are not displayed in the figure, they are recorded in the information system. A site may have zero or more facilities of interest. In the figure, these facilities are displayed as icons. In the information system, however, each facility is instead identified primarily by a facility code (e.g., 'DN') and also has a unique name (e.g., 'Dining'). The list of facility codes and names may include facilities not yet provided by any tour site (e.g., Horseback Riding). The code, name, and facilities of other potential tour sites may be recorded even if those sites are not yet included on a tour. For example, we might record 'SB' and 'South Bank' for Brisbane's magnificent South Bank).

Specify an ORM conceptual schema for this UoD. Include all relevant fact types. Model the information about the sequence of sites at which a tour stops using a ternary fact type. Include all uniqueness constraints (internal or external). Include all mandatory constraints (simple or disjunctive). If a fact type is derived, provide a derivation rule for it. You may include the derived fact type on the diagram, but this is not required. Make no use of nesting. Ignore any other constraints (e.g., a driver cannot drive more than one tour shift at the same time, and drivers on a tour shift that supports a language must be fluent in that language—you'll learn how to specify such constraints in later chapters). Do not include facility icons in the conceptual model (just include their codes and names). Make no use of subtyping (even if you know how to subtype—subtyping is covered in the next chapter).

2. Provide an alternative ORM schema fragment to store the data about the sequence of sites a tour visits, using *nesting* instead of a ternary. Exclude fact types for the other reports.

5.5 Logical Derivation Check

Step 3 of the design procedure included a check for basic arithmetic derivations—these are usually obvious (e.g., totals). Once uniqueness and mandatory role constraints are specified, we are in a good position to *check for logical derivations*—these can be harder to spot, especially if some important facts were missed at step 1.

As a simple example, consider the report extract shown in Table 5.14. You may recall this table from the previous chapter, where it was correctly modeled as two binaries. We'll now deliberately model it incorrectly to illustrate the logical derivation check. Suppose we verbalize the first row as the two facts: Lecturer 'Halpin' works for Department 'CS'; Lecturer 'Halpin' works in Building '69'. Assuming the table is significant with respect to uniqueness constraints and mandatory roles, this leads to the schema shown in Figure 5.34.

Table 5.14 University data.

Lecturer	*Department*	*Building*
Halpin	CS	69
Okimura	JA	1
Orlowska	CS	69
Wang	CN	1

To begin our logical derivation check we now ask ourselves: *are there any other associations of interest between the object types, especially functional ones?* A binary association is *functional* if at least one of its roles has a simple uniqueness constraint. Each column entry for this role functionally determines the entry for the other role. Hence a role with a simple uniqueness constraint is said to be a *functional role*.

Looking at Figure 5.34, we now notice that the following fact type is also of interest: Department is located in Building. Suppose we interpret Table 5.14 as also giving us this information. For example, the first row of the table also tells us that the computer science department is located in building 69. Is the new fact type functional?

From rows 2 and 4 we see that the same building may house more than one department. However the population suggests that each department is located in only one building. Suppose the domain expert verifies that this is the case. We might now add this fact type to our schema, to obtain Figure 5.35.

Figure 5.34 A first draft schema for Table 5.14. Is there any other association of interest?

Figure 5.35 A second draft schema for Table 5.14. Which fact type is derivable?

The optional roles of Department and Building in Figure 5.35 allow for other possibilities in the global schema (departments without lecturers, or buildings without departments) not covered by Table 5.14. We now ask ourselves the question: *can any fact type be derived from the others?*

Looking at the three binaries in Figure 5.35, you would probably suspect that one can be derived from the other two. Is this suspicion justified? If so, which binary should be derived? Is there a choice? What do you think?

Suppose we decide to make the new fact type about department location derivable, adding the following logical rule to our derivation rule section:

Department is located in Building **iff** Department employs **some** Lecturer
who works in **that** Building

This treats the department location fact type as a projection on Department and Building from the compound fact type that is the schema path from Department through Lecturer to Building. Suppose the domain expert agrees that this rule does apply. Is it now okay to make the department location derived? No! Why not? Recall the following design guideline: *by default, all constraints on a derived fact type are derivable*. Is this true with our choice of derived fact type?

With our example, the uniqueness constraint on the fact type Department is located in Building is not derivable. How can we know this? One way of seeing this is to provide a *counterexample*—in this case, a sample population that satisfies all constraints except the uniqueness constraint that each department is located in at most one building.

One such population is shown in Figure 5.36. If we populated the horizontal binaries as shown, the derivation rule proposed earlier generates a population for the vertical binary that locates the computer science department in two buildings.

If we derive the vertical binary, we must specify the uniqueness on it as an additional nonderived constraint to be enforced separately—that is, when one of the two horizontal binaries is updated we need to check that all lecturers working for the same department work in the same building.

Sometimes a design decision like this is made in order to speed up queries, even though updates become more expensive. However, the default design guideline should usually be followed, since this simplifies updates.

Figure 5.36 The uniqueness constraint on the vertical binary is violated.

Another problem with deriving the department location fact type is that its mandatory role constraint can't be derived unless we add a mandatory constraint on the employs role. But perhaps some departments have no lecturers. For example, if the public relations department does not employ any lecturers, its location cannot be derived. To satisfy our default guideline, the only possible choice of derived fact type is: Lecturer works in Building (see Figure 5.37).

The derived fact type is included on the diagram for discussion purposes, but would usually be omitted, with just the derivation rule being stated. If the domain expert agrees with the derivation rule shown here, it is safe to derive Lecturer works in Building because its constraints are implied by constraints on the asserted fact types. Since each lecturer works for exactly one department, and each department is located in exactly one building, the derivation rule implies that each lecturer works in exactly one building. So both the uniqueness and mandatory role constraints on the derived fact type are implied.

In cases like this we say the uniqueness constraint is *transitively implied*. You are probably familiar with the notion of transitive relations, where if the relation holds between each consecutive pair in a sequence it holds between the first and last member of the sequence (e.g., if $x > y$ and $y > z$ then $x > z$).

Figure 5.37 If the rule applies, the constraints on the derived fact type are derivable.

Similarly, functional determinacy is transitive: *if a chain of binaries has a uniqueness constraint on the first role of each, then the binary from start to end projected from their inner join has an implied uniqueness constraint on its first role*.

In Figure 5.38, the schema is populated with the sample facts from Table 5.14, as well as the fact that the public relations (PR) department is located in building 2. A chain of two functional binaries goes from Lecturer through Department to Building, and the derivation rule defines the *[*lecturer, building*]* projection on the conceptual join of these two binaries.

For exposition purposes, the FDs corresponding to the uniqueness constraints on the base fact types are displayed. In such cases, we may use the uniqueness constraint pattern (or FD chain) to help choose the derived fact type.

To ensure that no lecturers are lost along the chain, we also require that their departments have a location. So each department playing the employs role must also play the role of having a location. This requirement is an example of a *subset constraint* and is depicted in Figure 5.38 as a dotted arrow with a subset symbol from the incoming role to the outgoing role of Department.

This subset constraint is implied by the mandatory constraint on Department. In combination with the base mandatory constraint on Lecturer, this subset constraint implies the mandatory constraint on the derived association. Subset constraints are discussed in detail in the next chapter.

We always need to check with the domain expert whether our derivation rule is semantically correct. The mere existence of such a constraint pattern doesn't guarantee that the other fact type is derived. There are infinitely many associations that could be specified between Lecturer and Building. The other fact type cannot be derived from the work and location fact types unless within the UoD it is logically equivalent to the *[*lecturer, building*]* projection on their join path.

Figure 5.38 The derived association is projected from a functional chain.

So we still need to ensure that the following equivalence really does apply: Lecturer works in Building **iff** Lecturer works for **some** Department **that** is located in **that** Building. Is it possible for Okimura, say, who works in the Japanese department, located in building 1, to work instead in another building (e.g., building 5)? If the answer is "yes", then we have a counterexample to the equivalence, so we cannot derive the association Lecturer works in Building. If no such violation of the derivation rule is possible, then the association is derivable.

To clarify this point, note that the same constraint pattern may occur if we replace the derived fact type in Figure 5.38 by: Lecturer has lunch in Building. In most academic domains, it is common for some lecturers to have their lunch in a building different from their work building. In such a UoD, the lunch fact type is *not* derivable since the following derivation rule does not apply: Lecturer has lunch in Building **iff** Lecturer works for **some** Department **that** is located in **that** Building.

As an aside, the derivation rule in Figure 5.38 could be replaced by a "pair-equality" constraint between the "works in" predicate and the join–path projection formed by connecting the outer roles of Department's predicates. Such "join constraints" are discussed in Chapter 10.

Sometimes, fact types may be derived even if they do not have a mandatory role. For example, suppose some lecturers (e.g., guest lecturers) don't work for a department and don't work in a building. As long as the derivation rule in Figure 5.38 still applies, we may still derive the fact type Lecturer works in Building. The only difference is that the two mandatory role constraints on Lecturer are removed. Of course, in the global schema Lecturer would normally play some other mandatory role, but this is irrelevant to the derivation.

Now consider the case where some lecturers don't work for a department, but all lecturers do work in a building. For example, say Zeno is a visiting lecturer not assigned to any department, but he works in building 70. We might consider handling this case with a *semiderived fact type* (a fact type that is only *partly derived*), weakening the main operator of the derivation rule from "**iff**" to "**if**".

Semiderived fact types have their predicates marked with a "$^{+}$" superscript (intuitively, half a derivation asterisk). This sign is also displayed with the derivation rule, as shown in Figure 5.39, which includes a sample schema population. This approach allows us to enter building facts directly for those lecturers without a department.

Conceptually, semiderived fact types provide a convenient and compact way of dealing with cases where the derivation rule is one way only ("**if**"), rather than an "**iff**" equivalence.

One way to implement semiderived fact types is to replace them with *a disjunction of asserted and fully derived predicates*. For example, to cater for the lecturers without a department, we use the asserted fact type: Lecturer works individually in Building. For the lecturers with a department, we use the derivation rule: Lecturer works departmentally in Building **iff** Lecturer works for **some** Department **that** is located in **that** Building. Now the overall derivation may be specified as: Lecturer works in Building **iff** Lecturer works individually in **that** Building **or** works departmentally in **that** Building.

Figure 5.39 A semiderived fact type (not all lecturers work for a department).

As another example often cited in logic programming, suppose our knowledge of parenthood facts is incomplete, so that we can know a person is grandparent of somebody without knowing who the intermediate parents are. This case can be handled with an asserted parenthood fact type and a semiderived grandparenthood fact type, using the rule: Person$_1$ is a grandparent of Person$_2$ **if** Person$_1$ is a parent of **some** Person$_3$ **who** is a parent of Person$_2$.

Semiderived fact types are also useful in cases where a constraint governs three of more quantities in an equation, and one has a choice as to which quantity will be derived. For example, in a loan application we might want to enter any two of rate, loan term, and loan amount to see what the other figure would be.

Sometimes we encounter derived associations with neither a mandatory nor a functional role. Consider the schema shown in Figure 5.40. This has two optional, many:many associations. Can you see any logical derivation possibilities? This would be easier to deal with if we had an output report or sample data. At any rate, which (if any) of the associations do you consider derivable?

Knowledge about each group member is more precise than knowledge about the group as a whole. So if we are interested in the language expertise of each person we should store this and derive the group expertise from the rule: Group has expertise in Language **iff** Group includes **some** Person **who** is an expert in **that** Language. This approach requires a closed world approach to the expert role played by Person (if a person is an expert in a language, we must record it). Chapter 10 discusses this notion of "relative closure" in more detail.

Figure 5.40 Which (if any) of the *m:n* associations is derivable?

Obviously we cannot derive the expertise of a person from that of his/her group. But what if we are not interested in each person's expertise? In this case the fact type Person is an expert in Language should be deleted, and the remaining binaries would both be asserted fact types.

Exercise 5.5

1. In a computer company, workers are standardly identified by their initials, but also have a unique name. Each worker has access to exactly one personal computer (PC), and each PC is accessed by at least one worker. For each PC a record is kept of its room location, the worker(s) who access it, and the computer language(s), if any, installed on it, as shown.

PC	*Room*	*Workers with access*		*Languages installed*
pc01	507	EFC	(Ed Codfish)	Pascal, Prolog, SQL
		TAH	(Terry Happy)	
pc02	507	NW	(Nancy Wirth)	Pascal, Modula-2
pc03	618	PAB	(Paul Boles)	Hope, Miranda
		JM	(Joan Coffee)	
pc04	508	IN	(Ima Newie)	
pc05	508	PNC	(Peter Crusoe)	C#, SQL
…	…	…	…	…

Each computer language is one of three types (declarative, functional, or procedural) and is installed on a PC *or* has an expert (or both). The PC a worker accesses must be in the room in which he/she works. The next table provides a full record of the languages, their types, who are expert at each, and each expert's room.

Language	*Type*	*Experts (rooms)*
C#	procedural	PNC (508), REK (611)
Hope	functional	?
Java	procedural	JC (618)
Modula-2	procedural	NW (507)
Miranda	functional	PAB (618), DC (708)
Pascal	procedural	NW (507), TAH (507)
Prolog	declarative	JS (407)
SQL	declarative	EFC (507), PNC (508), TAH (507)

A workshop on computer languages is to be delivered by some of the workers. The full workshop program, shown in the final table, indicates how many hours (h) speakers talk about each language, and the total hours for each language type. Schematize this UoD, including uniqueness and mandatory role constraints and derivation.

Declarative (6 h)	*Functional (4 h)*	*Procedural (6 h)*
Prolog: JS (3 h)	Hope: PAB (1 h)	Modula-2: NW (3 h)
SQL: PNC (1 h)	Miranda: PAB (3 h)	Pascal: NW (2 h)
TAH (2 h)		TAH (1 h)

2. The following report refers to a UoD like that discussed in this section, except that departments may be located in many buildings. Each lecturer works in exactly one building. Some departments employ no lecturers. Schematize this UoD using three binaries, and note which if any is derivable. Specify any derivation rules.

Lecturer	*Dept*	*Building*
Halpin	CS	69
Okimura	JA	1
Orlowska	CS	69
Wang	CN	1
Yamamoto	JA	3
?	PR	2

3. Consider a UoD as in Question 2, except that each department employs a lecturer, so the final row of the table in Question 2 is illegal. Schematize this UoD including derivation.

5.6 Summary

For a given state of the database, the *population* of a role *r*, *pop(r)*, is the set of objects referenced as playing that role in that state. $A \cup B$, the *union* of sets *A* and *B*, is the set of all elements in *A*, *B*, or both. The *null set* or empty set, { }, has no members. The population of an entity type *A*, *pop(A)*, is the union of the populations of its roles. Roles of an elementary fact type are called *fact roles*. A relationship type used purely to identify or reference some object is a reference type; its roles are called *reference roles*. For a given state, the population of a fact type (stored or derived) is its set of fact instances.

Step 5 of the conceptual schema design procedure requires us to *add mandatory role constraints and to check for logical derivations*. A role *r* is *mandatory* (or *total*) for an object type *A* **iff**, each member of pop(*A*) is known to play *r* (for each state of the knowledge base); otherwise the role is *optional*. A mandatory role is indicated by a large dot where the role connects to the object type. The dot may instead be placed at the role end.

By default, if a primitive entity type plays only one fact role in the global schema, this role is mandatory—in this case the dot may be omitted since it is implied. Schema constraints apply to the model, not necessarily to the real world. If a role is optional in the real world, it is optional in the model. But a role that is mandatory in the real world may be optional in the model (e.g., the information may be unknown or omitted for privacy).

A *disjunction* of roles $r_1, \ldots, r_n$ is *mandatory* for an object type *A* **iff** each member of pop(*A*) is known to play at least one of these roles (in each state). This *inclusive-or*, or *disjunctive mandatory* constraint is shown by a circled dot ⊙ connected by a dotted line to the roles.

By default, the disjunction of fact roles played by a primitive entity type in the global schema is mandatory—in this case the dot may be omitted since it is implied.

However, a simple or disjunctive mandatory role constraint should be shown explicitly if it applies to a subtype (see next chapter).

To simplify the depiction of an object type with many roles, the object type may be *duplicated* on a schema page as well as over several pages. In this case the rule for implicit mandatory role disjunctions applies to the union of all the duplicates. An *external object type* is imported from another schema in which it is defined. Different CASE tools use different notations (e.g., shading for duplication).

Each entity type must have one or more candidate reference schemes. One of these is picked as the *preferred identification* scheme, and the others are treated as elementary fact types. Preferred reference schemes (also called primary identifiers) should be as simple and stable as possible. A *rigid identifier* identifies the same object throughout the application lifetime. The preferred identification must provide a mandatory 1:1 map of each entity to a tuple of one or more values.

A *simple reference* scheme maps each entity to a single value (e.g., Subject to SubjectCode). A *compound reference* scheme maps each entity types to two or more object types (e.g., Room to Building and RoomNr). Apart from nested cases, the identification aspect of compound reference is denoted by an external uniqueness constraint, depicted as a circled double uniqueness bar "⊜". With *disjunctive reference*, the number of values may vary for different entities of the same type (e.g., Person to Surname and Firstname and optionally Middlename).

Candidate identifiers for the same entity are called *synonyms*. When more than one candidate identification scheme exists, the preferred reference scheme is indicated on the schema as follows: if simple, use a parenthesized reference mode; if compound use "⊜". If a reference mode is expanded to an explicit binary association, the identifying role is marked with a double uniqueness bar.

If *different units* are used for the same quantity, the semantic connection between these units must be modeled (e.g., by declaring the units to be of the same *dimension*). Different units may be modeled as reference modes or by use of a Unit object type. For any given physical quantity, one unit is often picked as the standard unit; conversion rules are supplied to map the derived units to the standard unit.

Derived fact types may be omitted from the diagram, but their derivation rule must be specified. If a derived fact type is included on the diagram, all its constraints should normally be shown. *By default, all constraints shown on a derived fact type are derivable* (from its derivation rule, and other constraints).

Once mandatory role constraints are added, a *logical derivation check* should be performed to see if some fact types are derivable using logical rather than arithmetic operations. First check for missing fact types by asking: *are there any other relationships of interest* between the object types, especially functional relationships? A binary relationship type is *functional* if at least one of its roles is functional (i.e., the role is a simple key). Each column entry for a functional role functionally determines the entry for the other role.

We now ask: *can any fact type be derived* from the others? To help decide this, remember that constraints on a derived fact type should normally be derivable. If we have a chain of two or more functional fact types with uniqueness constraints on all

the first roles, then a functional binary from the start to the end of this chain is derivable if it is defined by projecting on the join of these fact types—its uniqueness constraint is *transitively implied*. In this case, the first role of the derived fact type is mandatory iff the first role of the chain is mandatory and the second role of each binary subsets the first role of the next binary in the chain (e.g., if the first role of each binary in the chain is mandatory).

Derivation rules should normally be biconditionals (i.e., their main operator is "**iff**"). If their main operator is "**if**", the fact type is *semiderived*. A fact type is semiderived iff for some state of the business domain it is possible that some of its instances are derived from other facts and some of its instances are simply asserted. Semiderived fact types are indicated by a plus "$^{+}$"after their predicate reading(s).

Derived fact types do not need to have simple keys. Whether derived or not, a fact type should be excluded from the schema unless it is of interest for the application.

Chapter Notes

Some versions of ORM use a universal quantifier (∀) instead of the mandatory role dot, and place this along the role connector instead of at one end (e.g., DeTroyer et al. 1988). As discussed in later chapters, ER and UML support simple mandatory role constraints for most situations, but not disjunctive mandatory role constraints. Some versions of ER use a solid line for mandatory and a broken one for optional (e.g., Barker 1990), some use double lines for mandatory (e.g. Elmasri and Navathe 1994), and some, including UML, use cardinality or multiplicity markers such as 0 for optional and 1 (or more) for mandatory. UML allows attributes to be declared mandatory or optional, but not all ER notations allow this.

The botanical example used in the discussion of disjunctive reference schemes came from Peter Ritson, who worked with one of us to develop a method for modeling such schemes. For a technical discussion of this topic, see Halpin and Ritson (1992).

This page intentionally left blank

6 Value, Set-Comparison, and Subtype Constraints

6.1 Introduction to CSDP Step 6
- Overview

6.2 Basic Set Theory
- Sets and other collections
- Set comparisons
- Set-forming operations

6.3 Value Constraints and Independent Types
- Object type value constraints
- Role value constraints
- Independent object types

6.4 Subset, Equality, and Exclusion Constraints
- Subset constraints
- Equality constraints
- Exclusion constraints

6.5 Subtyping

6.6 Generalization of Object Types

6.7 Summary

6.1 Introduction to CSDP Step 6

So far you have learned how to verbalize examples in terms of elementary facts, draw the fact types, mark uniqueness constraints and mandatory roles, specify rules for derived fact types, and use simple and complex reference schemes to identify entities. The next step of the conceptual schema design procedure covers three kinds of constraints: value, set comparison, and subtype. Set-comparison constraints are themselves of three kinds: subset, equality, and exclusion.

CSDP step 6: Add value, subset, equality, exclusion, and subtype constraints

This chapter covers step 6 in detail. To clarify the formal concepts underlying the constraints, some basic set theory is first reviewed. Then we consider value constraints (i.e., restrictions on value types). For example, in modeling color monitors we might restrict the values of ColorCode to 'R', 'G', and 'B' (for Red, Green, and Blue). After discussing the related notion of independent objects, we examine the three set-comparison constraints—these declare whether the population of one role sequence must be included in, be equal to, or be mutually exclusive with the population of another.

After that, we examine the notion of subtyping in some depth. Subtyping allows us to declare how types are related (e.g., each manager is an employee) and refines our ability to declare precisely what kinds of objects play what roles. Our treatment of subtyping proceeds from basics through to some reasonably advanced aspects.

6.2 Basic Set Theory

Since the constraints in step 6 make substantial use of sets and set operations, let's first review some basic set theory. To provide a comprehensive summary of the required background, some ideas met earlier are included.

Intuitively, a *set* is a well-defined collection of items. The items may be concrete (e.g., people, computers) or abstract (e.g., numbers, points) and are called *elements* or *members* of the set. Sets themselves are abstract. They are numerically definite in the sense that each has a definite number of elements. A *type* is a set of possible items, understood as sharing some common characteristics (possibly unstated).

The members of a set collectively constitute the *extension* of the set. While a set may *contain* members, it does not *consist* of those members. For example, the set of Martian moons is an abstraction over and above its members (Phobos and Deimos) and consequently has no physical properties such as mass or volume. Although sets (unlike heaps) are not to be equated with their members, they are determined by their members, since two sets are *identical* or **equal** just in case they have the same extension. Using "iff" for "if and only if", the *Law of Extensionality* may be stated thus:

Given any sets A and B, $A = B$ iff A and B have the same members.

Since sets are determined by their members, one simple way of *defining* a set is to *enumerate* or list the elements of the set. In so doing, braces are used as delimiters and commas as item separators (e.g., $A = \{3, 6\}$). Here A is defined as the set containing just the elements 3 and 6.

One consequence of the Law of Extensionality is that a set is unchanged by *repeating* any of its members. For instance, if $A = \{3, 6\}$ and $B = \{3, 6, 6\}$ it follows that $A = B$, since both sets contain precisely the same members (3 and 6). The *cardinality* of a set A, written $\#A$, is the number of different elements in A. For example, $\#\{3, 6\} = \#\{3, 6, 6\} = 2$.

When enumerating sets, it is usual not to repeat the members. However it is sometimes useful to permit this. For example, when stating general results about the set variable $\{x, y\}$ it may be handy to include the case $x = y$. Sometimes repetition of members occurs undetected. For instance, some people do not realize that the entity set {Morning Star, Evening Star} contains just one member (the planet Venus). Of course, the value set {'Morning Star', 'Evening Star'} contains two members.

If repetition is made significant, we have a *bag* or *multiset*. To distinguish between bags and sets, we use different delimiters. We use square brackets for bags, and braces for sets. The cardinality of a bag equals its number of elements, including duplicates. For instance $\#[3, 6] = 2$, but $\#[3, 6, 6] = 3$. One use of bags is in collecting values for statistical work. For example, the set $\{3, 6, 6\}$ has an average of 4.5 but the bag [3, 6, 6] has an average of 5. Bags are frequently used with languages such as SQL.

Another consequence of the Law of Extensionality is that the *order* in which elements are listed is irrelevant. For example, if $A = \{3, 6\}$ and $B = \{6, 3\}$ then $A = B$, since each set contains the same members. Bags are also insensitive to order, as in $[3, 6] = [6, 3]$. If order is made significant, we have an "ordered set".

Usually, when order is made significant, so is repetition. In this case we have a *sequence* (also called a list or permutation or tuple). Thus a sequence is an ordered bag. Sequences are often delimited by parentheses or angle brackets, e.g., $(1, 2)$ or $\langle 1, 2\rangle$. The sequence $(6, 3, 6)$ has three members and differs from the sequence $(3, 6, 6)$. The cardinality of a sequence is its member count, including duplicates. For example, $\#(3, 6, 6) = 3$. In practice, several different delimiting notations are used. For example, "[", "]" are used as set delimiters in Pascal and list delimiters in Prolog.

If a set is enumerated in full, the ordering does not matter. However, a natural ordering often provides an obvious pattern: in such cases a partial enumeration can define the set. For example, the set of decimal digits may be shown as $\{0..9\}$, where the ".." indicates the missing digits. Here 0..9 is often called a *range*. Infinite sets may be represented with ".." at one or both ends. For example, the set of natural numbers may be shown as $\{1, 2, 3..\}$.

The preceding set definitions enumerate, wholly or partially, the extension of the set and are thus examples of an *extensional definition*. A set may also be defined by an *intensional definition*. Here the *intension* or meaning is declared using a property that applies to just the members of the set. That is, a description is given that constitutes both a necessary and a sufficient condition (an "iff condition") for an item to belong to the set.

For example, the set A, defined extensionally as $\{1, 2, 3\}$, may be defined intensionally as: A = the set of natural numbers less than 4. This definition may be recast in set builder notation as $\{x\colon x$ is a natural number less then $4\}$ or more briefly as $\{x\colon x \in \mathbb{N}\ \&\ x < 4\}$, where "$\in$" abbreviates "is a member of" and $\mathbb{N}$ is the set of natural numbers. In set builder notation, a stroke "$|$" may be used instead of a colon, for example, $\{x \mid x < 4\}$.

Some set operations result in propositions, while others result in sets. Note the following *proposition-forming operators*: $=$, $\neq$, $\subseteq$, $\subset$, $\supseteq$, and $\supset$. These are read respectively as "equals", "is not equal to", "is a subset of", "is a proper subset of", "is a superset of", and "is a proper superset of".

Given any sets A and B, we say that A is a **subset** of B iff every member of A is also a member of B. For example, $\{1, 3\} \subseteq \{1, 2, 3\}$. An equivalent definition is: A is a subset of B iff A has no members that are not in B. This second definition makes it easy to see that the *null set* is a subset of every set. The null or empty set has no members and may be represented as $\{\ \}$ or $\varnothing$.

Every set is a subset of itself. For example, $\{1, 3\} \subseteq \{1, 3\}$. We say that A is a *proper subset* of B if and only if A is a subset of B but not equal to B. For instance, $\{1, 3\} \subset \{1, 2, 3\}$. We say that A is a **superset** of B iff B is a subset of A, and that A is a *proper superset* of B iff B is a proper subset of A. For example, $\{1, 2, 3\}$ is both a superset and a proper superset of $\{1, 3\}$.

Comparison relationships between two sets are often depicted by *Euler diagrams*. As developed by the Swiss mathematician Leonhard Euler (1707–1783 CE), these were *spatial* and *existential*. Each set is pictured as a set of points inside an ellipse. This enables the relationship between the sets to be "seen" by the spatial arrangement of the ellipses. For example, placing ellipse A inside B shows that A is a proper subset of B (see Figure 6.1). Here we see that every element of A is also an element of B (so A is a subset of B). The existential viewpoint implies that each of the regions in the Euler diagram contains some elements. So B has some elements not in A. Hence A is a proper subset of B.

To show that A is a subset of B on standard Euler diagrams, we use a disjunction of two diagrams, as in Figure 6.2. The right-hand diagram caters for the case $A = B$.

Figure 6.1 Standard Euler diagram for A is proper subset of B.

Figure 6.2 A disjunction of standard Euler diagrams denotes A is a subset of B.

Figure 6.3 Hypothetical Euler diagrams for set comparisons.

The notion of subsethood is more useful than proper subsethood. Partly to depict such relationships on a single diagram, we use *Hypothetical Euler Diagrams* (HEDs). In a HED, an asterisk is placed in a region to show something exists there, while shading the region indicates that it is empty. If a region is unmarked, the question of whether any elements exist there is left open or hypothetical. Figure 6.3 shows the HEDs for the six most important comparisons between two sets.

Figure 6.3(a) shows *equality* or *identity* (e.g., $A = B = \{1, 2\}$). In Figure 6.3(b), A and B are *disjoint* or *mutually exclusive*; that is they have no members in common (e.g., $A = \{1\}$, $B = \{2\}$). In Figure 6.3(c), A is a *subset* of B (equivalently, B is a superset of A). In Figure 6.3(d), A is a *proper subset* of B (so B is a proper superset of A). For example both $A = \{1\}$, $B = \{1, 2\}$ and $A = \{1\}$, $B = \{1\}$ are instances of subsethood, but only the former is an instance of proper subsethood. Figure 6.3(e) shows *overlap*: the sets have some members in common. Figure 6.3(f) shows *proper overlap*: the sets have common as well as extra members. For example, both $A = \{1, 2\}$, $B = \{2, 3\}$ and $A = \{1\}$, $B = \{1, 2\}$ are instances of overlap but only the former is a case of proper overlap.

Let's now consider *set-forming operations*, where the result is a set. Given any sets A and B, we define $A \cup B$ (i.e., A *union* B) to be the set of all elements in A or B, reading "or" in the inclusive sense. $A \cap B$ (i.e., A *intersect* B) is the set of all elements common to both A and B. Each of these operations is commutative, so the order of the operands doesn't matter. That is, $A \cup B = B \cup A$, and $A \cap B = B \cap A$.

The *set difference* (or relative complement) operation is defined thus: $A - B$ (i.e., A minus B) is the set of all elements that are in A but not in B. This operation does not commute (i.e., cases may arise where $A - B \neq B - A$). If we let U = the *universal set* (i.e., the set of all elements under consideration), we define the *complement* of A as $A' = \mathrm{U} - A$. The *symmetric difference* between A and B is the set of elements in just one of A or B (i.e., the union minus the intersection).

The three most important set-forming operations are union, intersection, and difference. These are depicted in Figure 6.4 by means of *Venn diagrams*, named after their inventor, the English logician John Venn (1834–1923 CE). Here shading indicates the result of the operation.

Figure 6.4 Venn diagrams for three set-forming operations.

Unlike in Euler diagrams, Venn diagram ellipses always overlap. Like HEDs, Venn diagrams adopt the hypothetical viewpoint. As examples of these operations, if $A = \{1, 2\}$ and $B = \{2, 4\}$, then $A \cup B = \{1, 2, 4\}$, $A \cap B = \{2\}$, $A - B = \{1\}$, and $B - A = \{4\}$.

Venn diagrams are sometimes used to discuss comparisons between two sets. For example, Figure 6.5(a) indicates that A ⊂ B, using line fill and "*", respectively, for empty and non-empty regions. However, Venn diagrams become extremely unwieldy as soon as the number of sets exceeds three—consider the Venn diagram for four sets in Figure 6.5(b). Euler diagrams have a similar scalability problem. Section 6.5 introduces a directed graph notation for diagramming subtype connections that enables many compatible object types to be related without incurring the jumble of line crossings that Euler and Venn diagrams would produce for such cases.

Figure 6.5 Venn diagrams for (a) A is a proper subset of B, and (b) four sets.

6.3 Value Constraints and Independent Types

A **value constraint** indicates which values are allowed in a value type or role and is sometimes called a "domain constraint". A value constraint should be declared only if the value list is at least reasonably *stable*; otherwise we would need to continually change the constraint, and hence the schema, as the value list changed.

A value type may be defined by declaring its extension (set of possible values) as one or more enumerations or ranges enclosed in braces (curly brackets). On a schema diagram, an *object type value constraint* is declared by displaying the extension next to either the value type itself, or the entity type that is referenced by the value type.

Figure 6.6 Object type value constraints list the possible values of a value type.

Figure 6.6 shows several examples. If we list or enumerate all the possible values, this is an *enumeration*, as shown in Figure 6.6(a). If the values may be ordered in a continuous list (no gaps) from first to last, we can simply list the first and last with ".." between, since we know how to fill in the intermediate values (see first two examples in Figure 6.6(b)). This is called a *range*.

If a range is *unbounded* at one end, no value appears at that end. For example, if we measure Age as a whole number of years, the value constraint is {0..}. If the values are based on *real* number (e.g., float or decimal), a *square bracket signifies inclusion of the end value* and a *parenthesis signifies exclusion of the end value*. For example, "(0..100]" denotes a range of positive (above 0) real numbers up to and including 100. The last three examples in Figure 6.6(b) all use a real number data type. One may also combine enumerations and/or ranges into a single constraint, as shown in Figure 6.6(c); the first of these examples uses integers while the second uses real numbers.

If the value constraint expression is short, it can be displayed in full on the diagram. If the value list is long, we can enter all the values in an ORM tool but choose to display just some leading values. A trailing ellipsis "…" then indicates that the display of later values is suppressed. For example, for the object type WeekDay we might specify the value list {'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'} but choose to display only the first two (see Figure 6.7). Although the list has been compacted on the diagram, it can be displayed in full textually when desired (e.g., by accessing a properties sheet for the object type)

Figure 6.7 The display of later values is suppressed.

Figure 6.8 Value constraints are declared for MedalKind and a Quantity role.

Now consider Figure 6.8, which is a modified version of an Olympics example discussed earlier, together with a sample population from the final results of the 24th summer Olympics held in Seoul, Korea in 1988 (ordered alphabetically by Country). For this application, only three kinds of medal are allowed: gold, silver, and bronze, denoted by the codes "G", "S" and "B". This constraint is specified by listing the set of possible values {'G', 'S', 'B'} next to the entity type MedalKind.

In the 1988 Olympics there were 237 events, with one gold, silver, and bronze medal to be awarded for each event (assume that there are no ties). It is possible (although highly unlikely) that the same country wins all medals of a given kind. So the role played here by Quantity (we could name this role "medal kind quantity") has a maximum possible value of 237, assuming that we exclude other Games (which could have more events) from the domain.

What is the smallest possible value of medal kind quantity? If we want to record that a country won no medals of a given kind, the lowest medal kind quantity is 0. To reduce the size of the database we could store only facts about actual wins—this approach is suggested by our sample data. In this case, medal kind quantity has a lower limit of 1. Taking the closed world assumption, the fact that a country has won 0 gold medals could then be derived from the absence of an asserted fact about that country winning some gold medals. So this value constraint has the value range {1..237}.

Since Quantity may well play other roles in the global schema, the {1..237} value constraint applies only to Quantity's role in this fact type rather than Quantity itself. We call this a *role value constraint* rather than an object type value constraint. Role value constraints are displayed next to the role to which they apply, as shown for {1..237} in Figure 6.8. An alternative would be to introduce a more specific object type (e.g. MedalKindQuantity) and place the value constraint on that; however such a move may lead to awkward verbalization, so role value constraints are quite handy.

In this example, "Country" denotes the set of all nations on Earth, or perhaps just nations competing in the Olympics. Should we add a value constraint for Country or Country's role? This would be impractical, because the list of country names is too long (e.g., 161 nations competed in the 1988 summer Olympics) and unstable (e.g., in 1991 the former Soviet Union fragmented into 15 nations). If the schema is reused for many games this value constraint would need to be updated continually.

Since the list of countries is both large and unstable, we will not specify a list of values for CountryName as a *constraint* (we'll see shortly another way to supply such lists). However, we can declare a syntactic *data type* as a weak form of "value constraint". For example, we might require each country name to be a string of at most 30 characters.

Although choosing syntactic data types is not the most conceptual or exciting aspect of modeling, it's something you need to do before you implement the model. Otherwise you'll get whatever default data type the system provides, and this won't always be what you want.

Now suppose that our Olympics information system is required to answer the following questions. *Which countries competed? Which countries didn't compete?* The first question could be addressed by adding the unary predicate "competed" to Country in Figure 6.8. The competing countries would then all be entered in the single-column table for this unary fact type. Alternatively, we could explicitly record zero for each competing country not winning a given kind of medal.

But what about the second question? This effectively asks: Which countries exist but didn't compete? This list is usually small, but could be large (e.g., because of a boycott). So somehow we need a way to record all the existing countries, whether or not they competed. As discussed earlier, putting this list of countries in a constraint is unwise because the list is large and unstable. Although schemas often do evolve, we usually try to design them to be as stable as possible. A classic strategy used is to model the feature so that the changes occur in the database population rather than the schema itself. How can we do this for the current example?

You might be tempted to add a unary predicate "exists" to Country. But such a predicate would apply to all object types in the schema. Moreover, there are formal problems with treating "exists" (an existential quantifier) as a predicate. In principle we could address the problem by adding another unary predicate "did not compete" for Country and storing noncompetitors here. But such an approach is awkward—Country might play several roles (birthplace, location, etc.) and it may be arbitrary which role is negated to cater for the rest of the countries.

A cleaner approach is to enter all existing countries in a *reference table* (or object table) for Country. A country may appear in this table without playing any elementary fact roles in the schema. Such a country is said to be *independent*, since it exists in our model but doesn't need to do anything. Recall that a *fact role* is a role in an elementary fact type (a role in an existential fact type is a reference role). We define an **independent object type** to be a *primitive object type whose fact roles (if any) are collectively optional*. The term is not used for subtypes, or for value types that play no fact roles. If an independent object type plays one or more fact roles, the disjunction of these roles is not mandatory (implicitly or explicitly). An independent type may have non-independent instances.

To signify that an object type is independent, an exclamation mark "!" is appended to its name. Figure 6.9 depicts Country as independent. Its reference table lists all existing nations. To save space, only the first six countries are listed—currently there are over 190 countries. The first entry may be verbalized as "The Country with name 'Afghanistan' exists"; this is a reference or existential fact.

Figure 6.9 In this schema, Country is an independent object type.

Figure 6.9 records win counts of zero. So all competing nations play in the ternary. This role is optional, since some nations don't compete (the data here is fictitious). If wins of zero were excluded (as in Figure 6.8) and the unary "competed" were added, the disjunction of this role and the wins role would be optional (and a subset constraint would run from the wins to the competed role—subset constraints are treated later). So the implied mandatory role rule does not apply to independent object types.

As with any entity type, the reference scheme of an independent entity type is mandatory. So no country may be recorded to compete or win unless it is recorded in the population of Country. To fully populate Country with a list of all nations is a tedious task; however, once done it can be imported into many applications as needed. In practice, most independent object types have fewer instances. As a minor point, the value constraint on Quantity's role may need to be updated to allow for more events.

If we did not want to know about noncompeting countries, then as an alternative to adding the unary "competed", we could restrict the meaning of Country to "Competing nation", mark it as independent, and record only nonzero wins.

Objectified associations are often independent, because we often wish to record an association before recording other facts about it. For example, the schema in Figure 6.10 allows us to record the fact that a student has enrolled in a subject (e.g., at the start of a semester) before knowing what result he or she achieves for that enrollment (e.g., at the end of semester).

Figure 6.10 Objectified associations are often independent.

Figure 6.11 Simple object types are rarely independent.

In this example, the simple object type Subject is also independent. In this UoD, it is possible that all we know about a subject is its subject code. It is not necessary that all subjects be taught. However, *simple object types are rarely independent*.

In most academic domains we need to know the title of any subject, not just its code. This more realistic situation is depicted in Figure 6.11(a), where Subject is no longer independent because it has a mandatory fact role. Similarly, in our Olympics example, if all countries had codes as well as names, and we chose code for preferred reference, then `Country has CountryName` becomes a mandatory fact type, and Country is no longer independent (see Figure 6.11(b)).

Use of an independent object type or unary predicates instead of a value constraint adds flexibility since the data can be changed without recompiling the schema (and relevant forms, etc.). However, the responsibility for this feature is now in the hands of the person entering the relevant data rather than the schema designer. Typically write-access to such reference tables is granted only to database administrators.

There is a Dutch version of ORM known as Fully Communication Oriented Information Modeling (FCO-IM) that treats all entity types, independent or not, as objectified predicates. For example, in Figure 6.12 the entity type Country is depicted as the objectification of a role played by the value type CountryName. The notion of an entity type can then be treated as a derived, rather than a base construct.

Figure 6.12 In FCO-IM, an entity type is an objectified predicate.

Exercise 6.3

1. Schematize the following sample report about elementary particles. Include uniqueness, mandatory role and value constraints. Set an upper limit of 2000 amu for mass.

Family	*Particle*	*Charge*	*Mass (amu)*
lepton	neutrino	0	0
	electron	−	1
	positron	+	1
meson	eta	0	1074
baryon	proton	+	1836
	neutron	0	1839

2. It is desired to record a list of all sports, and for each sport, which Olympic Games (if any) included it. Some sports (e.g., running) have been included in each Olympics, some (e.g., judo) only in some, and others (e.g., surfing) never. Schematize this application. You may identify Olympics by its year or by an Olympiad number.

3. An Australian software retailer, SoftMart, maintains an information system to help with invoice and stock control. It has recently opened for business, and has made only a few sales so far. The details of the software items it has in stock are shown. The software items are standardly identified by item codes, but also have unique titles. There are exactly three software categories, identified by codes (SS = spreadsheet, DB = database, WP = word-processsor). The full names of these categories are not recorded. The list price of an item is the normal price at which the item is currently sold. However, SoftMart may sell an item at less than the current list price (e.g., SoftMart may give a discount for bulk orders or to favored clients, and the list price itself may change with time). There is no rule to enable the unit price (i.e., the actual price charged for a copy of an item) to be derived from the current list price.

Itemcode	*Title*	*Category*	*Stock qty*	*Listprice*
B123	Blossom 123	SS	8	799.50
DL	DataLight	DB	10	700.00
DB3	Database 3	DB	5	1999.99
Q	Quinquo	SS	6	400.00
SQL+	SQL plus	DB	4	1890.50
TS	TextStar	WP	5	500.00
WL	WordLight	WP	10	700.00

Customers are identified by a customer number, but the combination of their name and address is also unique. For simplicity, customer name is treated as a single character string, as is address. Customers have at most one phone number recorded. The next table shows customer details.

Customer#	*Name*	*Address*	*Phone*
001	Starcorp	5 Sun St, St Lucia 4067	3765000
002	Eastpac	30 Beach Rd, Sandgate 4017	2691111
003	Dr I.N. Stein	7 Sesame St, St Lucia 4067	?

Customer details may be recorded before the customer places an order. Once an order is placed, the items are issued to the customer together with an invoice. At the time the database snapshot was taken for the reports, only four invoices had been issued. When a customer pays for the items listed in an invoice, the date of payment is recorded. The following table lists the payments so far. Each invoice is identified by its invoice number. For this simple exercise, assume that invoices are paid in full or not at all.

Invoice#	*Date paid*
0501	10/07/2004
0502	20/07/2004
0503	unpaid
0504	unpaid

The four actual invoices are shown. The invoice header giving the address of SoftMart is not stored.

SoftMart, 46 Gallium Street, Brisbane 4001

invoice#: 0501
customer#: 001
date: 03/07/04
customer name: Starcorp
address: 5 Sun St, St Lucia 4067

Item code	*Title*	*Qty ordered*	*Unit price*	*Subtotal*
WL	WordLight	5	650.00	3250.00
Q	Quinquo	1	400.00	400.00
SQL+	SQL plus	5	1701.45	8507.25
B123	Blossom 123	1	799.50	799.50

total amount due: $12956.75

SoftMart, 46 Gallium Street, Brisbane 4001

invoice#: 0502
customer#: 002
date: 03/07/04
customer name: Eastpac
address: 5 Beach Rd, Sandgate 4017

Item code	*Title*	*Qty ordered*	*Unit price*	*Subtotal*
Q	Quinquo	4	400.00	1600.00
TS	TextStar	4	500.00	2000.00

total amount due: $3600.00

SoftMart, 46 Gallium Street, Brisbane 4001

invoice#: 0503
customer#: 001
date: 10/07/04
customer name: Starcorp
address: 5 Sun St, St Lucia 4067

Item code	*Title*	*Qty ordered*	*Unit price*	*Subtotal*
Q	Quinquo	4	350.00	1400.00

total amount due: $1400.00

SoftMart, 46 Gallium Street, Brisbane 4001

invoice#: 0504
customer#: 003
date: 10/07/04
customer name: Dr I.N. Stein
address: 7 Sesame St, St Lucia 4067

Item code	*Title*	*Qty ordered*	*Unit price*	*Subtotal*
B123	Blossom 123	1	799.50	799.50
DL	DataLight	1	700.00	700.00

total amount due: $1499.50

An invoice includes a table of one or more rows, called “line items” or “invoice lines”. Each invoice line lists details about the order of one or more units (copies) of a software item. For simplicity, assume that on a given invoice the same item can appear on only one invoice line. For each invoice line, the item code, title, quantity of units ordered, and unit

price are listed. The total charge for the invoice line is displayed as a subtotal. The total charge for the whole invoice is displayed as the amount due.

Schematize this UoD, including uniqueness, mandatory role and value constraints. Identify a LineItem or invoice line by using the invoice number and the item code. If a fact type is derived, omit it from the diagram but include a derivation rule for it.

4. Consider the previous question, but suppose that the same item may appear on more than one line of the same invoice. Devise an alternative identification scheme to deal with this situation. (*Hint:* Compare this with the compact disc example. Sometimes you need to identify things in terms of their position.)

5. Now suppose that a cumulative record of purchases from suppliers is also to be recorded, an extract of which is shown in the following table. Only a cumulative record is kept (individual deliveries from suppliers are not recorded in the information system). No supplier can be a customer.

Itemcode	*Supplier*	*Quantity*
B123	Macrosoft	7
	TechAtlantic	3
DL	Macrosoft	11
DB3	PacificTech	5
etc.		

   (a) Draw a conceptual schema for this table.
   (b) Assume that this subschema is integrated with your solution to Question 3, and that your fact type for stock quantity is now declared to be derivable.
      (i) Specify a derivation rule for this approach.
      (ii) In a real business, how practical would it be to use such a rule to avoid storing stock quantity? Discuss.

6.4 Subset, Equality, and Exclusion Constraints

Set-comparison constraints restrict the way the population of one role, or role sequence, relates to the population of another. In an earlier section we considered six ways in which two sets might be related: subset, equality, exclusion, proper subset, overlap, and proper overlap. The last three of these require some objects to exist in at least one of the sets being compared. However, static constraints must apply to every state of the database, including the empty state. So the only set-comparison constraints of interest are subset, equality, and exclusion. These three kinds of constraint are examined in this section.

Suppose a fitness club maintains an information system about its members and that Table 6.1 is an extract of a report from this system. Membership includes access to the club's normal fitness equipment (e.g., a weights gym). However, the club also has a few squash and tennis courts. To help ensure fair access to these courts, the club has a policy that members may play only one of these two racquet sports.

Table 6.1 Details about members of a fitness club.

Member	*Sex*	*Birth year*	*Sport*	*Booking*	*Reaction time (ms)*	*Heart rate (beats/min)*
Anderson PE	M	1940	tennis	Mon 5 p.m.	250	80
Bloggs F	M	1940				
Fit IM	F	1975	squash		250	70
Hume D	F	1946	squash	Tue 9 a.m.		
Jones T	M	1965			300	93

Members pay an extra fee for this right and optionally may book one regular weekly hour to use a court. For simplicity, the handling of casual bookings and court allocations is excluded from the UoD. As a service to its members, the club arranges a fitness test to measure the resting heart rate and reaction time of any member who requests it. For simplicity, assume only the latest results are kept for each member.

The determination of fact types, uniqueness constraints, mandatory roles, and value constraints for this example is straightforward (see Figure 6.13). For simplicity, let's agree that hours may be identified by a simple code (e.g., 'Mon. 5 p.m.'), without needing to separate out their day and time components.

However, there are two more constraints that apply to this example. To begin with, only those members who have opted to play a sport may book a court. In terms of the schema diagram, each object that populates the booking role must also appear in the population of the playing role. In other words, the set of members who book an hour must be a subset of the set of members who play a sport. As shown in Figure 6.13, we mark this **subset constraint** by an *arrowed subset symbol* "⊆" running from the subset role to the superset role.

Our sample data agrees with this constraint since {'Anderson PE', 'Hume D'} ⊆ {'Anderson PE', 'Fit IM', 'Hume D'}. If we tried to add a booking for Jones without also adding a sport for him, we would violate this constraint.

In words, the subset constraint verbalizes as **If some** Member booked **some** HourSlot **then that** Member plays **some** Sport. Figure 6.14 summarizes the general case of a subset constraint from one role to another. For this comparison to make sense, both roles must be played by the same object type (or a supertype—see later).

Figure 6.13 A conceptual schema for Table 6.1.

Figure 6.14 A subset constraint between two roles.

In Figure 6.14, *a* is any member of *A*. The reference to database states reminds us that the constraint applies to our domain model (not necessarily to the real world). So "*a* plays *r*" means *a* plays *r* in our model. In terms of fact tables, the set of values in the column for *r2* is always a subset of the set of values in the column for *r1*. It doesn't matter if any of these columns has duplicate values; we are comparing sets, not bags.

In Figure 6.14, it is assumed that *A* plays some other role and that both *r1* and *r2* are *optional*. If *r1* were mandatory, then a subset constraint to it would be *implied*, since the total population of *A* would then play *r1* (see Figure 6.15). In general, *A* has a mandatory role *r* only if there is a subset constraint to *r* from each of *A*'s roles.

To reduce clutter, implied subset constraints should normally be omitted from conceptual schemas. Thus a subset constraint between two roles is typically specified only if both of these roles are optional. This rule does not extend to composite subset constraints (see later).

Our fitness club application has one more constraint to consider. Reaction time is recorded for a member if and only if his/her heart rate is too. Each member has both fitness measures taken or neither. This might be because the club wants to provide a balanced estimate of fitness rather than risk reliance on a single figure. When the test data are entered, a compound transaction is used to enter both measures.

Such a constraint is called an **equality constraint**, since for any state of the database the set of people whose reaction time is recorded equals the set of people whose heart rate is recorded. This constraint is depicted by a *circled equality symbol* "=" connected by dotted lines to the relevant roles (look back at Figure 6.13). It asserts that the populations of these two roles must always be equal. An equality constraint is equivalent to two subset constraints, running in opposite directions.

In words, the equality constraint in Figure 6.13 may be expressed as **For each** Member, **that** Member has **some** ReactionTime **if and only if that** Member has **some** HeartRate. Figure 6.16 summarizes the notion of an equality constraint between two optional roles.

If both roles are mandatory, an equality constraint is implied since each of pop(*r1*) and pop(*r2*) equals pop(*A*). Implied equality constraints should normally be omitted (see Figure 6.17).

Figure 6.15 The implied subset constraint should be omitted.

Figure 6.16 An equality constraint between two roles.

Figure 6.17 The implied equality constraint should be omitted.

If one role is mandatory and the other is optional, then an equality constraint cannot exist between them since it is possible to have a state in which only some of *A*'s population plays the optional role. If neither role is mandatory but their disjunction is, then there is no equality constraint, otherwise both the roles would be mandatory (why?).

In principle, all mandatory roles and equality constraints on a conceptual schema could be replaced by groups of subset constraints. In practice, however, this would lead to untidy schema diagrams, with subset constraint lines running all over them. Hence, the notions of mandatory role and equality constraint are very convenient.

Now consider the report extract shown in Table 6.2. In this UoD, employees may request a company parking bay or a refund of their parking expenses, but not both. Employees may make neither request (e.g., they might not have a car or they might simply want more time to decide).

For this report, different kinds of null are displayed differently. The "–" value indicates "*not to be recorded*" (because the other option is chosen). The "?" value simply indicates "not recorded": an actual value might still be recorded for employee 005 later (e.g., after this employee buys a car); once such a value is recorded, however, the system must disallow the other option.

Table 6.2 Employee details.

EmpNr	*Employee name*	*Parking bay*	*Parking claim ($)*
001	Adams B	C01	–
002	Bloggs F	–	200
003	Collins T	B05	–
004	Dancer F	–	250
005	Eisai Z	?	?

Figure 6.18 Conceptual schema for Table 6.2.

Figure 6.18 shows a schema for this example. A circled "X" symbol "⊗" indicates an **exclusion constraint** between the roles it connects (X for eXclusion). This asserts that for each state of the database no employee can be recorded as playing both these roles. That is, the populations of these roles are *mutually exclusive*. A textual version of this constraint is: **For each** Employee **at most one of the following holds**: **that** Employee is allocated **some** ParkingBay; **that** Employee claims **some** parking Expense.

Figure 6.19 formalizes this notion of an exclusion constraint between two roles: their populations are exclusive just in case their intersection is the null set (i.e., they have no element in common). If two roles are played by different, primitive entity types then an implied exclusion constraint exists between them (since the entity types are mutually exclusive); such implied exclusion constraints are always omitted.

An exclusion constraint may be asserted between two roles only if these roles are optional and are played by the same object type (or possibly a supertype—see later). If one of the roles were mandatory, any object playing the other role would also have to play the mandatory one so an exclusion constraint could not apply. It is possible, however that the disjunction of these two roles is mandatory.

Now consider Figure 6.20. In this simple application, married partners are identified by their first names, and their country of birth must be recorded. For the ring predicate "is husband of", the inverse predicate name "is wife of" is also shown. No partner can be both a husband and a wife, so the husband and wife roles are mutually exclusive. Moreover, each partner is either a husband or a wife, so the same two roles are disjunctively mandatory.

Taken together, the exclusion and inclusive-or constraints assert that each partner is either a husband or a wife (of someone) but not both. These two constraints can be depicted together by superimposing the exclusion and mandatory symbols (⊗ and ⊙) to form the *life-buoy symbol* ⊛, connected by dotted lines to the relevant roles, as shown in Figure 6.20.

Figure 6.19 An exclusion constraint between two roles.

Figure 6.20 *Exclusive or:* each partner is a husband or wife but not both.

When exclusion is combined with inclusive-or, we have an **exclusive-or constraint**. Each partner plays *exactly one* of the constrained roles. So each partner appears in one column of this fact table, but no partner can appear in both columns. This exclusive-or constraint may be verbalized thus: **For each** Partner, **exactly one of the following holds**: **that** Partner is husband of **some** Partner; **that** Partner is wife of **some** Partner. Contrast this with the parenthood binary considered in an earlier chapter, where each person is a parent or child or both—a case of inclusive or.

The life-buoy symbol is also called a *partition symbol*, since it partitions an object type's population into separate role populations. We'll return to the notion of partitions when we discuss subtypes. Although the exclusive-or constraint may be depicted as a single symbol, it is really an orthogonal combination of two constraints. As a tool option, it may be displayed and verbalized as two separate constraints.

Subset or equality constraints should not be displayed with disjunctive mandatory roles. For example, if the two constraints in the left-hand schema of Figure 6.21 apply, then it follows that role r_1 is mandatory (as an exercise, prove this). For clarity, the mandatory role should be shown explicitly; the subset constraint is now implied by this, and hence should be omitted (see right-hand schema of Figure 6.21).

Similarly, an equality constraint between disjunctive mandatory roles should be redrawn as two mandatory roles. Such redrawing rules are pragmatic. Formally the left-hand schema is equivalent to the right-hand one. However, the right-hand version is preferred because it is simpler for people to work with.

As another case of redrawing, an exclusion constraint among three or more roles may be depicted by a single "⊗" connecting the roles, in preference to exhaustively marking exclusion constraints between each pair of roles. A simple example is shown in Figure 6.22. In this UoD, each person has at most one of three vices.

Figure 6.21 The left-hand schema should be redrawn as the right-hand schema.

Figure 6.22 In this UoD, each person has at most one of three vices.

In the left-hand diagram of Figure 6.22, three exclusion constraints cover all the ways to pair the three roles. Four roles may be paired in six ways. In general, a single exclusion constraint across n roles replaces $n(n-1)/2$ separate two-role exclusion constraints. An equality constraint across n roles may also be specified using a single, circled "=" instead of multiple binary equality constraints. No notation for a subset constraint over n roles is used, since direction is involved.

Although exclusion constraints between roles are common, in practice simple subset and equality constraints are seldom used on conceptual schemas (although they are often used on relational schemas). It is fairly common to encounter "qualified" set-comparison constraints, but these are depicted by subtyping (see next section).

Domains commonly involve *set-comparison constraints between role sequences* (not just between single roles), which we cannot handle by subtyping. Let's consider some examples. Table 6.3 provides details about people and the cars they own or drive. For simplicity, assume that people are identified by their first name and that each car is identified by a registration number stamped on its license plate. If we allowed a car to change its license plate we would need to pick some other identifier (e.g., its vehicle identification number or compliance plate number).

Figure 6.23 shows the populated conceptual schema. The ownership association is many:many. A person may own many cars, and the same car may be owned by many people. For example, Fred and Sue are co-owners of two cars (e.g., they might be married). The drives association is also many:many.

Table 6.3 Details about car owners and drivers.

Person	*Sex*	*Cars owned*	*Cars driven*
Fred	M	272NCP, 123BOM	272NCP, 123BOM
Sue	F	272NCP, 123BOM	272NCP
Tina	F	105ABC	
Tom	M		

Figure 6.23 A populated conceptual schema for Table 6.3.

Note that although Fred and Sue both own car 123BOM, only Fred drives it. Perhaps it is a "bomb" of a car and Sue refuses to drive it! Tina owns a car but doesn't drive it (maybe she has a chauffeur). Tom neither owns nor drives a car.

For the population shown, people own every car that they drive. Although unusual, for discussion purposes let us assume the sample database is significant in this respect. This rule is indicated in Figure 6.23 by the *pair-subset constraint* running from the middle of the Drives predicate to the middle of the Owns predicate. This asserts that, for each state of the database, the set of (Person, Car) pairs referenced in the Drives fact table is a subset of the set of pairs listed in the Owns table. We may think of each pair as a tuple of values from a row of a fact table.

For the database state shown, the Drives table has three pairs: (Fred, 272NCP); (Fred, 123BOM); and (Sue, 272NCP). The Owns table has five pairs. Each pair in the Drives table is also in the Owns table: the subset constraint indicates that this is true for each state of the database. This constraint may be verbalized as: **If some** Person drives **some** Car **then that** Person owns **that** Car. An alternative verbalization is: **Each** Person **who** drives **some** Car **also** owns **that** Car.

As usual, the direction of the subset constraint is important. With our present example, if the arrow pointed instead to the Drives fact type this would signify that each person who owns a car also drives that car, which is quite a different constraint.

To indicate that a subset constraint applies between role pairs rather than single roles, each end of the arrow is positioned at the junction of the two roles in the pair. This notation may be used when the roles in the pair are contiguous (see Figure 6.24).

For each database state:

$\text{pop}(r_1, r_2) \subseteq \text{pop}(r_3, r_4)$

Each *ab* **pair in** $\text{pop}(r_1, r_2)$ **is also in** $\text{pop}(r_3, r_4)$

Figure 6.24 A pair-subset constraint.

For each database state:

$pop(r_1, r_2, r_4) \subseteq pop(r_5, r_6, r_8)$

Figure 6.25 An example of a tuple-subset constraint between sequences of three roles.

In terms of the associated fact tables, the projection on columns r_3, r_4 of the lower table is a subset of the projection on columns r_1, r_2 of the upper table. If the roles in the pair are not contiguous, a connecting line is used between them. Role connectors are also used when the arguments to the constraint contain more than two roles (see Figure 6.25).

In general, a *tuple-subset constraint* may be specified between two compatible role tuples (their corresponding roles are type compatible), where each tuple is a sequence of n roles ($n \geq 1$). However, n is rarely greater than two. In rare cases, roles may need to be reordered before being compared; in this case the constraint may be annotated by including the relevant permutation.

Now consider the case where people own a car if and only if they drive that car. Here we have subset constraints in both directions: this is a *pair-equality* constraint. In such cases where the role pairs form a whole predicate, we usually store only one fact type. For example, we might store Drives and derive Owns, or collapse both to IsOwnerDriverOf. However, if a role pair is embedded in a longer predicate, an equality constraint is not equivalent to a derivation rule (why not?). Figure 6.26 summarizes the case of pair equality for contiguous roles.

Exclusion constraints may be specified between two or more role sequences. As a simple example, suppose we want to record information about cars that people own and cars that they want to buy. Clearly, nobody would want to buy a car that they already own. We can indicate this by the *pair-exclusion constraint* shown in Figure 6.27.

This constraint is weaker than an exclusion constraint between just the first roles of these predicates (which would instead say that no car owner wants to buy any car, and hence would disallow Fred's appearance in both the fact tables shown). Although we must record each person's gender, both the "owns" and the "wants-to-buy" roles are optional (even disjunctively). For example, Linda neither owns nor wants to buy a car.

For each database state:

$pop(r_3, r_4) = pop(r_1, r_2)$

Each *ab* **pair in** $pop(r_3, r_4)$ **is also in** $pop(r_1, r_2)$ **and vice versa**

Figure 6.26 A pair-equality constraint.

Figure 6.27 A pair-exclusion constraint: nobody wants to buy a car that they own.

In this section, set-comparison constraints have been considered between pairs of role-sequences, where each role sequence comes from a single predicate. More generally, set-comparison constraints may be defined between compatible role-paths formed by joining different predicates (join-constraints are covered in Section 10.1).

To conclude this section, let's look at some general results, or theorems, about set-comparison constraints. Suppose we declare a tuple-subset constraint from role sequence *rs1* to role sequence *rs2*, where each sequence has n roles ($n \geq 1$). Is it now possible for these role sequences to be exclusive?

If both role sequences are unpopulated, then both the subset and the exclusion constraints are trivially satisfied. But for practical reasons, we need every conceptual schema to be *strongly satisfiable* (or *population consistent*). This means that each role sequence used as a predicate or as a constraint argument can be populated in some state. It is not necessary that these role sequences be populated together in the same state.

Suppose we populate *rs1* with some sequence *a*. The subset constraint entails that *a* now populates *rs2* as well, thus violating the exclusion constraint. Swapping *rs1* with *rs2* in this reasoning shows a subset constraint in the other direction cannot hold either, if an exclusion constraint does.

This proves theorem *NXS* (*N*o e*X*clusion with a *S*ubset constraint), depicted in Figure 6.28. Here a long box depicts a sequence of one or more roles, and the constraints apply between the *whole* sequences (not different parts of them).

Figure 6.28 Theorem NXS: No eXclusion with a Subset constraint.

Figure 6.29 The implied constraints should not be shown.

An equality constraint is equivalent to, and used instead of, two subset constraints. So at most one set-comparison constraint may be declared between two (whole) role sequences. However, different set-comparison constraints may be declared between different parts of role sequences. For instance, we may have a subset constraint between the first roles and an exclusion constraint between the second roles. Subset or equality constraints between two role sequences imply similar constraints between the individual roles. In contrast, an exclusion constraint between single roles implies exclusion constraints between all sequences containing these roles.

These *constraint implication* results are depicted for role pairs in Figure 6.29, using "⇨" for "necessarily implies". In each case, the implication is in one direction only. Results for equality constraints are similar to the subset results. In the exclusion example, a similar result holds if the simple exclusion constraint is between the right-hand roles. Implied constraints should usually be omitted from a schema. As an exercise, you may wish to prove these results. If you have trouble here, try inventing some counterexamples to equivalence claims. For example, if pop(r_1, r_2) = {(a1, b1), (a2, b2)} and pop(r_3, r_4) = {(a1, b2), (a2, b1)}, then pop(r_1) = pop(r_3) and pop(r_2) = pop(r_4) but pop(r_1, r_2) ≠ pop(r_3, r_4).

Some other constraint implication theorems are mentioned later in the book. One important result deals with the case where the target of a pair-subset constraint includes a functional role. For example, the left-hand pattern in Figure 6.29(a) has a pair-subset constraint from the lower role pair to the top role-pair. Suppose we add a simple uniqueness constraint over the left-hand role of the top role pair. What does this imply about the left-hand role of the lower role pair? If the lower predicate is just a binary, this role must also be unique! As an exercise, prove this. An important corollary of this result is that if the lower predicate is longer than a binary, then it must split in this case, since there will be an implied FD from the first role to the second role. This result is illustrated in Section 14.6 when we discuss normalization.

Exercise 6.4

1. A company allows some of its employees to use one or more of its company cars. The rest of its employees are given a travel allowance instead. The following report is an extract from the company's records in this regard. Schematize it.

Emp. Nr	*Emp. name*	*Cars used*	*Driver's license*	*Travel allowance ($)*
001	Harding J	123ABC	A74000	–
002	Oh C	111KOR, 123ABC	A51120	–
003	Halpin T	–	–	200

2. The diagram shows the conceptual schema and current population for a given UoD. Reference schemes are omitted for simplicity. Fact tables appear below their fact types. Predicates are named **R**..**U**. Constraints are named *C1*..*C9*.

Each of the following requests applies to the **same** database population as shown on the diagram (i.e., treat each request as if it was the *first* to be made with this population).

For each request, indicate the CIP's response. If the request is legal, write "accepted". Otherwise, indicate the constraint violated (e.g., "C_2 violated").

(a) add: a1 R b2 (b) add: b4 S c3 (c) add: b1 S c3 (d) delete: b1 S c1
(e) add: b2 S c2 (f) add: b3 T c3 (g) add: U a2 a1 b2 (h) add: U a1 a2 b2
(i) begin
add: a1 R b4
add: b4 S c1
end

3. Draw the conceptual schema diagram for the UoD declared in Exercise 2.2 Question 2, adding constraint labels $C_1..C_5$ to the relevant constraints marked on the diagram.

4. The following table records details about various (fictitious) movies. Schematize this UoD. The population is not fully significant, so make some educated guesses.

Movie	*Director*	*Stars*	*Supporting cast*
Earth Song	Ima Beatle	Bing Crosby	Neil Diamond
		Elvis Presley	Tina Turner
Me and my echo	Hugh Who	Hugh Who	
Kangaroos	Doc U. Mentary		

5. A software company has a large number of shops located in the state of Queensland and requires detailed knowledge of their distribution. The table shows some of their locations. City shops are located in suburbs, but country shops are located in towns. City and town names are unique (other states are not of interest). Schematize this UoD, revealing the reference scheme of the entity type Location in full detail. Model street addresses as values but towns, suburbs and cities as entities.

Shop#	*Location*			
	Suburb	*City*	*Town*	*Street address*
1	St Lucia	Brisbane		Suite 5, 77 Sylvan Rd
2			Strathpine	Unit 3, 1000 Gympie Rd
3	Sandgate	Brisbane		7 Sun St
4	Clontarf	Redcliffe		25 Beach Rd
5	Sandgate	Cairns		25 Beach Rd

6. The application described here is partly based on a real life banking system, but simplifications and changes have been made. To help you appreciate the privacy implications of a universal identification scheme, all clients and staff of the bank are identified throughout the application by their tax file number (taxNr). Thus, all bank customers and personnel are taxpayers.

   An information system is required to manage accounts and staff records for Oz Bank, which has branches at various locations. Each branch is standardly identified by its branch number but also has a unique name. The first table is an extract from staff records of Oz Bank. Each employee works at exactly one branch and has at most one phone listed. The mark "?" denotes a null. The mark "…" indicates "etc." (other instances exist but are not shown here).

BranchNr	*Branch name*	*Emp. taxNr*	*Emp. name*	*Emp. phone*
1	Uni. of Qld	200	Jones E	3770000
		390	Presley E	?
		…	…	…
2	Toowong Central	377	Jones E	?
		…	…	…
3	Strathpine	222	Wong M	2051111
		…	…	…

   Within the one branch, each account has a unique serial number, but different accounts in different branches may have the same serial number. Account users are identified by their taxNr, but also have a name and possibly a phone number (see the second table).

   Each account is a passbook account. Five sample passbook entries are shown. For each account, transactions are numbered sequentially 1, 2, 3, etc. Dates are formatted day/month/year. For simplicity, assume that each transaction is either a deposit (DEP) or a withdrawal (WDL). In practice, other types of transaction are possible (e.g., interest and fees). The balance column shows the account balance after the transaction is executed.

   Although the balance is derivable, for efficiency purposes *the balance is stored as soon as it is derived*. For example, this speeds up the production of monthly statements for the bank's customers (Oz bank has a few million customers who average several transactions each month). This *derive on update* (eager evaluation) decision contrasts with our normal *derive on query* (lazy evaluation) policy.

Account		*User*		
BranchNr	*SerialNr*	*TaxNr*	*Name*	*Phone*
1	55	200	Jones E	3770000
		311	Jones T	3770000
1	66	199	Megasoft	3771234
2	55	199	Megasoft	3771234
2	77	377	Jones E	?
3	44	300	Wong S	2051111

OZ BANK

BranchNr	SerialNr
1	55

Branch name: Uni. of Qld
Users: Jones E; Jones T

TranNr	Date	Deposit	Withdrawal	Balance
1	3/1/90	1000		1000
2	5/1/90		200	800
3	5/1/90		100	700

OZ BANK

BranchNr	SerialNr
1	66

Branch name: Uni. of Qld
Users: Megasoft

TranNr	Date	Deposit	Withdrawal	Balance
1	10/2/90	2000		2000
2	10/2/90		500	1500

OZ BANK

BranchNr	SerialNr
2	55

Branch name: Toowong central
Users: Megasoft

TranNr	Date	Deposit	Withdrawal	Balance
1	23/1/90	9000		9000
2	7/2/90	5000		14000
3	10/2/90		2000	12000
4	2/3/90		5000	7000

OZ BANK

BranchNr	SerialNr
2	77

Branch name: Toowong Central
Users: Jones E

TranNr	Date	Deposit	Withdrawal	Balance
1	3/1/90	500		500

OZ BANK

BranchNr	SerialNr
3	44

Branch name: Strathpine
Users: Wong S

TranNr	Date	Deposit	Withdrawal	Balance
1	5/1/90	100		100
2	12/1/90	600		700

(a) Draw a conceptual schema diagram for this UoD. Make use of an entity type indicating the type of transaction: TransactionType (code) {'DEP', 'WDL'}. Include the account balance fact type on the diagram; mark it with "**" since it is derivable but it is required to be stored. Express the derivation rule as clearly as you can.

(b) Consider the fragment of your conceptual schema that captures the kind of information required for deposits and withdrawals (only two fact types are involved). Transform this subschema into an equivalent subschema that uses two different fact types, removing the entity type TransactionType altogether.

(c) In a realistic banking application there are several kinds of transaction in addition to deposits and withdrawals. In such a situation is it better to include TransactionType as an object type or to extend the alternative approach proposed in (b)?

6.5 Subtyping

If instances of an object type are classified into a more specific type, this specialized type is known as a **subtype**. Consider, for example, the Person type, shown in Figure 6.30. For discussion purposes, this is populated with eight people, depicted by clip art shapes. Each person's gender and birth country are recorded. The two instances of Gender are denoted by the biological symbols for male and female. Countries are denoted by their map shapes (only Australia, Italy, and Mexico are shown here).

Based on gender, we could classify Person into the subtypes MalePerson and FemalePerson. We've placed the males on the left and the females on the right in Figure 6.30, so you can visualize creating these subtypes by vertically dividing the Person type in two. We could also classify persons into subtypes based on their country of birth. In Figure 6.30 this is depicted by the horizontal division of Person into Australian, Italian, Mexican, etc. For the small population shown, we have three Australians, two Italians, and three Mexicans, with other nationalities in various layers below (simply marked "etc." here). For simplicity, let's ignore issues about naturalization, so the term "Australian" is used here in the sense of "Australian-born", and so on.

The main reason for using a subtype in modeling is to *declare typing constraints* (in this case, to *declare that one or more specific roles are played only by a given subtype*). For example, a medical database might record whether a patient was circumcised, but this would be recorded only for male patients.

Figure 6.30 Person classified into subtypes according to gender or birthplace.

A second reason for subtyping is to *encourage reuse of model components*. For example, if we already have an elaborate subschema describing the object type Employee and want to talk about managers, we can declare Manager as a subtype of Employee and simply add the extra details specific to managers.

If Employee is defined in a reference model, it can be imported into many different models and specialized as appropriate for each case. If an object type has behavior defined for it, this can be reused as well as the data aspects. In programming applications, this can significantly reduce code duplication.

A third reason for subtyping is to reveal a *taxonomy* or classification scheme being used in the UoD. However, in ORM, we often avoid introducing subtypes purely to expose a classification scheme. Why? First, taxonomies may often be modeled in more economical ways. For example, to indicate that there are two genders we may simply add a value constraint such as {'M', 'F'} on Gender (.code). There is no need to introduce Male and Female subtypes unless we have some specific roles for them to play in our application domain.

Second, the number of subtypes can get out of hand rapidly if we use them purely for taxonomy. For example, suppose we wish to classify people based on country of birth. All we need is the fact type: Person was born in Country. If instead we modeled this by introducing subtypes of Person, one for each country, this would lead to almost 200 subtypes in the schema!

This is crazy. If one of these subtypes plays a specific role, then we should introduce it, but if it doesn't, we have no need to. For example, suppose we wanted to record whether Australians want their country to become a republic, but had nothing specific to record about the other nationalities. In that case, Australian is the only subtype we should introduce.

Figure 6.31 Euler diagrams are okay for simple subtyping (a) but not complex cases (b).

Earlier in the chapter, Euler diagrams were used to depict one set as a subset of another. In designing a conceptual schema we often need to introduce subtypes and spell out clearly what is a subtype of what. Euler diagrams may be used to do this for simple cases, such as Figure 6.31(a) where *B* and *C* are subtypes of *A*.

Interpreting this as a hypothetical Euler diagram, the question is open as to whether *A* contains any elements not in *B* or *C*. If *A* does have more elements, this can be shown by adding an asterisk to the region outside *B* and *C*. As discussed in Section 8.2, the Barker notation for ER instead uses a special type called "Other" for this purpose.

The good thing about Euler diagrams is that they show containment intuitively, by placing one set or type spatially inside another. This works well for simple cases. In practice however, an object type may have many subtypes, which might overlap, and a subtype may have many supertypes.

For example, the subtype pattern in Figure 6.31(b) depicts the following information: *B*, *C* and *D* are subtypes of *A*, and *E* is a subtype of both *C* and *D*. Moreover, *B* *overlaps* with *C* (i.e., they may have a common instance) and *C* overlaps with *D,* but *B* and *D* are *mutually exclusive* (cannot have a common instance). For example, *A* = Person; *B* = Asian; *C* = Consultant; *D* = American; *E* = TexanConsultant.

For such cases, Euler diagrams become hopelessly complicated. Moreover, individual subtypes may have many specific details recorded for them, and there is simply no room to attach these details if the subtype nodes are crowded together inside their supertype nodes.

For such reasons, Euler diagrams are eschewed for nontrivial subtyping. What we need is a simple notation that can be used to display subtypehood and subtype roles no matter how large or complex the situation is. This is achieved by displaying subtypes *outside* their supertype(s) and depicting the subtype–supertype connections by means of an *arrow*.

The basic idea is shown in Figure 6.32. Here *A* and *B* are object types. Suppose *A* = Employee and *B* = Manager. The type Employee is the set of all employees about which facts might possibly be recorded. The subtype Manager is the set of all managers about which facts might be recorded. At any given database state, pop(Employee) and pop(Manager) are, respectively, the set of employees and set of managers actually referenced in the database.

Figure 6.32 *B* is a proper subtype of *A*.

In general, *B* is a **proper subtype** of *A* if and only if pop(*B*) is always a subset of pop(*A*), and $A \neq B$. In this case, *A* is a *proper supertype* of *B*. If $A = B$ we do not specify any subtype connection. With this understanding, we usually omit "proper" when speaking about subtypes and supertypes.

The adjective "proper" applies to the type relationship, but not necessarily to the *population* relationship. There may be a database state in which pop(*A*) = pop(*B*). For example, if no facts about *A* have yet been entered, both pop(*A*) and pop(*B*) equal { }; with our example, we may choose to enter information about managers before the other employees.

This notation is less intuitive than Euler diagrams, because a subtype is shown "outside" its supertype even though every object instance within the subtype must also be contained inside the supertype. Moreover, information about subtype overlapping is lost unless we add some way to capture it.

This is a consequence of depicting subtype connections by arrows rather than spatial containment. However, this disadvantage is more than offset by the ability to conveniently represent subtype patterns of arbitrary complexity, while still allowing plenty of space around each node for details to be attached.

Using this notation, the complex subtype pattern shown in the Euler diagram of Figure 6.31(b) may be modeled as shown in Figure 6.33. Here *A* is a *common supertype* of *B* and *C* and *D*, and *E* is a *common subtype* of *B* and *C*. In general, a common supertype is at least the union of its subtypes; if it is the union of its subtypes then its subtypes are said to *exhaust* it. In our example, *B*, *C*, and *D* do not exhaust *A*. Note that Figure 6.33(a) depicts only the subtype connections.

Figure 6.33 This is another way to model the subtype connections in Figure 6.31(b).

Figure 6.34 *B* and *C* are mutually exclusive (a), exhaustive (b), or both (c).

Figure 6.31(b) contains the additional information that *B* overlaps with *C*, *C* overlaps with *D*, and *B* and *D* are mutually exclusive. Hence if the subtype arrow notation is used, some other means must be used to convey whether or not subtypes are mutually exclusive. One way to do this in ORM is to attach an *exclusion* symbol "⊗" via dotted lines to the relevant subtype links.

The absence of an exclusion symbol indicates that the populations of the types might overlap. With this added symbol, Figure 6.33(b) now conveys the exclusion/overlap information of the Euler diagram in Figure 6.31(b).

To indicate that two or more subtypes are *exhaustive* for their supertype (i.e., their union equals the supertype), an *inclusive-or* constraint symbol "⊙" may be connected by dotted lines to the relevant subtype links, as in Figure 6.34(b). This constraint applies to the populations as well as the types. So each instance in the population of the supertype must appear in the population of at least one of the subtypes.

If we combine both constraints, we have an *exclusive-or* constraint indicating that the supertype is *partitioned* into its subtypes, as shown in Figure 6.34(c).

Implicitly, underlying each subtype arrow there is a derived, instance-level 1:1 binary association "is / is". The subtype exclusion and exhaustion constraints correspond to exclusion and inclusive-or (disjunctive mandatory) constraints on the roles played by the supertype in these associations—hence the choice of symbols for these constraints.

In this light, the constraints in Figure 6.34 may be verbalized respectively as **No *A* is both *B* and *C*; Each *A* is some *B* or *C*; Each *A* is exactly one of *B* or *C*.**

Subtypes may overlap, especially if they are based on different classification schemes. For example, in Figure 6.35 the subtypes Australian and Female overlap. Since overlapping subtypes have elements in common, a common subtype may be formed from one or more elements in their intersection. For example, the subtype FemaleAustralian is a subtype of both Australian and Female.

The subtypehood relation is *transitive*: if *A* is a subtype of *B* and *B* is a subtype of *C*, then *A* is a subtype of *C*. In this case, *A* is said to be an *indirect subtype* of *C*. In Figure 6.35, for example, FemaleAustralian is a direct subtype of Australian, which is a direct subtype of Person. Hence, FemaleAustralian is an indirect subtype of Person. For exposition purposes, this indirect link is marked on the diagram with an asterisk to show its derived nature. To avoid clutter, however, indirect subtype links should be omitted from diagrams since they are transitively implied.

Figure 6.35 The indirect subtype connection is implied, so should be omitted.

If a subtype has two or more direct supertypes, this is called *multiple inheritance*. For example, in Figure 6.35, FemaleAustralian inherits from both Australian and Female. In general, a common subtype is at most the intersection of its supertypes. If a subtype has only one direct supertype, this is called *single inheritance*.

A supertype may have many direct subtypes, and a subtype may have many direct supertypes: so in general we have a *subtype graph* rather than a tree. The supertype and subtypes are referred to as the *nodes* of the graph. Since the arrowheads provide direction, we have a directed graph.

Since no type can be a proper subtype of itself, it follows that no cycles or loops are permitted, no matter what the length of the cycle (Figure 6.36 shows two illegal cycles). Thus any pattern of type–subtype relationships forms a *directed acyclic graph*.

An entity type that is not a proper subtype of any other entity type in the schema is said to be a *primitive entity type* for the schema. In Figure 6.30, for example, Person, Gender, and Country are primitive entity types. A single conceptual schema may have many subtype graphs (e.g., one based on Person and one based on Vehicle). Each subtype graph must stem from exactly one primitive entity type (which may be an objectified association) that is the common supertype, or *root* node (or *top*) of that graph.

Figure 6.36 No cycles are permitted in a subtype graph.

Figure 6.37 A subtype graph may arise by (a) specialization or (b) generalization.

Subtype graphs have only one root, since primitive entity types are mutually exclusive. In contrast, subtypes in a graph necessarily overlap with their supertype(s) and might even overlap with one another (e.g., Australian and Female). Since root entity types are mutually exclusive, there is no overlap between entity subtypes from different subtype graphs. To reinforce these general ideas about subtypes, you may wish to try Question 1 of the section exercises before continuing.

A subtype graph may arise in a top-down way, by *specializing* an object type into subtypes (subtype introduction), as in Figure 6.37(a). A subtype graph may also arise in a bottom-up way, by *generalizing* object types to a common supertype and retaining the subtypes for specific details (supertype introduction), as in Figure 6.37(b). Sometimes a subtype graph arises from a combination of specialization and generalization. The rest of this section discusses the specialization process, and the following section deals with the generalization process.

Now consider the extract from a hospital patient record system shown in Table 6.4. Only males have a prostate gland. In later life this gland may suffer various medical problems. Only women can become pregnant. In this table, a question mark "?" is an ordinary null, indicating that an actual value is unknown. The *minus sign* "–" is a special null, indicating that an actual value is *not to be recorded* there.

We met something like this when we considered subset and exclusion constraints. But there is a difference. Here the "–" means that an actual value is *inapplicable because of the specific value of some other entry* (or entries) for the entity involved.

On the first row, pregnancies must not be recorded for patient 101, because that patient is male ("M" entry for gender). Prostate status must not be recorded for women. The number of pregnancies must be recorded for women, even if this number is zero.

In contrast to "–", the "?"merely indicates that the information is missing. For example, patients 103 and 104 might have no phone, and patient 105 might not have had his prostate checked. Suppose we schematize this UoD as in Figure 6.38.

Table 6.4 Details about hospital patients.

PatientNr	*Name*	*Gender*	*Phone*	*Prostate status*	*Pregnancies*
101	Adams A	M	2052061	OK	–
102	Blossom F	F	3652999	–	5
103	Jones E	F	?	–	0
104	King P	M	?	benign enlargement	–
105	Smith J	M	2057654	?	–

Figure 6.38 An incomplete conceptual schema for Table 6.4.

Notice the optional roles. Although correct as far as it goes, this schema fails to express the constraints that prostate status is recorded *only for* male patients, and the number of pregnancies is recorded *just for* the females. The phrase "just for" means "for and only for" (i.e., "for all and only"). To capture these constraints, we introduce subtypes, and attach their specific roles, as shown in Figure 6.39.

The subtypes MalePatient and FemalePatient are marked with an *asterisk* "*" to indicate that they are *derived subtypes* (i.e., they are derived by applying a derivation rule to their supertype(s)).

The derivation rules appear as formal *subtype definitions* beneath the diagram. By default, subtypes inherit the identification scheme of their supertype, so there is no need to repeat it here.

Recall that a role is played only by instances of the object type to which the role is attached. Hence the subtyping reveals that prostate status is known (to the information system) **only if** the patient is male, and pregnancy count is known **only if** the patient is female. The role attached to MalePatient is optional. Not all men need to have their prostate status recorded.

However, the role attached to FemalePatient is mandatory. So pregnancy count is recorded **if** the patient is female. The combination of the subtype constraint and the mandatory role on FemalePatient means that a pregnancy count is recorded **if and only if** the patient is female.

Figure 6.39 Subtyping completes the conceptual schema for Table 6.4.

In this case, the MalePatient subtype plays only one role, and this is optional. Unlike primitive entity types, there is no default assumption of mandatory if only one role is attached to a subtype. *With subtypes, any mandatory role constraints must be explicitly shown* (including disjunctive cases).

A subtype with no specific roles attached is said to be an *inactive subtype*. For example, MalePatient and FemalePatient subtypes might be introduced merely for taxonomy purposes as a way to display that patients can be classified in this way. In ORM, we rarely use inactive subtypes, as it is usually better to simply display such a classification scheme as a binary fact type with a value constraint (e.g., Figure 6.39 uses Patient is of Gender {'M', 'F'}) or as unaries (e.g., isMan and isWoman may be attached to Patient as exclusive, disjunctively mandatory unaries). The binary form is much neater when the classification scheme involves several subtypes.

While specific fact types are attached to the relevant subtype, common fact types are attached to the supertype. In this example, name, gender, and phone details may be recorded for any patient, so are attached directly to Patient. In general, each subtype inherits all the roles of its supertype(s). Although drawn outside its supertype, the subtype is totally contained inside the supertype. In Figure 6.39, all instances of MalePatient or FemalePatient have their name, gender, and possibly phone number recorded. Attaching a common fact type to the supertype avoids duplicating it on all the subtypes.

To determine membership of derived subtypes, *each derived subtype must be defined in terms of at least one role played by its supertype(s)*. With the present example, the gender fact type is used to determine membership in MalePatient and FemalePatient. The subtype definitions are formal—they are not just comments.

The operator "**is a**" or "**is an**" is used for "is defined as", and "**who**" or "**that**" is used after the supertype name. For persons, "who" sounds more natural than "that". The quantifier "**Each**" normally precedes the name of the subtype being defined, as shown in Figure 6.39.

Here each MalePatient is a Patient, but since derived subtypes must be well defined, this "is a" connection must be *qualified* in some way. For example, **Each** MalePatient **is a** Patient **who** is of Gender 'M'. Subtyping provides a means of *qualifying an optional role, a simple set-comparison constraint*.

Subtyping may also be used *instead of unqualified, simple set-comparison constraints, especially if the subtype plays many specific roles*. Consider a UoD where each employee drives cars or catches buses but not both. Cars or buses are recorded for all employees, bus allowance is recorded just for the bus catchers, and driver's license numbers are recorded only for the drivers. This can be modeled without subtypes, using exclusion, equality, and subset constraints. As an exercise draw the schema for this and then try a subtyping solution.

Different subtyping approaches are possible. One solution is to specify the fact type Employee drives Car and then introduce the subtypes Driver (Employee **who** drives **some** Car) with the license predicate attached and NonDriver (Employee **who** drives **no** Car) with both bus caught and bus allowance predicates attached. The more subtype-specific information required (e.g., total distance traveled by bus, driving violations) the tidier the subtyping portrayal becomes in comparison to the no-subtyping alternative. Note however

that subtyping cannot replace composite set-comparison constraints (between sequences of two or more roles).

Be careful to *avoid circular definitions*—don't try to define a subtype in terms of itself. For example, suppose we defined BusCatcher as Employee **who** catches **some** Bus and then attached the bus predicate to this subtype (i.e., BusCatcher catches Bus). This specifies that the role of catching a bus is played only by those who play the role of catching a bus. Such circular reasoning is not very informative! In very rare cases, a role used to define a subtype may be a reference role. For example, we might record some property only for people whose surname begins with the letter "Z".

For the schema of Figure 6.39, the subtypes MalePatient and FemalePatient are mutually exclusive and collectively exhaustive. When these two conditions are met, the subtypes are said to form a partition of their supertype. We may display this situation on a hypothetical Euler diagram as shown in Figure 6.40(a). This is analogous to the process of partitioning or slicing a pie or pizza into pieces, as shown in Figure 6.40(b). The pieces don't overlap (mutual exclusion), and all the pieces together make up the whole pizza (collective exhaustion).

Our understanding of the subtype names "MalePatient" and "FemalePatient" helps us to "see" a partition. But such names are only character strings to the computer system, so how is the partition formally captured in the model?

Look back at Figure 6.39. The mutual exclusion between MalePatient and FemalePatient is implied by their definitions, together with the uniqueness constraint on the gender predicate (**Each** Patient is of **at most one** Gender). The constraint that MalePatient and FemalePatient exhaust Patient is implied by the subtype definitions, the {'M','F'} constraint, and the mandatory role on the gender predicate (**Each** Patient is of **at least one** Gender). Because ORM demands that derived subtypes be well defined and all relevant constraints on defining predicates be declared (graphically or textually), *any exclusion or exhaustion constraints on derived subtypes are always implied.*

As humans, we can use our understanding of meaningful subtype names to "see" whether subtypes are exclusive or exhaustive. This shortcut depends on a judicious choice of subtype names. For example, if we unwisely chose the subtype names "Man" and "Woman" to partition the Patient type, the exhaustion constraint would not apply in any UoD that includes child patients or adult people who are not patients.

For clarity, it is usually best to display subtype exclusion and exhaustion constraints explicitly, even if these constraints are implied. With very complex subtype graphs, the explicit display of such implied constraints can make the diagram appear cluttered. Ideally, a CASE tool should enable display of such constraints to be toggled on or off.

Figure 6.40 A partition of Patient (a) and a pizza (b).

Figure 6.41 The constraints are satisfied, but can you spot a problem?

Since MalePatient and FemalePatient are exclusive, and an object type must be populatable, no object type can be a subtype of both, since such a type would always be empty (nobody is simultaneously both a man and a woman). In general, *exclusive object types cannot have a common subtype*.

In some other modeling methods a classification predicate or attribute (e.g., gender) may be specified, then subtypes introduced but not formally defined, and relevant exclusion and exhaustion constraints added. This alternative approach suffers from the problem that *exclusion and exhaustion constraints are weaker than subtype definitions*. For example, consider the populated schema in Figure 6.41. Here the partition constraint is declared but the subtype definitions are removed. The sample population satisfies all the constraints, but there is still something wrong. Can you spot the problem?

As you no doubt noted, prostate status is recorded for patient 102 (a female) and a pregnancy count is recorded for patient 101 (a male). This is nonsense.

We need to record the gender of each patient regardless of their subtype roles, but if they do play a subtype role we need to check that they are in the right subtype. The normal way to do this in ORM is to define the subtypes formally in terms of their gender and then enforce the definition. Figure 6.42 shows the correctly populated schema for this approach.

Figure 6.42 The subtype definitions are stronger than the partition constraints.

Figure 6.43 With asserted subtypes, exclusion/exhaustion constraints are not implied.

In conjunction with the constraints on the gender fact type, the subtype definitions ensure not only that the Patient type is partitioned into the subtypes, but that patient instances appear in the correct subtype.

Note that *subtype exhaustion constraints apply to the populations of the subtypes, not necessarily to their active populations* that play a specific subtype role. Here the prostate status role is optional for male patients, so some instances of MalePatient (in this case patient 103) need not play this role.

The problem with the model in Figure 6.41 arose because the patient taxonomy was expressed in two ways (with a gender fact type and with subtyping) that were not kept in sync. The solution in Figure 6.42 used subtype definitions to keep these two specifications of the taxonomy consistent. An alternative solution is to specify the taxonomy in one way only, removing the gender fact type, and simply asserting the subtyping scheme with no subtype definitions, as shown in Figure 6.43.

A subtype that is simply asserted (no derivation rule) is called an *asserted subtype*. Unlike derived subtypes, asserted subtypes are not marked with an asterisk. Introducing an asserted subtype is equivalent to attaching a unary predicate (e.g., isMalePatient) to the supertype. With asserted subtypes, any subtype exclusion/exhaustion constraints must be explicitly declared, as there are no subtype definitions to imply them.

A third alternative is to derive a taxonomy fact type from the asserted subtypes, as shown in Figure 6.44. Although this practice is typically closer to the way subtyping is implemented in object-oriented programming, the first solution using an asserted taxonomy fact type for gender and derived subtypes with definitions is closer to the way subtyping is best implemented in relational database systems.

Figure 6.44 Deriving gender from membership in asserted subtypes,

But the main point to remember is that *if a taxonomy is captured both by a classifying fact type and by subtyping then these two ways must be kept consistent by means of derivation rules* (either subtype definitions or fact type derivation rules).

Sometimes a classification predicate is required. For example, we may need to record the salary of employees and then use this to subtype employees. Employees with a salary ≥ $80,000 might be classified as HighPaidEmployee, for instance. Here we could not derive the precise salary of an employee by knowing his or her membership in an asserted subtype. For such reasons, as well as uniformity, in ORM we normally prefer to derive rather than assert subtypes.

In very rare cases, a subtype may be semiderived (some instances may be derived, and some may be simply asserted). Whereas derived subtypes are fully determined by their definitions, *semiderived subtypes* are only partly determined by an implication rule. This rare situation usually occurs only when we have incomplete knowledge about the fact type used in the derivation rule.

For example, in Figure 6.45(a) the Grandparent subtype may be derived because we have complete knowledge about parenthood facts. If instead we have incomplete knowledge of parenthood, then we might know that someone is a grandparent without knowing who their children or grandchildren are. This situation is depicted in Figure 6.45(b) using a semiderived subtype for Grandparent.

Semiderived subtypes are marked with a "+" superscript (intuitively, half an asterisk), and the derivation rule is specified as an implication rather than an equivalence rule (either by inserting "derived" before the subtype name, or by using "if" instead of "iff").

The *specialization* (subtype introduction) procedure may now be summarized as follows. Here "known subtype" means that the subtype is known in the UoD. A subtype definition is stronger than a set-comparison constraint if it adds a restriction to a defining predicate (e.g., Gender 'M', Rating > 3). The procedure is recursive: since a subtype may itself have optional roles, we apply the procedure to it to see if we need to form subtypes of it.

Figure 6.45 Derived (a) and semiderived (b) subtypes,

Specialization Procedure (SP):

Specify all mandatory role constraints;

For each optional role:
if it's recorded only for a known subtype
  and (there is a subtype definition stronger than a set-comparison constraint
    or another role is recorded only for this subtype)
then introduce the subtype;
  attach its specific roles;
  declare the subtype derived (*), asserted, or semiderived (+);
  unless the subtype is asserted, add a subtype derivation rule;
  apply SP to the subtype

Note that exclusion and exhaustion constraints among subtypes are to be interpreted in a *static* sense. For example, since the subtypes MalePatient and FemalePatient are exclusive, in any given database state their populations cannot overlap. However this does not rule out an object migrating from one subtype to another between database states. This is unlikely for our present example!

However, our next example includes the subtypes SeniorLecturer and Professor (see Figure 6.46). SeniorLecturer and Professor are exclusive in the sense that no academic can be a member of both types simultaneously.

Note that a senior lecturer in one state can be promoted to professor in a later state. The subtypes SeniorLecturer and Professor are said to be *role types* (migration allowed) rather than rigid types (migration forbidden). This distinction is discussed in more detail in Chapter 10.

In this domain, academics have exactly one rank (lecturer, senior lecturer, or professor) and have been awarded one or more degrees. Each professor holds a unique "chair" (e.g., information systems), indicating the research area that he or she manages. Each student has at most one counselor, and this must be a senior lecturer.

Nothing special is recorded for lecturers, so we don't bother introducing a subtype for them. The subtype definitions and the constraints on the rank predicate imply that the SeniorLecturer and Professor subtypes are exclusive but not exhaustive. The exclusion constraint is implied.

Although subtypes are drawn outside their supertypes, every member of a subtype is also a member of its supertype(s).

Figure 6.46 The subtypes are exclusive but not exhaustive.

Figure 6.47 The subtyping of Figure 6.46 shown with a hypothetical Euler diagram.

To clarify this point, Figure 6.47 depicts the subtyping of Figure 6.46 by means of a HED; the shaded region is empty. For simplicity, reference schemes and subtype definitions are omitted. Although such diagrams may be used in simple cases for explanatory purposes, they are too awkward for complex cases and are no replacement for our subtyping notation.

Consider now a UoD peopled just by lecturers and students. Gender and PersonKind (L = Lecturer; S = Student) are recorded for all persons, salary is recorded just for lecturers, and course of study is recorded just for students (see Figure 6.48(a)). Here, a person may be both a lecturer and a student, as indicated by the many:many constraint. So the Student and Lecturer subtypes overlap. Although not exclusive, they are exhaustive, given the mandatory role and {'L', 'S'} constraint on the personkind fact type.

Figure 6.48(b) displays the subtyping on a HED. Since these subtypes overlap, they may have a common subtype. For example, if some further information was required only for those lecturers studying a course, we could introduce a subtype StudentLecturer (both Student and Lecturer).

Suppose we expanded the UoD to include administrative staff, who may be students but not lecturers. This classification scheme may be depicted by three unaries (studies, lectures, and administrates) with an exclusion constraint between the last two. If specific details were required for administrators, we would introduce a third subtype for this. As an exercise, draw a schema diagram and a HED to depict this situation.

The rest of this section focuses on the problem of providing a correct definition for subtypes. Sometimes it is not obvious from an output report what criteria have been used to decide whether some fact is to be recorded. As a simple example, consider Table 6.5. What rule determines when to record a person's favorite group?

Figure 6.48 Here, Student and Lecturer are overlapping and exhaustive subtypes.

Table 6.5 What determines whether we record a person's favorite group?

Person	*Age*	*Favorite group*
Bill	17	Abba
Fred	12	–
Mary	20	–
Sue	13	Dire Straits
Tom	19	Beatles

One pattern that fits the sample data is to record a person's choice of favorite group if and only if the person is aged between 13 and 19 inclusive (i.e., the person is a teenager). This is not the only pattern that fits, however. Maybe favorite group is recorded just for people with odd ages (13, 17, and 19 are odd numbers).

While these might be argued to be the two most "obvious" patterns, there are infinitely many patterns that are consistent with the data. Just based on age, we could specify any set of natural numbers minus the set {12, 20} and within the age range of the UoD.

A database population is *significant with respect to a constraint* if and only if it satisfies no alternative choice for this constraint (excluding weaker constraints implied by this choice). In this case, the constraint is suggested by the population. Obtaining a sample population that is significant with respect to uniqueness constraints can be tedious, but at least it can be done in a modest amount of time. Unfortunately, no decision procedure exists to automatically churn out the correct subtype definitions.

If we remove the restriction that a computer system has finite memory, then we encounter the problem that for any finite set of data there will always remain an infinite number of possible patterns that fit the data. This is the basis of the philosophical "problem of induction".

As a scientific example, consider the graph shown in Figure 6.49. This shows three data points determined from an experiment in which the pressure (p) and absolute temperature (T) of a gas are measured at a fixed volume. What pattern, rule or law does this suggest to connect pressure and temperature?

One pattern that fits the data is the linear relationship between pressure and temperature shown by the straight line in Figure 6.50(a). This indicates that when the volume of gas is fixed, its pressure is directly proportional to its absolute temperature. You may recall learning this law of physics in high school. However, the curved line in Figure 6.50(b) also fits the data. How do we know which (if any) is correct?

Figure 6.49 What pattern matches the data?

Figure 6.50 More than one pattern fits the data.

An obvious move is to pick a temperature where the two patterns differ in their pressure reading, and then measure the actual reading. The result is shown in Figure 6.51(a). Here the extra data point (the third "×") fits the straight line hypothesis, but not the curved line. This counterexample is enough to disprove that curved line. Does this prove that the straight line law is correct? No. We can always add another curved line to fit the expanded data, as shown by the dotted line in Figure 6.51(b). Indeed there are infinitely many curved lines that fit the data points. You can keep this process going as long as you like. Any finite set of data is consistent with infinitely many rules. Hence you can never really prove any scientific law.

Returning to our subtype definition problem, it is clear that any subtype data sample is consistent with infinitely many subtype definitions. Instead of getting bogged down in philosophical speculation at this point, you can use the following pragmatic approach to finalizing subtype definitions.

Look carefully at the data, use "common sense" to spot the simplest pattern, and then check with the domain expert whether this pattern is the one intended. Since the subtyping constraints reflect the decision of the domain expert on what should be recorded for what, the domain expert will always be able to resolve the matter. In a sense, it's like asking God what the rules are for the universe He/She created.

Although output reports cannot be significant with respect to subtype definitions, input forms such as tax returns or application forms often do provide a significant set of conditional instructions from which the subtyping definitions can be deduced. Typically such forms include a number of fields to be filled in only by certain kinds of people. To let users know which questions they must answer, instructions are included that reveal the conditions under which the specific details are to be recorded. From these instructions it is usually fairly easy to determine the subtyping.

Figure 6.51 Extra data can eliminate a pattern, but other patterns may still fit.

FormNr: 5001

1. Age (years): .....

2. Nr hours spent per week watching TV: .....

If you answered 0 then go to Question 4

3. What is your favorite TV channel? .....

---

4. Nr hours spent per week reading newspapers: .....

If you answered 0 then Stop (no more answers are required).

5. What is your favorite newspaper? .....

If you are younger than 18, or answered 0 to question 2 or 4 then Stop (no more answers are required)

---

6. Which do you prefer as a news source? (Check the box of your choice)

❑ Television

❑ Newspaper

Figure 6.52 A sample media survey form.

As a simple example, consider the media survey form shown in Figure 6.52. Each copy of the form has a unique form number. Each form is filled out by a different person, and each person fills out only one form. The enforcement of this 1:1 correspondence between people and forms is the responsibility of the company conducting the survey rather than the information system itself. This correspondence is left implicit in the model, where people are identified directly by the form number.

The conditional instructions on this form are shown in italics. Everybody must answer questions 1, 2, and 4. Anybody who answers 0 to question 2 is told to skip question 3. Hence favorite TV channel is recorded just for those who indicate they do watch some TV; let's call this subtype Viewer. People who answered 0 for question 4 are told to skip all later questions. Hence favorite newspaper (question 5) is recorded just for those who read newspapers; let's call this subtype Reader.

People who are younger than 18 or are not both viewers and readers skip question 6. Hence the preferred news source is recorded just for adult viewers who are also readers; let's call this subtype MediaAdult. This analysis leads to the schema of Figure 6.53.

Alternatively, an output report from the domain might help with the modeling (see Table 6.6). The sample data have been carefully chosen to be significant with respect to the subtype graph. Given this, and recalling that the "–" sign means "not to be recorded", we may reason as follows. Age, viewing, and reading figures are recorded for everybody. The set of people for which favorite channel is recorded properly overlaps with the set of people for which favorite paper is recorded, so these correspond to overlapping subtypes. The set of people for which preferred news source is recorded is a proper subset of the previous two sets, so this corresponds to a common subtype.

Figure 6.53 The conceptual schema for the media survey example.

This analysis yields the "diamond shaped" subtype graph of Figure 6.53, as well as the information recorded for each node in the graph. However, we can only make educated guesses as to the actual subtype definitions—these should be checked with a domain expert if one is available.

In practice, reports often display different kinds of nulls in the same way. For example, a blank space might be used for both our simply unknown "?" and inapplicable "–" marks. So you should clarify the meaning of any nulls with the domain expert before deciding how to handle them.

Sometimes, subtype specific details appear in separate reports, and the titles of these reports may then correspond to subtypes or to subtype components (disjuncts or conjuncts).

Table 6.6 An output report from the media survey

Person	*Age (y)*	*Television (h/week)*	*Newspaper (h/week)*	*Favorite channel*	*Favorite paper*	*Preferred news*
5001	41	0	10	–	The Times	–
5002	60	0	25	–	The Times	–
5003	16	20	2	9	The Times	–
5004	18	20	5	2	Daily Mail	TV
5005	13	35	0	7	–	–
5006	17	14	4	9	Daily Sun	–
5007	50	8	10	2	Daily Sun	NP
5008	33	0	0	–	–	–
5009	13	50	0	10	–	–

Male Patients:

PatientNr	*Name*	*Phone*	*Prostate status*
101	Adams A	2052061	OK
104	King P	?	benign enlargement
105	Smith J	2057654	?

Female Patients:

PatientNr	*Name*	*Phone*	*Pregnancies*
102	Blossom F	3652999	5
103	Jones E	?	0

Figure 6.54 Patient data presented in subtype-specific reports.

As a simple example, the hospital patient data discussed earlier in Table 6.4 might be presented in two tabular reports as shown in Figure 6.54. A more complex example of subtype-specific reports is included in the exercises.

To help you do the section exercises without a domain expert, the data provided are significant with respect to the subtype graph. However, this is an artificial situation. In designing real world applications, you would rarely be handed data guaranteed to be significant in this way. Only the domain expert could provide this guarantee, and to do this he or she must know the subtyping constraints anyway. It is better to get these constraints directly from the domain expert simply by asking.

Apart from the methods discussed, an online appendix discusses two matrix algorithms to deduce the configuration of a subtype graph from a population significant in this regard. Although it is unsafe to assume that populations are significant, such procedures can at least be used to check whether a population is significant with respect to a known subtype graph.

Exercise 6.5

1. (a) For each of the following subtype graphs assume that $A = \{1, 2, 3\}$. Provide populations for the subtypes to complete a satisfying model for each diagram.

(b) Explain what is wrong with each of the following subtype graphs.

Note: *In the following questions, populations are significant with respect to the subtype graph, and "–" means "not to be recorded".*

2. The following table is a sample weekly report about animals in a certain household. Provide a conceptual schema for this UoD, including uniqueness constraints, mandatory roles, value constraints, and subtype constraints. Include a definition for each subtype.

Animal	*Animal kind*	*Sex*	*Nr cars chased*	*Nr mice caught*
Fido	dog	M	12	–
Felix	cat	M	–	1
Fluffy	dog	F	0	–
Tweetie	bird	F	–	–

3. A simplified fragment of an income tax return form is shown.
   (a) Schematize this UoD, assuming that in section 5 the same employer can appear on only one line.
   (b) Modify your solution to part (a) to handle these changes. The same employer may appear on more than one line of section 5. For each of these lines, the taxpayer attaches a wage and tax statement from his or her employer that includes the employer's name as well as the employee's tax file number, gross salary and tax installments deducted (i.e., tax paid). Wage and tax statements are identified by serial numbers.

1. Your Tax File Number: .....

2. Your full name: .....

3. Indicate the kind(s) of income you received from an employer by checking the relevant box(es).
   If you had no employer income, skip this and go to question 7.

   ❑ salary ❑ benefit (employment related)

4. Your main employment occupation: .....

5. If you earned a salary, complete the following income details:

Employer	*Tax paid*	*Gross salary*

6. If you earned employment related benefits indicate total benefit: .....

7. etc.

4. Schematize the following table. Here a supervisor supervises one or more persons. You may find it helpful to draw a tree showing who supervises whom. The missing value "–" means "inapplicable because of some other information". The missing value "---" means "does not exist".

Emp Nr	*Emp name*	*Supervisor*	*Smoker?*	*Home phone*	*Health bonus ($)*	*Company car*
101	Brown CA	102	N	–	300	–
102	Jones E	103	Y	2053771	–	–
103	White TA	---	N	2062050	500	597VTP
104	Mifune K	103	N	3651000	500	123KYO
105	Nelson HF	102	N	–	300	–
106	Adams PC	102	N	3719872	400	321ABC
107	Wong S	104	N	–	300	–
108	Collins T	104	Y	–	–	–
109	Smith JB	106	Y	–	–	–

5. A loan agency records information about clients borrowing money. The following report is extracted from this information system. Clients are identified by clientNr. Marital codes are "D" (divorced); "M" (married); "S" (single); "W" (widowed). Residential codes are: "B" (home buyer); "O" (home owner); "R" (home renter).

ClientNr	*Marital status*	*Residential status*	*Number of dependants*	*Home Value ($)*	*Spouse's income ($)*	*Spouse's share of home (%)*
103	M	O	3	100,000	40,000	50
220	M	R	3	–	0	–
345	S	O	–	150,000	–	–
444	W	B	2	90,000	–	–
502	D	R	0	–	–	–
600	S	R	–	–	–	–
777	M	B	0	100,000	0	40
803	D	B	1	60,000	–	–
905	W	O	0	90,000	–	–

Each client has borrowed one or more loans. For this agency, any given loan can be borrowed by only one client. The following report from the same information system provides details on some of the loans. Schematize this UoD.

ClientNr	*Loans borrowed*		
	LoanNr	*Amount $)*	*Term (years)*
103	00508	25000	7
220	00651	25000	7
	00652	3000	1
etc.			

6. The following three reports come from an information system about these bodies in our solar system: the Sun, the naked eye planets, and their moons. Planets travel in roughly circular orbits about the Sun. A planet in inferior conjunction is lined up directly between the earth and the Sun. A planet is in opposition if the Earth is lined up directly between it and the Sun. The missing value "---" means "does not exist", and "M_E" means one Earth mass. An astronomical unit (a.u.) equals 150,000,000 km. Schematize this.

Sun:

Name	*Mass (M_E)*	*Radius (km)*
The Sun	34 000	696 000

Planets:

Name	*Nr of moons*	*Mean distance from Sun (a.u.)*	*Mass (M_E)*	*Radius (km)*	*Orbital period (y d)*	*Next inf. conj.*	*Next oppos.*	*Atmos-phere (main gases)*
Mercury	0	0.39	0.06	2 440	0y 88d	Dec	–	---
Venus	0	0.72	0.81	6 050	0y 224d	Jan	–	CO_2
Earth	1	1.0	1.0	6 378	1y 0d	–	–	N_2,O_2
Mars	2	1.5	0.11	3 095	1y 322d	–	Feb	CO_2
Jupiter	63	5.2	318.0	71 400	11y 315d	–	Sep	H_2,He
Saturn	60	9.5	95.0	60 000	29y 167d	–	Sep	H_2,He

Moons:

Name	*Planet orbited*	*Radius (km)*	*Orbital period (y d)*	*Mean apparent magnitude*
Luna	Earth	1737	0y 27.3d	-13.0
Phobos	Mars	6	0y 0.3d	11.5
Deimos	Mars	4	0y 30.0d	12.0
Io	Jupiter	1867	0y 1.7d	5.5
etc.				

6.6 Generalization of Object Types

The previous section discussed a specialization procedure in which subtypes of a more general object type are introduced to declare that specific roles are recorded only for these subtypes. Apart from this top-down procedure, a subtype graph may also arise in a bottom-up way when we need to introduce a supertype of object types that already occur in the model.

The process of introducing a supertype for object types that already exist is known as object type *generalization*. Hence generalization is the reverse of specialization. A supertype is a more general form of its subtypes, and a subtype is a special form of its supertype(s).

Figure 6.55 A schema before generalization has been applied.

Step 3 of the CSDP outlined the main reasons for generalizing existing object types (e.g., Doctor, Physiotherapist, Pharmacist) into a more general type (e.g., Practitioner). At that stage, we ignored any need to retain an original object type as a subtype of the new general type. This need arises when the original object types have specific roles. As an example, consider Figure 6.55. Here name and gender are recorded for each kind of practitioner. In this case, a practitioner type is already suggested by the common identification scheme (practitionerId), but this type has not yet been introduced.

If we want to list all the practitioners together in some query then we should introduce a Practitioner supertype, attach the common roles to it, and add a fact type to retain the classification scheme (see Figure 6.56). This assumes pharmacists, physiotherapists, and doctors are mutually exclusive. If a global identifier for the supertype does not already exist, (e.g., practionerId), then we should introduce one.

The original schema shows that prescriptions may be prescribed only by doctors, and dispensed only by pharmacists. These constraints are retained in the new schema by using Doctor and Pharmacist subtypes for the prescription predicate. Here, physiotherapists have no specific role, so we don't introduce a subtype for them.

Figure 6.56 The schema after generalization has been applied.

Doctor, pharmacist, and physiotherapist are mutually exclusive, but were generalized partly because of their common roles (having a gender and a name). In practice, further common information would normally be recorded (e.g., phone number, address, birth date), making this generalization more worthwhile.

However, as pointed out in Step 3, the sharing of a common role is not by itself sufficient to justify generalization. *If the object types are exclusive, we needn't generalize them unless we want to list common details for them in the same query.*

For our medical clinic example we usually would want to issue queries such as "List the name, phone number, and address of all the practitioners". Generalization facilitates this, if we provide a uniform identification scheme for all the practitioners.

To clarify this point, consider a veterinary clinic that records the gender and name of its employees, as well as the gender and name of the animals treated in the clinic. Employees are identified by employee numbers, but animals are identified by some other scheme (e.g., combining their name with the client number of their owner).

It is highly unlikely that we would ever list the same information about the staff and the animals together in the same query. So Employee and Animal may be left as primitive object types, without generalizing them to EmployeeOrAnimal.

As a related example, consider the schema of Figure 6.57. Here an organization records the cars owned by its employees (e.g., to help the car park attendant check whether a car is parked legally). It also records equipment (e.g., computers, photocopiers) owned by its departments.

Cars are identified by their registration number, and equipment items by an item number. An ownership predicate appears twice in the schema. Employee and Department are exclusive, but both play the role of owning. Car and Equipment are exclusive, but both play the role of being owned.

Suppose we generalize Employee and Department to Owner, and Car and Equipment to Asset. As an exercise, you might like to draw the resulting schema for yourself before peeking at the solution.

The result of this generalization is shown in Figure 6.58. It's quite messy. The ownership predicate now appears just once, in the fact type Owner owns/is owned by Asset. But the rest of the schema is harder to conceptualize. New identifiers are required for the supertypes, and predicates to classify owners and assets by OwnerKind and AssetKind are added to retain the original type information.

Figure 6.57 Should the object types be generalized?

Figure 6.58 Don't generalize the types in Figure 6.57 like this unless you need to.

This solution assumes that CarRegNr and EquipmentItemNr are exclusive and compatible, so may be unioned into AssetNr as a global identifier for assets. If this is not the case, a new global AssetNr is introduced, and Equipment and Car subtypes are introduced to preserve their old identifiers (assuming these are still needed).

It would be wrong to model the working predicate as Owner works for Owner. Because of their specific roles (working, employing), Employee and Department are retained as subtypes, so we still have the fact type Employee works for Department. Subtype definitions are added as shown.

As the subtypes inherit the ownerNr identifier, the employee number and department name predicates must now be explicitly depicted. Additional textual constraints declare that each equipment item is owned by at most one Department (note the different constraints on the original ownership predicates), that only employees own cars, and that only departments own equipment.

Should we generalize exclusive object types in such a case? Not unless we want to ask a common question for all members of the general type (e.g., List all owners and their assets). If the objects are not to be listed together in the same column of an output report, then there is no need to generalize them. The more common properties there are, the more likely it is that such queries would be formulated.

For example, in many applications, such as banking or insurance, a customer might be a person or a company. Because common information needs to be accessed for them in the same query, it is worthwhile generalizing Person and Company to Customer (or Party, or whatever term is appropriate for the application). Of course we still retain Person and Company as subtypes to record details specific to them.

If two fact types have the same predicate reading and object type name sequence, they are treated as the same fact type. However, different predicates can have the same

name. This often happens with trivial predicates such as "has' and "is of". It can also happen if we don't generalize object types that share common roles. For example, in Figure 6.57 there are two predicates called "owns". We can distinguish between these predicates by their containing fact types (e.g., Employee owns Car differs from Department owns Equipment). In a CASE tool, predicates with the same reading may be distinguished internally by surrogates or expanded names (e.g., P56, P57, or "owns1", "owns2"). Apart from obvious cases like "has" and "is of", however, if we duplicate a predicate reading on a schema we should ask ourselves whether we should generalize.

The examples so far have exclusive subtypes. What about *overlapping types*? Recall that primitive entity types on a conceptual schema are mutually exclusive. The only way to allow different entity types to overlap is to make them subtypes. So generally speaking, if we find two primitive entity types that do overlap, then we should introduce a supertype for them. In rare cases however, we might portray entity types that overlap in reality, to be mutually exclusive in our model, because we are simply not interested in knowing about any overlap.

Consider the schema shown in Figure 6.59. This might be part of a university schema. Students are identified by their student number, and university employees are identified by an employee number. Employees are classified as academics, administrators, or general staff. Only academics can teach subjects, and only students can study subjects. Apart from their specific roles, students and employees both have their gender and birthdate recorded. In the global schema, other specific details would be recorded for students (e.g., enrollment mode) and employees (e.g., department), and other common properties would be recorded (e.g., name, address, phone number).

If you cut Figure 6.59 vertically down the middle, but duplicate Gender, Date, and Subject in both, the left-hand subschema deals with student records, and the right-hand subschema deals with employee records. In the past, it was common practice to develop such subsystems independently of one another. Nowadays many organizations are integrating subsystems into a general system to provide access to more information and to enforce constraints between the formally separate schemas.

Figure 6.59 Students and employees have some details in common.

Suppose Student and Employee are mutually exclusive. Despite their common roles, it is unlikely that we want to list such information for both students and employees in the same query. In this case we may leave the schema as it is.

Now suppose that in reality Student and Employee overlap (i.e., a student may be an employee). If we want to model this possibility, we must introduce a supertype (e.g., Person). But if we are not interested in knowing whether a student is an employee, and we have no common question for the two types, then we may leave the schema as is. However, our model now differs in this respect from the real world.

By presenting Student and Employee as primitive, our model declares that these types are exclusive, whereas in reality they overlap. To avoid misunderstanding, any such disagreement between the model and reality must be clearly documented in technical and user manuals so that people are adequately warned of this decision.

Such decisions should not be taken lightly, as there will often be good reasons for wanting to know whether some instance is a member of both types. For example, suppose we want to enforce the constraint that no academic can be enrolled in a subject he or she is teaching. If we want the information system to enforce this constraint for us, we have to provide it with a way of detecting whether some student is an employee. To do this we must introduce a supertype (e.g., Person), preferably with a global identification scheme (e.g., personNr).

Moreover, we might want to record specific data for student employees (e.g., study leave). In this case, we also need to introduce StudentEmployee as a subtype of both Student and Employee. Figure 6.60 depicts the subtype graph for this situation. As an exercise, expand this graph to the full schema by adding fact types and subtype definitions. The gender and birthdate fact types attach to Person. If included, the constraint about teachers should be specified textually.

In previous cases, all subtypes in a subtype graph have inherited the identification scheme of their top supertype. For example, in Figure 6.60(a), all people, whether or not they are students and/or employees, are always identified by person numbers.

In practice, however, we may sometimes meet a subtype with a preferred identification scheme that differs from that of its direct supertype(s). In this case we have a *context-dependent reference scheme*, since the preferred identifier of an object depends on the context in which it is being considered.

Figure 6.60 The subtype graph after generalization and further specialization.

Suppose that within the context of student records we want to identify students by studentNr, and in the context of employee records we want to identify employees by employeeNr. We can indicate this choice on the subtype graph by introducing the context-dependent identifiers on the subtype nodes themselves.

For example, in Figure 6.60(b), Student is identified by StudentNr, whereas Employee is identified by EmployeeNr. If a subtype does not introduce its preferred identifier, it must inherit its identifier from one of its supertypes (either direct or indirect).

A *solid subtyping arrow* provides a path to the subtype's preferred identifier, using the first identifier found on the path. A *dashed subtyping arrow* does not provide a path to the subtype's preferred identifier, so the subtype either introduces its own identifier or must inherit its identifier via another path to a supertype.

In Figure 6.60(b), for example, Student and Employee introduce their own identifiers so they have only a dashed subtyping connection to Person. StudentEmployee has two supertypes, and is identified by EmployeeNr (shown by the solid subtyping connection) rather than by StudentNr (dashed connection).

The procedure for generalizing object types may now be summarized as follows. Here *A* and *B* are entity types which are initially depicted separately, with no common supertype.

Generalization Procedure (GP):

if *A* and *B* overlap, or can be compared (e.g., to compute ratios)
and we wish to model this
or
A and *B* are mutually exclusive
and common information is recorded for both
and we wish to list *A* and *B* together for this information

then introduce their supertype $A \cup B$ with its own identification scheme;
add predicate(s) to the supertype to classify members into *A*, *B*;
attach common roles to the supertype;
if *A* (or *B*) plays some specific roles
then define it as a subtype and attach these roles

It should be emphasized that the decision to generalize or specialize is a conceptual one. This does not commit us to a particular way of mapping the resulting subtype graph to a logical database schema. A number of mapping options exist (e.g., subtypes may be absorbed into the same table as their supertype, or mapped to separate tables). Chapter 11 discusses these mapping options in detail.

Exercise 6.6

1. A taxi company records these details about its employees: employeeNr, name, address, gender, phone, salary, and weight. It records these details about its cars: regNr, model, year manufactured, cost, and weight. It also records who drives what cars (optional *m:n*).
   (a) Schematize this with no subtyping. Use two predicates with the name "weighs".
   (b) Generalize Employee and Car, so that "weighs" appears only once.
   (c) Which of these solutions is preferable? Discuss.

2. A hospital maintains an information system about its employees and patients. The following report is extracted from its **employee records**. Employees are standardly identified by their employee number. Each employee has exactly one job: administrator (admin); doctor (doc); or pharmacist (pharm). A pager is a portable electronic device that beeps when its number is rung. Some employees are assigned unique initials, which may be used as a secondary identifier. The mark "–" means "inapplicable because of other data".

EmpNr	*Name*	*Job*	*Office*	*PagerNr*	*Initials*
e10	Adams A	admin	G17	–	–
e20	Watson M	doc	302	5333	MW1
e30	Jones E	pharm	–	–	EJ
e40	Kent C	admin	G17	–	–
e50	Kildare J	doc	315	5400	JK
e60	Brown C	pharm	–	–	CB
e70	Collins T	pharm	–	–	TC
e80	Watson M	doc	315	5511	MW2

In the following extract from the **patient records**, patients are standardly identified by their patient number (patNr) and are typed as inpatients (in) or outpatients (out), but not both. Some patients are placed in wards. Patients may be allergic to various drugs. Prescriptions are identified by their scriptNr. Each dispensed prescription has been prescribed by a doctor, dispensed by a pharmacist, and issued to a patient. Each prescription specifies exactly one drug. Since initials are easier for humans to remember, they are used in this report; however, the system uses empNr as the primary identifier for employees. Unlike "–", a blank simply means "not recorded".

The population is significant with respect to mandatory roles and the subtype graph. Note, however, that any patient may be dispensed a prescription, any doctor may prescribe one, and any pharmacist may dispense one.

				Prescriptions dispensed to patients			
PatNr	*Type*	*Ward*	*Allergies*	*Script*	*Prescriber*	*Dispenser*	*Drug*
p511	in	5B	aspirin doxepin	7001	MW1	EJ	warfarin
p632	out	–		7132 7250	JK MW1	CB EJ	aspirin paracetamol
p760	in	4C	warfarin	8055	JK	EJ	aspirin
p874	in	5B					

(a) In the real world, *no patient can be an employee*. Schematize this UoD.
(b) In reality, a patient may be an employee, but we are not interested in knowing such facts. What changes, if any, should be made to the model or system documentation?
(c) In reality, a patient may be an employee. Moreover, the hospital requires such facts to be known. For example, we may want to ensure that no doctor prescribes a drug for himself/herself. The hospital is willing to replace its old identification schemes for employees and patients by a new one. Modify your schema to deal with this situation.
(d) In reality, a patient may be an employee, and we want to know this. However the hospital demands that empNr and patNr be retained as preferred identifiers within their context. Modify your schema to deal with this situation.

6.7 Summary

A *set* is determined by its members, so order and repetition don't matter, e.g., {3, 6} = {6, 3, 6}. A *bag* or *multiset* makes repetition significant, e.g., [3, 6] = [6, 3] but [3, 6] ≠ [3, 6, 6]. A *sequence* is an ordered bag, e.g., (3, 6) ≠ (6, 3). A set *A* is a *subset* of *B* (written $A \subseteq B$) iff each member of *A* belongs to *B*; in this case *B* is a *superset* of *A*. *A* is a *proper subset* of *B* iff $A \subseteq B$ and $A \neq B$. *A* and *B* are *mutually exclusive* or disjoint iff they have no common members, i.e., their *intersection*, $A \cap B = \{\ \}$. If sets have common members they *overlap*; if each also has extra members, we have a case of *proper overlap*.

CSDP step 6 adds any value, set-comparison (subset, equality, exclusion), and subtype constraints. A *value constraint* specifies the possible values for a value type or role. It may provide a *full listing* or *enumeration* of all the values, e.g., {'M', 'F'}, a *subrange definition,* e.g., {1..7}, {'A'..'E'}, or a combination of lists and ranges.

A value constraint may also indicate a syntactic data type (e.g., variable length text of at most 20 Unicode characters). If the underlying data type is real number (e.g., float or decimal), a square bracket/parenthesis signifies inclusion/exclusion of the end value. For example, "(0..100]" denotes the positive real numbers up to and including 100.

An *independent object type* is a primitive object type whose instances may exist independently of playing any fact roles (the fact roles are collectively optional). Independent object types have "!" appended to their name. For example, we may populate a reference table for "Country!" with the names of all existing countries, even if some of these countries do not play in any elementary facts. In principle, the values that reference members of an independent object type could instead be declared in a value constraint; however, this is awkward if the values may change or there are many values that cannot be specified as a range.

Set-comparison constraints restrict the way the population of a role, or role sequence, relates to the population of another. Let *rs1* and *rs2* be role sequences (of one or more roles) played by compatible object types. A *subset constraint* from *rs1* to *rs2* is denoted by an arrow in that direction, including a circled "⊆" symbol. This indicates pop(*rs1*) ⊆ pop(*rs2*).

An *equality constraint* is equivalent to subset constraints in both directions. It is shown by a dotted line with a circled "=" symbol, demanding that pop(*rs1*) = pop(*rs2*).

An *exclusion constraint* among two or more role sequences is shown by connecting them to "⊗" with dotted lines: this means their populations must be disjoint (mutually exclusive).

If a set of roles is disjunctively mandatory and mutually exclusive, this may be shown by the lifebuoy symbol ⊗. This *exclusive-or constraint* symbol is simply the inclusive-or constraint symbol ⊙ overlaid on the exclusion constraint ⊗.

If each role sequence contains two roles, we talk of *pair*-subset, pair-equality, and pair-exclusion constraints. A tuple-subset constraint implies simple subset constraints. A simple exclusion constraint implies tuple exclusion. Theorem NXS states that no exclusion constraint can have exactly the same arguments as a subset constraint.

An object type A is a (proper) subtype of B iff ($A \neq B$ and) for each database state, $\text{pop}(A) \subseteq \text{pop}(B)$. We show this by an *arrow* from A to B. If A is a subtype of B, and B is a subtype of C, then it is transitively implied that A is a subtype of C; such indirect subtype links should not normally be displayed.

An object type may have many subtypes and many supertypes. Subtype connections among a family of compatible object types are displayed in a directed, acyclic *subtype graph*. This graph has exactly one root node (or top), which must be a primitive object type (or, in rare cases, an objectified association).

In ORM, subtypes are typically introduced to declare that one or more roles are played only by that subtype. The process of introducing subtypes is called *specialization*. Subtypes inherit all the roles of their supertype(s) and normally have at least one specific role. By default, a subtype inherits the preferred reference scheme of a direct supertype; in this case the reference scheme is not displayed on the subtype. With subtypes (and objectified associations), any mandatory roles must be explicitly shown.

A subtype may be derived, asserted, or semiderived. Each *derived subtype* must be formally defined in an equivalence derivation rule using one or more roles of its supertype(s). For example, **Each** FemalePerson **is a** Person **who** is of Gender 'F'. Derived subtypes are marked "*". With derived subtypes, all relevant constraints on defining predicates must be declared; this ensures that any subtype exclusion "⊗" or totality "⊙" constraints that do exist are implied. Exclusive object types (e.g., Man, Woman) cannot have a common subtype. *Asserted subtypes* are simply asserted without any derivation rule. *Semiderived subtypes* may have both derived and asserted instances; these rare subtypes are marked "$^{+}$" and have conditional derivation rules.

Output reports often have missing values or nulls. Subtyping is indicated if a missing value means "*not to be recorded*" (i.e., inapplicable because of other recorded data). This book uses a minus sign "–" for this purpose. If an output report is significant with respect to the subtype graph, we can determine the graph by examining the pattern of "–" marks. However, background knowledge is required to determine the subtype definitions, and hence meaningful subtype names. Input forms often provide a set of conditional instructions that indicate the conditions under which particular entries on the form are needed. Such instructions can be used to deduce the subtyping constraints. Sometimes reports are already separated on the basis of subtyping and their report titles give clues as to the subtyping scheme.

Without a full set of output reports, we might wrongly declare object types to be primitive. We then need to introduce supertypes; this process is called object type *generalization*, the reverse of specialization. We introduce a supertype if object types overlap, and we want to know this. A supertype is also introduced if the object types are exclusive, but have common roles, and we want to list them together for this common information. If the original object types have specific roles, they must be retained as subtypes.

In some cases there is a need for subtypes to have a preferred reference scheme different from that of their supertype(s). Such a *context-dependent reference* is indicated by displaying the preferred reference scheme on the relevant subtypes. A subtyping arrow appears as a *solid arrow* if it provides a path to the subtype's preferred identifier, otherwise it is shown as a *dashed* arrow.

Chapter Notes

Not all versions of ORM allow independent entity types. FCO-IM (Bakema et al., 1994) handles the problem by treating entity types as objectifications of unaries.

Most versions of ER do not allow subset, equality, or exclusion constraints between role-sequences, or attributes, to be included on the schema diagram (see Chapter 8). UML includes graphic symbols for subset constraints only between whole associations, and exclusion constraints only in the context of an exclusive-or constraint between roles (see Chapter 9). Recent versions of ER, as well as UML, offer basic support for subtyping, but ignore the need for formal subtype definitions or context-dependent reference (see Chapters 8 and 9).

Some versions of ORM restrict subtypes to derived, active subtypes. Context-dependent reference is not supported in some versions. The PSM variant adopts a different approach to generalization in which the supertype uses a disjunction of the identification schemes of its subtypes (e.g., see ter Hofstede et al. 1993). Although this raises additional implementation problems when a common query is executed on overlapping subtypes, this does provide another viable approach.

For early discussion of subtyping from an ORM perspective, an informal treatment can be found in Halpin (1995) and a formal treatment in Halpin and Proper (1995b). A recent treatment updated for ORM 2 is provided in Halpin (2007a); this paper also distinguishes rigid subtypes from role subtypes, a distinction examined in Chapter 10.

7 Other Constraints and Final Checks

7.1 Introduction to CSDP Step 6
- Overview

7.2 Occurrence Frequencies
- Internal frequency constraints
- External frequency constraints

7.3 Ring Constraints
- Logical definitions
- Graphical notations

7.4 Other Constraints and Rules
- Value-comparison constraints
- Type cardinality constraints
- Textual constraints
- Derivation rules

7.5 Final Checks
- Consistency (internal and external)
- Redundancy
- Completeness

7.6 Summary

7.1 Introduction to CSDP Step 7

In previous steps of the CSDP, we verbalized familiar examples in terms of elementary facts, sketched the fact types on a diagram, and then added various constraints and derivation rules. Most applications include uniqueness, mandatory role, value, set-comparison, and subtyping constraints. Although less common, other kinds of constraint may also apply. Some of these have graphic notations. The rest are specified textually. Once all the constraints are specified, some final checks can be made to help ensure that our information model is consistent and free of redundancy. This concludes the basic conceptual schema design procedure.

CSDP step 7: Add other constraints, and perform final checks

This chapter deals with step 7 in detail. The next three sections discuss some important graphic constraints (occurrence frequencies, ring constraints, cardinality constraints, and value-comparison constraints). After that, we consider some constraints that must be declared textually. Finally we examine some ways of checking that the schema is consistent and redundancy free.

7.2 Occurrence Frequencies

Let n be some positive integer. To indicate that each entry in a fact column must occur there *exactly n times*, the number n is written next to the role (see Figure 7.1). This is an example of a **frequency constraint** (also called an occurrence frequency constraint). If $n = 1$, this is equivalent to a uniqueness constraint—in this case the usual bar notation for uniqueness should be used instead of a "1" mark.

Like uniqueness constraints, frequency constraints are *local* constraints on the role(s), not global constraints on the object type. The constraint in Figure 7.1 means that for each state of the database, *if* an instance of *A* plays role *r*, it must do so *n* times. It does *not* mean that each instance of *A* must play that role *n* times. If the role is optional, some instances of *A* might not play the role at all.

One use of occurrence frequencies is to ensure that if details are recorded then they must be recorded for all instances of an enumerated type. Suppose a company stocks and sells three kinds of computer drives (C = CD drive, D = DVD drive, H = hard drive) and operates in just two cities (Sydney and Tokyo). Now consider the populated conceptual schema shown in Figure 7.2. The UC on the first two roles ensures that each city–drivekind pair appears on only one row. The frequency constraint of 3 on the first role requires each city that occurs in the first column to do so three times.

Each member of pop(*r*) occurs there exactly *n* times.

Figure 7.1 A simple occurrence frequency constraint on role *r*.

Figure 7.2 If stock figures are recorded, they are recorded for each city and drive kind.

Hence if any stock figures are recorded for a city, then figures for all three kinds of drive must be recorded. Similarly, the frequency constraint of 2 on the second role ensures that each drive kind in the second column must appear there twice. Because of the uniqueness constraint, this means that if stock is recorded for a drive kind, then figures for both cities must be included.

The combination of frequency, uniqueness, and value constraints ensures that any recorded stock figures are *complete* with respect to both cities and drive kinds. A compound transaction is needed to initially populate this fact type, requiring at least six facts to be added.

A frequency constraint that spans two or more roles of a fact type is a *compound frequency constraint*. In this case, we link the relevant roles by a line and connect the frequency to this role link by a dotted line (this dotted line may be hidden if one moves the frequency number close enough to the constrained roles). If there are only two constrained roles and these are adjacent, the frequency may be attached by a dotted line to the junction of those roles. For example, in Figure 7.3 each city–year pair in the fact table must occur there three times. Given the uniqueness constraint across the first three roles, sales must be recorded for all three drives for each city–year pair in the table. Note that this does not require that yearly sales be recorded for both cities. The sample population includes only one city, yet it satisfies all the constraints.

Figure 7.3 Yearly sales figures may omit a city but not a drive kind.

Sydney	2007	C	10000
Sydney	2007	D	7000
Sydney	2007	H	2000
Tokyo	2007	C	7000
Tokyo	2007	D	5000
Tokyo	2007	H	1000

Figure 7.4 Yearly sales figures must cover all cities and drive kinds.

Figure 7.4 strengthens the constraints by requiring yearly sales figures for both cities as well as the three kinds of drive. Given the uniqueness and value constraints, this is achieved by the simple occurrence frequency of 6 on the year role—as an easy exercise, prove this. The compound frequency of 3 across city–year is omitted since it is implied. A compound frequency of 2 across year–drivekind is also implied, and hence omitted.

Frequency ranges may also be specified. Use "≤ *n*" for *at most n*, where $n \geq 2$. Use "≥ *n*" for *at least n*, where $n \geq 1$. Use "*n* .. *m*" for *at least n and at most m*, where $2 \leq n < m$. A lower frequency of 1 is the smallest frequency allowed and is assumed by default, since each entry in a column is already there once. Because the constraint applies to the population of the role(s), not the population of the object type(s), it doesn't make sense to declare an occurrence frequency of zero. An optional role indicates that some members of the object type population might not play that role.

As an example, consider Figure 7.5. This is part of a schema for a conference application. Each expert is on at least one panel (as shown by the mandatory role) and reviews at most five papers (possibly none, since the role is optional). Each panel with members has at least four and at most seven members. Since Panel is independent, we may record the existence of a panel before assigning members to it. Each reviewed paper has at least two reviewers. It is possible that some papers have not been assigned reviewers yet. In principle, frequency constraints may combine values and ranges (e.g., "1, 5..7") but in practice this is extremely rare.

Figure 7.5 Examples of minimum and maximum frequency constraints.

Figure 7.6 Example of an external frequency constraint.

Just as internal uniqueness constraints can be generalized to frequency constraints, external uniqueness constraints can be generalized to *external frequency constraints*, which apply to roles from different predicates. For example, at least some universities allow students to enroll in the same course at most twice (if they can't pass the first time, they are allowed only one more attempt). This external frequency constraint is displayed by connecting a circled frequency indicator to the relevant roles, as shown in Figure 7.6. Within the context of these fact types, each student–course combination is associated with at most two enrollments.

Exercise 7.2

1. The following annual report provides information about two kinds of DBMS (R = Relational, NR = Non-Relational) for each of the four seasons. Schematize this UoD, including uniqueness, mandatory role, value, and frequency constraints.

Database	*Season*	*Nr sold*
R	spring	50
NR	spring	70
R	summer	60
NR	summer	60
R	autumn	80
NR	autumn	40
R	winter	120
NR	winter	15

2. A software retailer maintains a database about various software products. Two sample output reports are shown. There is a sales tax of 20% on all software. The *ex-tax* price excludes this tax; the *with-tax* price includes it. The functions of a software product are the tasks it can perform. A "✓" indicates the rating of the product for the criterion listed on that row (e.g., both products shown have a good performance). A "?" denotes missing information.

   Schematize this UoD, including uniqueness, mandatory role, value, and frequency constraints and any derivation rules. Don't nest or subtype. Model the product evaluation ratings (for performance, etc.) in terms of a single fact type.

Product: WordLight 4.0 **List price**: *ex tax* *with tax*
Functions: word processor $500 $600

	Poor	*OK*	*Good*	*Excellent*
Performance			✓	
Documentation				✓
Ease of learning			✓	
Ease of use			✓	
Error handling				✓
Support			✓	
Value				✓

Release date: 2007 Feb
Next upgrade: ?

Product: PCjobs 1.0 **List price**: *ex tax* *with tax*
Functions: word processor $1000 $1200
spreadsheet
database

	Poor	*OK*	*Good*	*Excellent*
Performance			✓	
Documentation	✓			
Ease of learning		✓		
Ease of use	✓			
Error handling		✓		
Support			✓	
Value		✓		

Release date: 2007 Oct
Next upgrade: 2009 Jun

3. In Megasoft corporation, at the start of each year, each salesperson is assigned at most three different software products to sell during that year (not necessarily the same each year). At the end of each year, each salesperson reports how many of each product he/she sold in each month of that year. Historical data is kept on who is assigned what to sell in each year and the monthly sales figures for that person (if the figures are available). No month may be omitted in these annual sales figures. The system retains information from previous years. Schematize this, including value and frequency constraints. Use nesting.

4. (Acknowledgment: This question is based on an exercise devised by Dr. E. Falkenberg and is used by permission.)

   The following tables are sample reports from a system that maintains information about communities and roads. A "–" mark means "inapplicable because of other data" (see the asterisked notes). Mayors are identified by the combination of their first name and surname. A person can be mayor of only one city, and each city has exactly one mayor. Each road connects *exactly two* communities. Schematize this UoD, including uniqueness, mandatory role, value, subtype and frequency constraints.

Community	*Population*	*Longitude* *	*Latitude* *	*Size*** (km^2)	*Mayor* ***	
					Firstname	*Surname*
Astraluna	900 000	–	–	145	Fred	Bloggs
Bradman	90 000	+120°50′	+48°45′	12	–	–
Cupidville	9 000	+120°50′	+48°40′	–	–	–

* Recorded only for villages and towns (at most 100 000 inhabitants).
** Recorded only for towns and cities (more than 10 000 inhabitants).
*** Recorded only for cities (more than 100 000 inhabitants).

Road Nr	*Road kind**	*Connection***		*Length (km)*	*Avg. travel time****	*Max. slope (%)*****
		Community	*Community*			
11000	f	Astraluna	Bradman	25	0 h 20 m	–
11500	h	Astraluna	Bradman	22	0 h 25 m	2
11561	p	Bradman	Cupidville	17	0 h 20 m	2
11562	d	Bradman	Cupidville	15	–	15

* f = freeway; h = highway; p = paved minor road; d = dirt road.
** All connections are two-way (so the order of the connection is irrelevant).
*** Recorded only for freeways, highways and paved minor roads.
**** Recorded only for dirt roads, paved minor roads and highways.

5. An award granting institute has a rule that any given person (identified by a personNr) may apply for the same award (identified by an awardNr) at most three times. For each award application (identified by an applicationNr), its status is recorded.
   (a) Model this using an external frequency constraint.
   (b) Try to model this without using an application number, instead objectifying "Person applied for Award" as "Application". Explain why this approach fails.

7.3 Ring Constraints

When two roles in a predicate are played by the same object type, the path from the object type through the role pair and back to the object type forms a "ring". If the roles are played by subtypes with a common supertype, the path from and back to the supertype also forms a ring. The role pair typically forms a binary predicate, but may be part of a longer predicate (see the shaded role pairs in Figure 7.7) or may even come from different predicates in a join path (join constraints are discussed in Chapter 10). A *ring constraint* may apply only to a pair of roles like this.

Figure 7.7 Ring constraints may apply to the shaded role pairs.

Figure 7.8 A ring binary (possibly embedded in a longer fact type).

Before discussing the various kinds of ring constraint, some standard definitions from the logic of relations are noted. Figure 7.8 is used as a template to discuss any ring binary (the predicate *R* might be obtained from a longer predicate by projecting on roles r_1 and r_2, and *A* might be a supertype of the object types playing r_1 and r_2). For each state of the database, pop(r_1,r_2) is a set of ordered pairs, and hence a relation, of type *R*. Derived as well as asserted relations are allowed.

The infix notation "*xRy*" (read "*x R*s *y*") is convenient for defining relational properties such as reflexivity, symmetry, and transitivity. Consider the populated fact type shown in Figure 7.9(a). The fact table or relation contains three facts. For simplicity, assume people in this UoD are identified by their first name. Note that, for this population, each person likes himself/herself. Suppose that in our UoD, this is true for all possible populations—each person who plays the role of liking or of being liked must like himself/herself. In this case, the Likes predicate is said to be *reflexive over its populations*. For the general case shown in Figure 7.8, we have

R is **reflexive over its populations** iff for each possible population of *R*,
xRx for each *x* in pop(r_1) ∪ pop(r_2)

If we picture an object as a black dot and the Likes association as an arrow, we can picture this reflexion as a relationship line starting and ending at the same object, as shown in Figure 7.9(b). One can imagine less happy universes. For example, if we are allowed to delete just the third row from Figure 7.9(a), then Bill doesn't like himself, and the relation would then not be reflexive.

Many important relations are reflexive (e.g., =, ≤, is parallel to, implies). A relation that is reflexive over all things is said to be "totally reflexive" (i.e., for all *x*, *xRx*). Of the four examples cited, only "=" is totally reflexive. For example, "implies" is defined for propositions but not for people (people do not imply people). When we say a relation is reflexive, we usually mean reflexive over its populations (not necessarily over everything in the universe).

Figure 7.9 In this world, the Likes relation is reflexive.

Figure 7.10 In this world, the Likes relation is symmetric.

Now consider a UoD where anybody who likes a person is also liked by that person. One satisfying population comprises the three facts: Ann likes Bill; Bill likes Bill; Bill likes Ann (see Figure 7.10 (a)). Here given any persons x and y, not necessarily distinct, if x likes y then y likes x. If this is always the case, we say that Likes is *symmetric*.

The dot–arrow diagram in Figure 7.10(b) conveys the idea that if the relationship holds in one direction, it also holds in the other direction. Using "→" for "implies", we may define this notion for any predicate R as follows:

R is **symmetric** iff for all x, y $\quad xRy \rightarrow yRx$

Now consider a UoD where, if one person likes a second, and the second person likes a third, then the first person must like the third. In this case we say that likes is *transitive*. The definition includes the case where the persons are the same. The populated schema in Figure 7.11(a) illustrates this property.

The dot–arrow diagram in Figure 7.11(b) indicates that if a chain of relationships exists, the relationship also applies from the first to the last member of the chain. In general, transitivity may be defined as follows. Here we use "&" for "and", and give "&" precedence over "→" (so that the & operation is evaluated first). Note that x, y, and z need not be distinct.

R is **transitive** iff for all x, y, z $\quad xRy \,\&\, yRz \rightarrow xRz$

Figure 7.11 In this world, the Likes relation is transitive.

is a parent of

Ann	Ann
Ann	Bill
Bill	Ann
Bill	Colin
Ann	Colin

Figure 7.12 What is wrong here?

We leave it as an exercise to prove that a relation that is both symmetric and transitive must also be reflexive over its population. A relation that is reflexive, symmetric, and transitive is called an "RST relation" or "equivalence relation". The classic example of this is the identity relation "=". The inequality relation "≠" is symmetric only, while the less than relation "<" is transitive only.

Relational properties such as reflexivity, symmetry, and transitivity might be thought of as constraints, since they limit the allowable relations. However, such properties are *positive* and are used more often to *derive* additional facts than constrain existing facts. We now turn to a study of *negative* relational properties such as irreflexivity and asymmetry. Given the existence of some facts, such negative properties imply the non-existence of certain other facts. Such properties are best handled as simple constraints. An example will help clarify things. Look at the populated parenthood fact type in Figure 7.12 and see if you can spot anything that seems wrong.

The population shown is consistent with the constraints specified. Parenthood is many:many, and each person has at most two parents. But as you no doubt noticed, some of the facts in the table are inconsistent with the real world concept of parenthood. We need to add more constraints so that the information system is able to reject such erroneous populations.

To begin with, the first fact (Ann is a parent of Ann) has to be rejected since nobody can be his/her own parent. We say that the parenthood relation is *irreflexive* and define this notion for any relation type *R* as follows. Here tilde "~" denotes the logical "**not**" (i.e., "it is not the case that").

R is **irreflexive** iff for all *x* $\sim xRx$

In the dot–arrow icon in Figure 7.13(b), the arrowed arc depicts the relationship from the object to itself, and the bar across it indicates that this is not allowed. A simplified version of this icon (with the arrow-tip dropped) is used in ORM 2 to depict the irreflexive constraint, with a dotted line connecting the symbol to the relevant roles (if the roles are contiguous, it connects at their junction), as shown in Figure 7.13(c). This constraint verbalizes as "**No** Person is a parent of **itself**". An alternative notation used in ORM 1 for the irreflexive constraint is "^{O}ir".

Figure 7.13 Parenthood is declared irreflexive. But there is still something wrong.

Figure 7.13(c) adds the irreflexive constraint and reduces the population accordingly. Note that "irreflexive" is stronger than "not reflexive". For example, if Likes is irreflexive, then nobody likes themselves. If some but not all people like themselves, then Likes is neither reflexive nor irreflexive.

Irreflexivity is an intrarow constraint (i.e., violation of the constraint by a row can be determined by examining that row only). It specifies that no entry may occur in both cells of the same row. This makes the constraint easy and inexpensive to enforce. Note that the parenthood relation is irreflexive but not exclusive. For example, Bill may appear both as a parent of Colin and as a child of Ann.

There are still some problems with Figure 7.13(c). If we accept the first row, we should reject the second. If Ann is a parent of Bill, then it cannot be true that Bill is a parent of Ann. The parenthood relation is *asymmetric*. In general, for any predicate *R*:

R is **asymmetric** iff for all *x, y* $xRy \rightarrow \sim yRx$

That is, if the first object *R*s the second, then the second cannot *R* the first. Here, if one person is a parent of another, then that other cannot be a parent of the first. To say that a relation is asymmetric is stronger than saying the relation is not symmetric (you may wish to prove this as an exercise). An intuitive icon for asymmetry is shown in Figure 7.14(b).

Figure 7.14 Parenthood and reporting are asymmetric relations.

Figure 7.15 Antisymmetry.

An asymmetric constraint is depicted graphically by a simplified version of this icon, without arrow-tips, connected by a dotted line to the constrained role pair, as shown in Figure 7.14(c). With this constraint enforced, the parenthood population is reduced as shown. Figure 7.14(d) shows another example of the asymmetry constraint, where subtyping is involved. An alternative constraint notation for asymmetry used in ORM 1 is "^{O}*as*".

The irreflexive constraint is omitted from Figure 7.14(c). Is this a mistake? No. Any relation that is asymmetric must be irreflexive. The proof is left as an exercise (*Hint: x* and *y* need not be distinct). The converse does not hold. Some irreflexive relations are not asymmetric (e.g., is a sister of). To avoid implied constraints, do not declare irreflexivity if you have already declared asymmetry.

If we remove the irreflexivity requirement we obtain the weaker property of antisymmetry. For example, ≤ and ⊆ are *antisymmetric* but not asymmetric. The intuitive icon in Figure 7.15(a) uses different colored nodes to indicate the objects are different. A simplified version of this icon, without arrow-tips, is used to depict an antisymmetric constraint, as shown in Figure 7.15(b). The equivalence at the bottom of Figure 7.15(b) indicates that asymmetry combines antisymmetry and irreflexivity.

R is **antisymmetric** iff for all *x, y* $\quad x \neq y \;\&\; xRy \rightarrow \sim yRx$

Before continuing with the parenthood example, let's look at some other examples to clarify the ideas discussed so far. Our first case deals with a system to record synonyms. A synonym for a word has roughly the same meaning as that word. A sample taken from a poorly designed book of synonyms is shown in Table 7.1.

Table 7.1 Extract from a poorly constructed table of synonyms.

Word	*Synonyms*
abate	diminish, lessen, reduce, decline
abbreviate	shorten, reduce, condense
:	:
condense	compress, consolidate, abridge
:	:
reduce	shorten, weaken, abate
:	:

Figure 7.16 Examples of (a) symmetric and irreflexive, and (b) asymmetric fact types.

Suppose we decide to store this information in the has-synonym binary shown in Figure 7.16(a). Is this reflexive? Although we might argue that each word is a synonym of itself, this is a trivial result that we are not usually interested in seeing. So, as far as this relation goes, let us agree that no word is its own synonym. This makes the relation irreflexive, as shown by the inner ring constraint.

If one word is a synonym of another, then the other is a synonym for it. So has-synonym is symmetric. But Table 7.1 violates this condition. For example, the pair ("abbreviate", "reduce") appears in this order only. If we look up the word "reduce" we won't see "abbreviate" as one of its synonyms. This is a defect of the synonym book to avoid in a computerized synonym system. How can we avoid this problem?

One solution is to add the constraint that the has-synonym predicate is symmetric. This is specified on the diagram using a simplified (no arrow-tips) version of the symmetry icon in Figure 7.10, shown as the outer ring constraint in Figure 7.16(a). The symmetric and irreflexive constraints are combined into one icon to save space. ORM 1 uses the alternative ^{O}sym notation for symmetry constraints.

Suppose the symmetric constraint is declared, the table is currently empty, and the following update is attempted: **add**: Word 'abbreviate' has synonym Word 'shorten'. A naive information processor would simply reject this for violating the symmetry constraint. A sophisticated processor might accept the fact and automatically add its converse: "shorten" has synonym "abbreviate". A consistent scheme would be adopted for the delete operation. Although this approach works, it doubles the size of the table, since each pair of synonyms is stored twice, once for each ordering.

To save space, we could store each pair in one order only. The post-synonym binary in Figure 7.16(b) has the second word alphabetically after the first. The pair ("abandon", "leave") is stored in this order only. The relation is then asymmetric, as indicated. The larger synonym relation can now be derived, using a derivation rule such as

Word_1 has synonym Word_2 **iff** Word_1 has post_synonym Word_2
or
Word_2 has post_synonym Word_1.

In SQL, the derived relation could be defined as a view on the base relation by using the **union** operator. For speed, both columns can be indexed. Users might perform updates on the base relation, but issue queries on the derived relation. Thus, we can

sometimes choose whether to capture an aspect of the UoD in terms of a database constraint or as a derivation rule. Here we distinguished the asserted and derived fact types by name. It is also possible to have the fact type semiderived.

What about transitivity of synonymy? This issue is complex. You may be familiar with the "bald-hairy paradox" based on the following premise. In all cases, if we add one hair to the head of a bald man, he is still bald (where is the paradox?). A similar slippery-slope argument argues against synonym transitivity (sequences of approximations finally lead to nonapproximations). Moreover, the same word may have different meanings, leading to different synonym groups. If we still want to retain some transitivity, we may include a derivation rule for synonym transitivity, constrained by a maximum length of the transitivity chain and relativized to group meanings.

Now consider the marriage fact type in Figure 7.17. The husband and wife roles are mutually exclusive. In contrast to an irreflexive constraint, which requires inequality between entries on the same row, an exclusion constraint applies between columns and hence is much stronger. If no entry may occur in both columns, we cannot have two rows of the form (a, b) and (b, a). So *exclusion implies asymmetry*. Moreover, *asymmetry implies irreflexivity*. Since they are implied, neither the asymmetric nor the irreflexive constraint should be added to the schema.

Now let's return to the populated parenthood model in Figure 7.18(a). There is still a problem with it. Can you spot it? Let us agree that incest cannot occur in our UoD (if it can, we instead declare a deontic rule, as discussed in Chapter 10). If Ann is a parent of Bill and Bill is a parent of Colin, then Ann cannot be a parent of Colin. The parenthood relation is *intransitive* (i.e., if one person is a parent of a second and the second person is parent of a third, then the first person cannot be parent of the third). In general, for any relation type *R*:

R is **intransitive** iff for all *x, y, z* $\quad xRy \;\&\; yRz \rightarrow \sim xRz$

Figure 7.18(b) shows an intuitive icon for intransitivity, with a bar across the forbidden jump. A simplified version of this icon (without arrow-tips) is used for the intransitive constraint symbol, as shown on the right in Figure 7.18(b). Figure 7.18(c) signifies that parenthood is both intransitive and asymmetric by combining the constraints in a single icon. The population is changed to satisfy the constraints.

Figure 7.17 The exclusion constraint implies asymmetry and hence irreflexivity.

Figure 7.18 The parenthood relation is asymmetric and intransitive.

As an exercise, prove that intransitivity implies irreflexivity (*Hint: x, y,* and *z* need not be distinct). The term "atransitive" is sometimes used instead of "intransitive". Saying a relation is intransitive is stronger than saying it is not transitive. Note that antisymmetry, asymmetry, and intransitivity are not intrarow constraints, since their enforcement requires comparing the row in question with other existing rows.

If an irreflexive relation includes a functional role (rather than being *m:n*) it must also be intransitive. Figure 7.19 illustrates the proof of this by *reductio ad absurdum*. Let *R* be an irreflexive relation whose first role is functional (i.e., it has a simple uniqueness constraint). Now populate *R* with the first two rows shown. Since R is irreflexive, $a \neq b$ and $b \neq c$. Assume that *R* is not intransitive. This allows us to add the third row shown. Since $b \neq c$, rows 1 and 3 are distinct rows. This means that *a* plays the first role of *R* twice. But this contradicts the uniqueness constraint. So row 3 must be rejected (shown here by the dotted line through the third row). Hence the relation must be intransitive.

Are we finished with ring constraints? Not quite. Have a look at the parenthood population in Figure 7.18(c). All the specified constraints are satisfied, but something is still wrong. Can you spot the problem?

Here Ann is a parent of Bill, who is a parent of Colin who is a parent of Ann. This makes Ann a great grandparent of herself. While reincarnation might make this possible, let's assume that we wish to exclude this possibility from our UoD. We can do this by declaring the relation to be *acyclic*. This means the relation has no cycles (i.e., there is no path via the relation from an object back to itself).

Figure 7.19 An irreflexive, functional relation must be intransitive.

Figure 7.20 In an acyclic relation, no loops are allowed.

Figure 7.20(a) shows an intuitive icon for acyclicity, with the bar indicating that no cycles back are allowed. A simplified version of this icon (without arrow-tips) is used for the acyclic constraint symbol, as shown in Figure 7.20(b). ORM 1 uses the alternative acyclic constraint symbol ^{O}ac.

Cycles may contain one or more links. Asymmetry is just the special case of acyclicity with two links in the cycle. Since *acyclic implies asymmetric*, if a role pair is declared acyclic, then don't declare it asymmetric.

Figure 7.20(c) shows a population for the parenthood relation that is both acyclic and intransitive. The two ring constraints are combined into a single icon as shown. A homogeneous binary relation can be displayed as a graph, using named circular nodes for the objects and arrows for the relationship instances, as in Figure 7.20(d). This makes it visually obvious whether a given population forms an acyclic relation.

A recursive definition for *acyclicity* is provided in the section exercise. Recall that the graph of a subtype family is directed and acyclic. Because of their recursive nature, acyclic constraints may be expensive or even impossible to enforce in some database systems. To reduce system overhead, such recursive constraint checks might be run in batch mode overnight, or even left to the data entry operators to do manually. In any case, the constraints should be identified so that a conscious decision can be made about their enforcement.

Figure 7.21 Relationships between the main ring constraints.

Of all the ring constraints, irreflexivity is the least expensive and acyclicity the most expensive to enforce. With expensive constraints such as acyclicity, *incremental checking* (for violations of the current update request, assuming the previous population is consistent with the constraint) is typically much cheaper than full checking.

The Euler diagram in Figure 7.21 shows the relationships between the main ring constraints discussed so far. For example, the set of acyclic relations forms a proper subset of the set of asymmetric relations, so acyclic relations must be asymmetric. Some other results you can read off include the following: an intransitive relation must be irreflexive; an asymmetric relation is an irreflexive, antisymmetric relation.

The symmetry constraint may be combined with intransitive or irreflexive constraints. In rare cases, a relation may be constrained to be *purely reflexive* (an object may bear the relationship to itself only); this constraint is depicted by the symmetric constraint symbol with an embedded "=" sign. Other properties of ring relations may apply (e.g., connectivity) but are not discussed here.

A classic example of an embedded ring relation is the "Bill of Materials" or "Parts Explosion" problem (see Exercise 4.4), which involves the ternary: Product contains Product in Quantity. You are invited to provide a more complete solution to this problem in the exercise that follows.

Exercise 7.3

1. For each predicate shown, state which, if any, of the following properties hold: reflexive, irreflexive; symmetric; asymmetric; transitive; intransitive.
   (a) has the same age as (b) is a brother of
   (c) is a sibling of (d) is shorter than
   (e) is an ancestor of (f) is at least as clever as
   (g) lives next to

2. Schematize the following report, which indicates composition of items (e.g., Item A contains two B parts and three C parts). Read "contains" as "directly contains" (e.g., A does not contain D). A more general reading for "contains" is taken in a later exercise. To help decide about ring constraints, draw a labeled graph for the containment relation, using an arrow for binary containment labeled with a number for the quantity.

Item	*Part*	*Quantity*
A	B	2
A	C	3
B	C	1
B	D	1
B	E	3
C	E	3
C	F	2

3. If you are familiar with predicate logic, prove the following theorems.
   (a) irreflex(R) & trans(R) $\rightarrow$ asym(R)
   (b) R is reflexive over its population iff $\forall x \forall y(xRy \vee yRx \rightarrow xRx)$. Show this is implied by sym(R) & trans(R).

4. The diagram shows the conceptual schema and the current population for a given UoD. Reference schemes are omitted for simplicity. Fact tables appear next to their fact types. Predicates are identified as **R**..**W**. Constraints are identified as $C_1..C_{15}$.

Each of the following requests applies to the **same** database population as shown (i.e., treat each request as if it was the *first* to be made on this population). For each request, indicate the CIP's response. If the request is legal, write "accepted". Otherwise, indicate a constraint violated (e.g., "C_2 violated").

(a) add: a1 S b1 (b) add: a3 R b3 (c) add: V a1 b1 d2 (d) add: a1 R b2
(e) add: a1 R b3 (f) add: a2 S b3 (g) delete: b3 T c3
(h) begin
add: b4 T c1
add: b4 U c2
end
(i) begin
add: b4 T c3
add: b4 U c2
end
(j) begin
add: V a1 b2 d2
add: V a1 b3 d2
end
(k) add: d1 W d3 (l) add: d1 W d1

5. Schematize the following output report. Here "bordering" means sharing a land border. In this sense, countries such as Australia have no bordering nations.

Nation	*Bordering nations*
Australia	
Belgium	France, Germany, Luxembourg, Netherlands
France	Belgium, Germany, Italy, Luxembourg, Spain, Switzerland
…	…

6. (a) Which, if any, of asymmetry and intransitivity are implied by acyclicity?
(b) Convince yourself that the following recursive definition of acyclicity is correct. Model relationships of type *R* graphically: consider the fact *xRy* as a directed line from *x* to *y*. The quantifier "∀" = "for all", and "∃" = "there exists".
x has a path to *y* **iff** [*xRy* **or** ∃*z*(*xRz* & *z* has a path to *y*)]
R is acyclic **iff** ∀x ~(*x* has a path to *x*)

7.4 Other Constraints and Rules

We now have covered most of the main graphical constraints in ORM. This section considers three more graphical constraints: object cardinality, role cardinality, and value-comparison constraints. It then discusses basic textual constraints and derivation rules. More advanced graphical and textual rules are discussed in Chapter 10.

An *object cardinality constraint* restricts the cardinality (number of members) of the population of an object type for each state of the database. The cardinality constraint "# = 12" on JuryMember in Figure 7.22(a) means there are always exactly 12 jury members. Here "#" denotes cardinality. Equality settings like this are very rare, as most databases start with the empty state, which would violate the constraint.

Inequality settings are more typical. For example, in Figure 7.22(b), "# ≤ 100" next to the Senator object type means there are at most 100 senators in each database state. Object cardinality constraints apply to the populations of the types rather than the types themselves. Over a long period, the total number of jury members or senators who ever filled these positions could be much higher.

A *role cardinality constraint* restricts the cardinality of the population of a role for each database state. The cardinality constraint "# ≤ 1" on the role in Figure 7.22(c) means there is always at most one employee playing the director role. Note that ORM cardinality constraints are not the same as so-called "cardinality" constraints in ER, which instead correspond to frequency and/or mandatory role constraints.

Figure 7.22 Object cardinality constraints (a and b) and role cardinality constraints (c).

Cardinality constraints are rarely included on diagrams, since they are *often implied by existing value constraints or frequency constraints*. For example, an {'M', 'F'} value constraint on Gender(.code) implies "# ≤ 2" on Gender. If there is at most one senate, then a frequency constraint of ≤ 100 on Senate's role in Figure 7.23(b) implies the # ≤ 100 cardinality constraint on Senator. Note however that the cardinality constraint # ≤ 1 on Senate is still required.

Figure 7.23 Object cardinality constraints implied by value or frequency constraints.

Figure 7.24 Examples of value-comparison constraints.

In practice, however, it is often more useful to allow many senates, in which case we constrain the size of each senate to at most 100 members using a frequency constraint, even though a cardinality constraint on Senate no longer applies (see Figure 7.23(c)).

A *value-comparison constraint* compares one value with another using one of four comparison operators $<$, $\leq$, $>$, $\geq$. The constraint is depicted by a circled operator, connected by a dotted arrow from the left operand role to the right operand role. Two dots are added to the circle to represent the two instances being compared (reminding us that this is a comparison between instances, not between sets). The value-comparison constraints in Figure 7.24 may be verbalized thus: **For each** Employee, birthdate < hiredate; **For each** Project, **existing** enddate ≥ startdate. For optional attributes (e.g., enddate), "**existing**" restricts the comparison to cases where attribute instances exist. So "**existing** enddate ≥ startdate" is short for "**if** enddate **exists then** enddate ≥ startdate". Hence if a project has no end date, the constraint is still satisfied (it maps to the SQL check clause **check**(enddate >= startdate) which is violated only when it evaluates to false).

Apart from being directional, value-comparison constraints are a bit like ring constraints. The roles being constrained are compatible (e.g., the birthdate and hiredate being compared belong to the same employee). Like irreflexive ring constraints, value-comparison constraints are typically inexpensive to enforce, as they typically compare values on the same row of a table rather than comparing sets of values.

You have now met almost all the ORM graphical notations. Some more advanced constraints and rules are discussed in Chapter 10. Adding many graphic notations for further constraints could make the graphical language hard to learn, and lead to cluttered diagrams. Even the value-comparison constraints just discussed might be considered graphically intrusive by some. A conceptual schema diagram provides a human-oriented, unambiguous, but often incomplete specification of the UoD structure.

Rules not expressed using predefined graphic symbols may be specified as *textual rules*, preferably in a high level formal language. In ORM 2, the preferred way to display these textual rules is via *footnotes*, with footnote numbers marking the involved model elements on the diagram. The textual rules (constraints, derivation rules, and subtype definitions) themselves may be displayed in text boxes, appearing as footnotes or elsewhere on the diagram, ideally with the option of toggling their display on or off.

Figure 7.25 shows one way to display three textual rules. Rule 1 declares that employees are hired after their birth. This is the same as the value-comparison constraint in Figure 7.24(a), but expressed textually. The footnote number 1 is appended to the name of the Employee type. A pair-exclusion constraint also exists between the birth and hire predicates, but is omitted since it is now implied.

Figure 7.25 Three examples of textual rules, one of which can be declared graphically.

Rule 2 says that non-executives use at most one company car. This is a *restricted uniqueness constraint*, since it strengthens the uniqueness constraint on Employee uses CompanyCar, but restricts this to a subtype of Employee.

Rule 3 says that each executive must have use of a company car. This is a *restricted mandatory role constraint*, since it makes an optional Employee role mandatory for a subtype of Employee. Some other versions of ORM declare restricted constraints by repeating the predicate on the subtype, with the stronger constraint shown there. Restricted constraints could also be displayed in views that restrict the supertype to the subtype.

To conclude this section, let's consider an example that involves textual constraints and some other interesting issues. Suppose our information system is to maintain details about computer networks. In a bus network, one or more computer workstations (WS) are connected along a bus or transmission line to a file server (FS) at one end. Figure 7.26 provides a topological picture of one such network (i.e., it shows which computers are connected to which but not their distance apart or directional pattern).

To simplify our discussion, the computers are identified by a single letter. The main memory of each node and the hard disk capacity of the file server are recorded. This diagram packs in a lot of information. Our task is to design a schema to record this information, and similar details about other bus networks, so that users can extract as much information from this model as they could extract from the actual network diagrams. Try this yourself before reading on. There are many different, correct solutions.

Figure 7.26 Topology of network 1.

Verbalizing the diagram in terms of elementary facts leads to several fact types, such as: Node belongs to Network; Node is of NodeKind {'FS', 'WS'}; Node has main memory of Capacity; FileServer has disk of Capacity. The reference schemes are obvious. For a subtype definition, we may use **Each** FileServer **is a** Node **that** is of NodeKind 'FS'. Now comes the hard part. How do we describe how the nodes are linked together?

Depending on our choice of primitive predicates, there are various options. If we find it hard initially to express the information in terms of elementary sentences, we can begin by just jotting down samples of whatever comes to mind. For example:

1. Node 'A' is directly connected to Node 'F'.
2. Node 'F' is directly connected to Node 'A'.
3. Node 'A' is indirectly connected to Node 'E'.
4. Node 'A' is an end node.
5. There is at most one link between each pair of nodes.
6. No link connects a node to itself.
7. Each node is directly connected to at most two nodes.

Let's stand back from the problem for a minute. There are two basic aspects about the topology that need to be captured. First, the nodes in a given network all lie along the same *continuous line* (in contrast to other shapes such as star networks, ring networks, and fancy loop structures). Second, the nodes are *positioned in some order* on this line.

If we consider the linearity of the structure to be a constraint, then how do we express the order of the nodes? Given that the nodes are positioned on a line, and that the file server is at one end of this line, let's first establish a *direction* for the line by saying the line starts at the file server. This avoids potential redundancy problems associated with an undirected, symmetric predicate (e.g., is directly connected to).

The ordering may be specified in terms of relative ordering or absolute ordering. With the first approach, we use a predicate for *directed linkage* (e.g., is just before). With the second approach, we assign each node an absolute position (e.g., 0, 1, 2, 3, 4 numbered from the file-server end).

The first approach leads to the ring fact type shown in Figure 7.27(a). This is populated with data from the sample network. An extended version catering for multiple networks is shown in Figure 7.27(b). This assumes that nodeId is a global identifier.

The inclusive-or constraint is displayed explicitly, since Node plays other roles in the complete schema. You are invited to add the relevant ring constraints, as well as the other fact types. If only complete networks are considered, the kind of node can now be derived (e.g., a node is a file server if and only if it has no nodes before it).

By storing linkage information only for direct, directed links, we reduce the size of our database as well as avoiding the problems of symmetry and transitivity. If desired, information about indirect links can be derived by using a recursive rule such as

For each Node_1 **and** Node_2,
Node_1 is before Node_2 **iff** Node_1 is just before Node_2 **or**
Node_1 is just before **some** Node_3 **that** is before Node_2.

Figure 7.27 A fact type for directed links in (a) one network and (b) many networks.

Other constraints exist. For example, each network has exactly one file server. If we are using the fact type Node is in Network, this may be specified textually. We can specify it graphically if we instead use separate fact types: Node is the file server for Network; Node is a workstation for Network.

However, some constraints have to be specified textually no matter how we choose our fact types. For example, directly linked nodes belong to the same network. If each network is to be continuous (no gaps), a constraint for this also needs to be declared textually. You are invited to express such constraints (and any others) formally.

As an exercise, try to model the linkage information using node position numbers instead, and notice the impact on constraints and derivation rules. Note that this absolute position approach makes it more awkward to add or delete nodes in the network since global changes to positional information would be required.

For instance, if we add a node, say D, between F and E, then not only will D be assigned the position 2, but nodes E, C, and B will all need to have their position numbers increased by 1. With our previous approach, only a local change would be needed. Swapping nodes is simple with both approaches. As an exercise, set out the compound transactions for some sample updates.

Exercise 7.4

1. (a) Rugby union teams have 15 positions, numbered 1 through 15. Teams are identified by name. Players have names (not necessarily unique) and are identified by their position in the team. Each player is a member of exactly one team and has exactly one position. It is possible for any team that not all of its 15 positions are currently filled. Specify an ORM schema for storing team membership details for multiple teams.
   (b) Now restrict the UoD to a single team, whose name must still be recorded.
   (c) Now restrict the UoD to a single team, whose name is not recorded.

2. A given company always has exactly one head employee, and each employee other than the head reports to exactly one employee. The reporting relationship is acyclic. Schematize.

3. Schematize the following table about the current age categories for humans. Unpack the semantics of ranges, and include any applicable graphical constraint.

Age category	*Age range (y)*
Child	0..14
Youth	15..24
Adult	25..64
Senior	65..140

4. A television survey on soap operas is conducted. Participants in this survey are identified by a number. Their gender is recorded as well as what soap operas they watch. Women may nominate all the soap operas they watch, if any, but to be included in the survey each man must specify exactly one soap opera. Model this UoD.

5. A *star network* has a centrally located file server directly linked to zero or more workstations. A sample star network is shown. For each node the generic kind of computer is recorded (PC or Mac). Only for the file server is the processor speed recorded. The maximum number of computers in any network is 20. The cable distance (in meters) of each node from the file server is shown next to each link. Schematize.

6. With reference to the Community-Roads UoD of Exercise 7.2, assume that roads are continuous, but may connect more than two communities. Discuss whether or not it is appropriate to use the ternary fact type Road goes from Community to Community.

7. Recall the Bill of Materials question in Exercise 7.3. A report is required to display parts contained at all levels (e.g., the fact that A contains part D is to be shown). Model this.

8. A *ring network* has a central file server with workstations arranged in a ring. A sample network is shown. Details are as for Question 4, except that no metric information is required. Specify the conceptual schema for this UoD.

7.5 Final Checks

The conceptual schema design procedure facilitates early detection of errors by various checks, including communication with the user by way of examples. Four final checks may also be performed to help detect any errors that might have slipped through. These are designed to ensure internal consistency, external consistency, lack of redundancy, and completeness. We now consider these in order.

A conceptual schema is *internally consistent* if and only if each role sequence used as a predicate or constraint argument can be populated in some state. This topic was discussed in the previous chapter, where it was called "*population consistency*" or "*strong satisfiability*". Basically it means that the specified constraints do not contradict one another when the system is populated. For example, the schema in Figure 7.28 has three faults. Can you spot them?

First, the frequency constraint of 2 on predicate *R* clashes with the uniqueness constraint; moreover, any frequency above 1 should be rejected if it spans the whole predicate. Second, the mandatory roles on *A* imply an equality constraint between these roles, so if *A* is populated, the exclusion constraint cannot be satisfied (recall theorem NXS).

Finally, the frequency constraint of 3 on the first role of *S* cannot be satisfied once *A* is populated. If it were satisfied, then *C* would include at least three instances (since each row of *S* is unique), but this is impossible since *C* is a proper subtype of *B* and hence has a maximum cardinality of 2 (the value constraint shows *B* has only three possible values).

With practice, it is usually easy to spot contradictory constraint patterns. CASE tools can be of assistance here. For example, the NORMA tool prevents most inconsistencies from even being entered.

A conceptual schema is *externally consistent* if it agrees with the original examples and requirements used to develop the schema. If not done earlier, we populate the schema with some of the sample data and look to see if some constraints are violated by the population. If they are, then the constraints are *too strong* in this respect, since they reject legal examples. We should then modify the schema, typically by removing or softening some constraints, until the population is permitted.

As a simple illustration, consider Table 7.2. To simplify the discussion, assume that people may be identified by their first name. The null value indicates that Bob seeks a BA degree but we do not know any of Bob's subjects.

Figure 7.28 This schema is population inconsistent on three counts.

Table 7.2 Student details.

Person	*Course*	*Degree*
Ann	CS112	BS
Ann	CS100	BS
Bob	?	BA
Sue	CS112	BA
Tom	CS213	BS

Now suppose that we propose the schema in Figure 7.29. Assuming that our fact types are correct, we might still get the constraints wrong. As well as asking the client directly about the constraints, we check by populating the fact tables with the sample data, as shown. Consider first the uniqueness constraint on studies. This asserts that each entry in this column is unique (i.e., each person studies at most one subject).

Figure 7.29 The constraints are inconsistent with the data.

However, 'Ann' appears twice here. So this constraint is wrong. The sample population makes it clear that the uniqueness constraint should span both columns. Now consider the uniqueness constraint on Person seeks Degree. This asserts that each entry in the degree column is unique. But the entry 'BS' appears twice there, as does the entry 'BA'. The presence of either of these cases proves that this constraint is wrong. This does not imply that the correct uniqueness constraint should span both columns: although the population is consistent with this weaker constraint, it is also consistent with a uniqueness constraint over the Person column.

To decide between the two possibilities, we ask the domain expert an appropriate question (e.g., Can the same person seek more than one degree at the same time?). If the answer to this is No, then we should replace the old uniqueness constraint with one across the Person column for this fact type.

Finally, note that the mandatory role constraint on studies is violated by the sample data, since Bob is referenced elsewhere in the database but does not play this role. So this role should be made optional. Figure 7.30 shows the fully corrected schema.

Figure 7.30 The corrected conceptual schema for Table 7.2.

While populating the schema with original examples may detect constraints that shouldn't be there, this will not automatically reveal missing constraints. In other words, while this check can pick up aspects of the constraint section that are too strong, we require significant examples or background knowledge to determine whether the constraint section is *too weak* in other respects.

We now check that the schema is free of *redundancy* by ensuring that no elementary fact can appear twice. Stored redundancy was covered in step 4, in the context of arity checking. We now turn to the notion of *derived redundancy*. The most common case of this is when a recorded fact can be derived from other facts by means of specified derivation rules.

The task of identifying derived fact types was considered in earlier steps. For example, the markup of an article was derived by subtracting its wholesale price from its retail price. Here it would be redundant to store the markup values. If markup values are not stored, then they need to be computed upon request: this has the disadvantage of adding slightly to the response time for markup queries.

If markup values are stored, this takes up extra storage. Moreover, if the wholesale or retail price of an article is changed, then failure to update markup accordingly leads to an inconsistent database. If such an update anomaly can occur, the redundancy is "*unsafe*".

You can arrange for *safe redundancy* or *controlled redundancy* in derived cases by having the derivation rule triggered by relevant updates. For example, the system can be configured to automatically "recalculate" the markup prices whenever wholesale or retail prices are updated (we assume markup prices may not be updated directly; this avoids the further overhead of a mutual recalculation among the three prices). This kind of approach is often used with spreadsheets.

Another derived case of safe redundancy occurs when the relevant facts are made nonupdatable. For example, a bank account balance may be computed at the end of each transaction and then stored immediately. This is a case of "derive on update" rather than "derive on query". This "eager evaluation" of derived information can improve the performance of an application considerably.

For example, if a bank with millions of customers produces monthly statements of account for each customer, accessing stored balances can save lots of recomputation. This is safe if the bank has a policy of never overwriting any balances. If an accounting error is found, it is left there but is compensated for by a later correcting transaction. This practice can also facilitate the task of auditing.

We sometimes meet cases where a stored fact type could be handled as a *semi-derivable* fact type. Recall the owner–driver example from the previous chapter (reproduced in Figure 7.31). In this UoD if a person drives a car then that person also owns that car. This feature is captured by the pair-subset constraint shown.

From a logical point of view, the subset constraint may be specified instead as a derivation rule: Person owns Car **if** Person drives Car. Unlike most derivation rules, which have "**iff**" as their main operator (and hence assert an equivalence), this is an "**if**" rule (and hence a mere conditional). Some people might own a car but not drive it. So the ownership fact type must be at least partly asserted.

Figure 7.31 Some owner facts are both asserted and derived.

From an implementation viewpoint, it is usually better to store the whole of the ownership relation, under the control of the subset constraint. This may take up a bit more storage, but access to ownership facts is fast. Moreover, this approach is safe and easy to understand.

If storage space is a problem, you could reduce storage of owner facts about owner–drivers by using the derivation rule. In the extreme case, a pair-exclusion constraint could even be enforced between owner and driver facts. However, this exclusion constraint approach requires great care, especially for managing updates. For example, consider deleting a driver fact for a person who remains an owner. Unless memory is tight, the subset constraint approach has much to recommend it.

From the *external* viewpoint, a user interface may be constructed to automatically add the required owner fact when a driver fact is added and to delete a driver fact when its owner fact is deleted. Such an interface could inform the user of its intended action and provide the option of canceling the transaction if this is not desired.

Apart from stored and derived redundancy, redundancy can occur within the set of derivation rules. For instance, it might be possible to derive a fact in more than one way, using different rules. Moreover, some rules might be derivable from more primitive rules. This situation is fairly common in formal inference systems (e.g., computer-aided reasoning systems).

In a wider context, redundancy can sometimes be very useful from the point of view of information retrieval, understanding (e.g., human communication), and coping with partial system failure (e.g., backup systems). This book itself exhibits a great deal of redundancy (e.g., by repeating important points). Although this makes the book longer, it hopefully makes it easier to follow.

The last check in the conceptual schema design procedure is to see if it is *complete* with respect to the original requirements specified for the application. This can be done systematically by going through each requirement, one at a time, and identifying which aspects of the conceptual design cater for it. If some requirements are found to be unfulfilled, the design should be extended to cater for them.

Apart from advanced aspects covered in Chapter 10, this completes the conceptual schema design procedure. In terms of the application development life cycle, there are still several things to be done (e.g., mapping the conceptual design to a database schema; implementing physical and external schemas; testing; and maintenance). However, the most crucial stages of the cycle have now been considered. The final exercise question is a long one, covering most aspects of the CSDP.

Exercise 7.5

1. Discuss any problems with this schema (reference schemes are omitted for simplicity).

2. From the sample output report shown, a student draws the conceptual schema diagram shown. Check to see if the data in the original report is a legal population of this schema. If not, modify the schema accordingly.

Particle	*Family*	*Charge (e)*
neutrino	lepton	0
electron	lepton	-1
muon	lepton	-1
pion	meson	1
kaon	meson	1
proton	baryon	1
neutron	baryon	0

3. Is this schema guilty of redundancy? If so, correct it.

Bookland	New York	100	300705
Websters	New York	100	300705
Bookland	New York	50	123555
OKBooks	London	200	123555

4. With respect to the previous question, suppose it is now possible that bookshops in different cities have the same name. For example, the following population is legal. Draw the correct conceptual schema diagram for this UoD.

Bookland	New York	100	300705
Websters	New York	100	300705
Bookland	New York	50	123555
Bookland	London	200	123555

5. In Prolog, relations are often partly stored and partly derived. For example, at one state the knowledge base might consist of the following facts and rules. Discuss this situation making reference to the notion of derived redundancy.

   parent_of(ann, bob).
   parent_of(bob, chris).
   grandparent_of (david, chris).
   grandparent_of (X, Y) **if** parent_of(X, Z) **and** parent_of(Z, Y).

6. An information system is to deal with various colors that are classified as primary (P), secondary (S) or tertiary (T). There are three primary colors: blue, red, and yellow. A secondary color is composed of a mixture of exactly two primary colors. A tertiary color is composed of exactly three primary colors.

   Each color has a unique, identifying trade name (e.g., "forest green"). The trade names of the primary colors are "blue", "red", and "yellow". Each color has a (perhaps zero) percentage of blue, red, and yellow. This percentage is expressed as a whole number in the range 0..100. For example, forest green is 70% blue and 30% yellow, but has no red. The following extract from a sample output report indicates the sort of information that needs to be accessed.

Color	*% blue*	*% red*	*% yellow*	*Class*
forest green	70	0	30	S
mud brown	30	30	40	T
red	0	100	0	P

   It is required to reduce the size of the database as much as possible by using derivation rules. For example, the percentage of a given color that is yellow should be derived from the percentages for blue and red. Schematize this UoD, clearly indicating all constraints. State each derivation rule—you may use obvious abbreviations (e.g., "%B", "%R", and "%Y"). As long as circularity is avoided, the rule for computing the percent yellow may be assumed in formulating other rules (i.e., other rules may use the term "%Y").

7. An information system is required to manage data about various astronomical bodies in our solar system. The first table shows an extract from a report about moons. A "?" mark indicates unknown, and "--" indicates non-existent.

Moon	*Planet*	*Mean radius (km)*	*Discovered*	*Explored?*	*Has water ice?*	*Atmosphere*	
						Gas	*%*
The Moon	Earth	1738	?	True	True	Ne	29.0
						He	25.8
						H	22.6
						Ar	20.6
Deimos	Mars	6.3	1877	True	False	--	--
Phobos	Mars	11.1	1877	True	False	--	--
Europa	Jupiter	1561	1610	True	True	O_2	100
Ganymede*	Jupiter	2631.2	1610	True	True	O_2	100
S/2003 J2	Jupiter	1	2003	False	False	--	--

* Largest moon in our solar system

Until a moon is given a normal name, it is assigned a temporary name based on the order in which the satellite was discovered for a given year and planet (do *not* bother unpacking these semantics). Here "Explored" means the moon has been investigated by a spacecraft (manned or unmanned) from Earth. If a moon has an atmosphere, only its main gases are listed.

If a moon has been explored, details about the missions to that moon may be recorded. A relevant extract of a report in this regard is shown in the next table. The meaning of bracketed contents in the mission entries in the first column needs to be made explicit. A "?" mark is a simple null (unknown) and a "–" mark indicates inapplicable because of other data.

Mission	*Moon*	*First Arrival*	*Crewed*	*Type*	*Astronauts*
Luna 2 (USSR)	The Moon	1959 Sep 13	No	Impact	–
Ranger 7 (US)	The Moon	1964 Jul 31	No	Impact	–
Luna 9 (USSR)	The Moon	1966 Feb 3	No	Lander	–
Apollo 11 (US)	The Moon	1969 July 20	Yes	Lander	Neil Armstrong, Buzz Aldrin, Michael Collins
Luna 17 (USSR)	The Moon	1970 Nov 17	No	Rover	–
Galileo (US)	Europa	1996 Oct 20	No	Flyby	–
Galileo (US)	Ganymede	1996 Nov 22	No	Flyby	–
Voyager 2 (US)	Europa	?	No	?	–

Details of interesting features of various moons may be recorded. The next table shows a small extract of details from this record. Feature length measures are of only two types (diameter and linear). The type of a moon's feature does not determine its size type. For example, some grooved terrains might have a pencil-like shape and have their linear dimension measured, whereas other grooved terrains might have a roughly circular shape and have their diameter measured. Moreover, the size of a feature is recorded if and only if the size type of that feature is recorded.

Moon	*Feature*	*Feature Type*	*Age (y)*	*Size (km)*	*Size Type*
The Moon	Aristarchus Crater	Impact Crater	300 000 000	37	Diameter
	Mare Crisium	Sea	3 900 000 000	563	Diameter
	Mare Tranquillitatis	Sea	3 600 000 000	873	Diameter
	Montes Apenninus	Mountain range	3 900 000 000	?	?
Ganymede	Rupes Altai	Cliff	4 200 000 000	507	Linear
	Nicholson Regio	Dark area	?	3900	Diameter
	Uruk Sulcus	Grooved terrain	?	7	Linear
	Sippar Sulcus	Grooved terrain	?	10	Diameter

The next table shows an extract of details about astronauts. The information system identifies astronauts by numbers to allow different astronauts with the same name and to track astronauts who change their names (ignore name tracking in your solution).

Astro#	Name	Gender	Nationality	FirstSpaceTrip	Born
1	Yuri Gargarin	M	RU	1961	1934
2	Alan Shepherd	M	US	1961	1923
3	Valentina Tereshkova	F	RU	1963	1937
4	Neil Armstrong	M	US	1966	1930
5	Buzz Aldrin	M	US	1966	1930
6	Michael Collins	M	US	1966	1930
7	Sally Ride	F	US	1983	1951
8	Rodolfo Neri Vela	M	MX	1985	1952

The next table shows an extract from another report about moons and planets in our solar system. A body's name type must be either normal or temp (temporary). If recorded, the origin of a body's name is treated as an atomic value (an uninterpreted character string).

Body	*Type*	*Name Type*	*Name Origin*
Mercury	Planet	Normal	God of trade, and messenger to the gods (wore winged shoes)
Venus	Planet	Normal	Goddess of love
Earth	Planet	Normal	?
Mars	Planet	Normal	God of war
Jupiter	Planet	Normal	?
The Moon	Moon	Normal	The only moon of Earth
Deimos	Moon	Normal	A son of the god Ares (Mars)
Phobos	Moon	Normal	A son of the god Ares (Mars)
Europa	Moon	Normal	Cup-bearer to the Greek gods
Ganymede	Moon	Normal	Daughter of Agenor (king of Tyre)
S/2003 J2	Moon	Temp	–

The next table shows a report of details about planets in our solar system. Each planet has either its average surface temperature or its temperature range recorded. At most four main gases may be recorded for each planet (or moon—see first table). Any data that can be derived should be.

Planet	*Avg. Surface Temperature*	*Temperature Range*		*Ringed?*	*Nr rings*	*Atmosphere*
		Min	*Max*			
Mercury	–	-180°C (-290°F)	430°C (810°F)	No	0	O_2 (52%), Na (39%), He (8%)
Venus	464°C (867°F)	–	–	No	0	CO_2 (96.5%), N_2 (3%)
Earth	15°C (59°F)	–	–	No	0	N_2 (78.1%), O_2 (20.9%)
Mars	–	-125°C (-195°F)	25°C (77°F)	No	0	CO_2 (95.3%), N_2 (2.7%)
Jupiter	-110°C (-160°F)	–	–	Yes	3	H_2 (89.8%), He (10%)
Saturn	-140°C (-220°F)	–	–	Yes	7	H_2 (96.3%), He (3.5%)
Uranus	-214°C (-353°F)	–	–	Yes	11	H_2 (82.5%), He (15.2%), CH_4 (2.3%)
Neptune	-200°C (-320°F)	–	–	Yes	6	H_2 (79%), He (18%), CH_4 (2.9%)
Pluto	-230°C (-364°F)	–	–	No	0	N_2 (99.97%)

Over time, some countries may divide into other countries. For example, in 1991 the USSR (country code SU) split into 15 independent countries, and recently Bosnia and Herzegovina split into 2 countries. The next table shows a small extract of a table to keep track of this.

Country	*Parent*
LV	SU
EE	SU
RU	SU

Specify an ORM conceptual schema for this UoD. Include all relevant uniqueness, mandatory, value, set-comparison, subtyping (including subtype definitions), frequency, ring, and value-comparison constraints. If an object type is independent, indicate this. If a fact type is derived, include it on the diagram, as well as providing its derivation rule.

7.6 Summary

CSDP step 7 adds other constraints and performs final checks. An *occurrence frequency constraint* indicates that an entry in a column (or column combination) must occur there exactly n times (n), at most n times ($\leq n$), at least n times ($\geq n$), or at least n and at most m times (n .. m). A simple occurrence frequency appears next to the role. A compound occurrence frequency appears next to a line connecting the relevant roles.

A *ring constraint* can apply only to a pair of roles played by the same (or a compatible) object type. The role pair may form a binary predicate or be embedded in a longer predicate. Let R be the relation type comprising the role pair. R is *reflexive* (over its populations) iff for all x playing either role, xRx. R is *symmetric* iff for all x, y, $xRy \rightarrow yRx$. R is *transitive* iff for all x, y, z, xRy & $yRz \rightarrow xRz$. These positive properties tend to be used for derivation rather than as constraints.

The following negative properties may be marked as ring constraints next to the role pair (or role connector). R is *irreflexive* iff for all x, $\sim xRx$. R is *asymmetric* iff for all x, y, $xRy \rightarrow \sim yRx$. R is *antisymmetric* iff for all x, y, $x \neq y$ & $xRy \rightarrow \sim yRx$. R is *intransitive* iff for all x, y, z, xRy & $yRz \rightarrow \sim xRz$. Asymmetry and intransitivity each imply irreflexivity. Exclusion implies asymmetry (and irreflexivity). An irreflexive, functional relation must be intransitive. A recursive ring constraint that may be difficult to enforce is *acyclicity*. Symbols for all the ring constraints are summarized in the ORM symbol glossary at the end of the book.

An *object cardinality constraint* or *role cardinality constraint* limits the cardinality of each population of an object type or role, respectively. An upper limit of n is denoted by "$\# \leq n$" next to the object type or role. Cardinality constraints are rarely used, since they are *often implied by existing value constraints or frequency constraints*.

A *value-comparison constraint* compares one value with another using one of the operators $<$, $\leq$, $>$, $\geq$. The constraint is depicted by a circled operator, connected by a dotted arrow from the left operand role to the right operand role. Two dots on the circle represent the two instances being compared. A typical example of this kind of constraint is **For each** Employee, birthdate < hiredate.

Constraints for which no graphic symbol exists may be specified as *textual rules*. The model elements involved in the rule are marked with a footnote number, and the rule itself may be displayed in a text box.

A restricted uniqueness constraint declares stronger uniqueness for a given subtype. A restricted mandatory role constraint indicates that an optional role is mandatory for a given subtype.

When modeling connections between nodes, choose directed, immediate links for the stored relation. Indirect links may then be derived recursively.

At the end of the *CSDP*, final checks are made to ensure that the conceptual schema is internally consistent (its constraint pattern can be populated without contradiction), is externally consistent (agrees with original data and conditions), is redundancy free (elementary facts can't be repeated), and is complete (covers all the requirements). Various cases of derived redundancy can be safely managed.

Chapter Notes

In contrast to our usage, occurrence frequencies are sometimes defined as the number of times members of an object type population must play a role—frequencies of 0 are then allowed and mean optional. Ring constraints are often discussed in logic books, within the context of the theory of dyadic relations, but are rarely discussed in modeling approaches other than ORM. Some extended versions of intransitivity were introduced by Peter Ritson.

8 Entity Relationship Modeling

8.1 Overview of ER
- Chen's original ER notation
- Industrial ER

8.2 Barker Notation
- Barker ER notation
- Comparison with ORM

8.3 Information Engineering Notation
- IE notation
- Comparison with ORM

8.4 IDEF1X
- IDEF languages
- IDEF1X notation
- Comparison with ORM

8.5 Mapping from ORM to ER
- Mapping procedures

8.6 Summary

8.1 Overview of ER

The *Entity Relationship* (ER) modeling approach views a business domain in terms of entities that have attributes and participate in relationships. For example, the fact that an employee was born on a date is modeled by a birthdate attribute of the Employee entity type, whereas the fact that an employee works for a department is modeled as a relationship between them. This world view is quite intuitive, and despite the recent rise of UML for modeling object-oriented applications, ER is still the most popular data modeling approach for database applications.

In Chapter 1, we argued that ORM is better than ER for conceptual analysis. However, ER is widely used, and its diagrams are good for compact summaries, so you should become familiar with at least some of the mainstream ER notations. This is the main purpose of this chapter. A second purpose is to have a closer look at how ORM compares to ER. To save some explanation, we assume that you have already studied the basics of ORM in earlier chapters so that we can examine ER from an ORM perspective.

The ER approach was originally proposed by Dr. Peter Chen in the very first issue of an influential journal (Chen 1976). Figure 8.1 is based on examples from this journal paper. Chen's original notation uses rectangles for entity types and diamonds for relationships (binary or longer). Attributes may be defined, but are excluded from the ER diagram. As in ORM, roles are defined as parts played in relationships. Role names may optionally be shown at relationship ends (e.g., Employee plays the worker role in the Proj-Work relationship).

Chen formalized relationships in terms of ordered tuples of entities, allowing the order to be dropped if role names are used (as with attribute names in the relational model). Although not displayed on the ER diagram, relationships may have attributes, but cannot play roles in other relationships. So objectified associations are not fully supported.

Figure 8.1 The original ER notation used by Chen.

Figure 8.2 An ambiguous ER schema.

Roles may be annotated with a maximum cardinality of 1 or many. For example, read left to right, the Proj-Mgr relation is one-to-many (each employee manages zero or more projects, and each project is managed by at most one employee).

As shown in Figure 8.1, Chen used noun phrases for relationships, eliminating natural verbalization. Even if verb phrases are used, the direction in which relationship names are to be read is formally undecided, unless we add additional marks (e.g., arrows) or rules (e.g., always read from left to right and from top to bottom). For example, does the employee manage the project or does the project manage the employee? Although we can use our background knowledge to informally disambiguate this example, it is easy to find examples of relationships whose direction can only be guessed at by anybody other than the model's creator.

This problem is exacerbated if the verb phrase used to name the relationship is shortened to one word (e.g., "work"), unfortunately still a fairly common practice. As a simple example, is the ER diagram in Figure 8.2 meant to capture the fact type Person killed Animal, or Animal killed Person? We could disambiguate this diagram by adding role names (e.g., "victim", "killer") but there is no requirement to do so. Similarly, if we populate the Component relationship in Figure 8.1 with the pair (*a*, *b*), we don't know whether this means *a* is a component of *b*, or vice versa. To disambiguate this, we need to add role names (e.g., "subpart", "superpart") or use a verb phrase (e.g., "is a component of") with a defined direction.

A rectangle with a double border denotes a *weak entity type*. This means that the entity type's identification scheme includes a relationship to another entity type. In Figure 8.3(a) for example, an instance of Dependant might be identified by having the name "Eve Jones" and being related via the Emp-Dep relation to employee 007. In ORM this would be modeled by a coreferenced object type as in Figure 8.3(b).

Figure 8.3 A weak entity type in ER (a) remodeled in ORM (b).

The arrow tip at the Dependant end of the ER relationship indicates that Dependant is existence dependent on Employee (its existence depends on the existence of the other). Given that Dependant is weak, this is basically redundant.

A far better approach is to introduce the concept of a mandatory role, as in ORM and many other ER versions (e.g., **Each** Dependant is a dependant of **at least one** Employee). This ability to establish a minimum multiplicity of at least one for any given role was absent from Chen's original notation.

Over time, many variant notations developed. Attributes were sometimes displayed as named ellipses, connected by an arrow from their entity type, with double ellipses for identifier attributes. Chen's current ER-Designer tool uses hexagons instead of diamonds. One problem with the ER approach is that there are so many versions of it, with no single standard. In industry, the most popular versions of pure ER are the Barker and Information Engineering (IE) notations. These are discussed in the next two sections. Another popular data modeling notation is IDEF1X, which is a hybrid of ER and relational notation, so is not a true ER representative. Nevertheless, many people talk of IDEF1X as a version of ER, so we cover it in this chapter.

The best way to develop an ER or IDEF1X model is to derive it from an ORM model; we briefly discuss mapping from ORM later in the chapter. The UML class diagram notation may be regarded as an extended version of ER, but because of the importance of UML, it is considered separately in the next chapter.

8.2 Barker Notation

We use the term "Barker notation" for the ER notation discussed in the classic treatment by Richard Barker (Barker, 1990a). Originating in the late 1980s at CACI in the United Kingdom, the notation was later adopted by Oracle Corporation in its CASE design tools. Oracle now supports UML as an alternative to the Barker ER notation, although for database applications, many modelers still prefer the Barker notation over UML. Recently, Embarcadero has added basic support for the Barker notation in its EA/Studio product. Although dozens of ER dialects exist, we consider the Barker notation to be the best ER notation with wide support in industry.

Dave Hay, an experienced modeler and fan of the Barker notation, argues that "UML is … not suitable for analyzing business requirements in cooperation with business people" (Hay, 1999b). While UML class diagrams are less than ideal for data modeling, the Barker ER notation, like UML, is attribute-based. As discussed in Chapter 1, using attributes in a conceptual model adds complexity and instability, and makes it harder to validate models using verbalization and sample populations.

Attributes are great for logical design, since they allow compact diagrams that directly represent the implementation data structures (e.g., tables or classes). However, for conceptual analysis, we just want to know what the *facts* and *rules* are about the business and to communicate this information in *sentences* so that the model can be understood by the domain experts. Whether some fact ends up in the design as an attribute should not be a conceptual issue.

Figure 8.4 A simple ER model in the Barker notation.

In defense of ER, it can be useful to view a conceptual schema in attribute-style to gain a compact but still high level view of the business domain. For this reason, we value ER, and the NORMA tool for ORM is currently being extended to provide ER diagrams as live views of underlying ORM schemas.

But, as Ron Ross (1998, p. 15) says, "Sponsors of business rule projects must sign off on the sentences—not on graphical data models. Most methodologies and CASE tools have this more or less backwards". ORM allows the domain expert to inspect ORM models fully verbalized into sentences with examples, making validation much easier and safer.

Now that we've stated our bias up front, let's examine the Barker notation itself. The basic conventions are illustrated in Figure 8.4. *Entity types* are shown as soft rectangles (rounded corners) with their name in capitals. *Attributes* are written below the entity type name. Some constraint information may appear before an attribute name. An octothorpe "#" indicates that the attribute is, or is a component of, the primary identifier of the entity type.

A "*" or heavy dot "•" indicates that the attribute is mandatory (i.e., each instance in the database population of the entity type must have a nonnull value recorded for this attribute). A "°" indicates the attribute is optional. Some modelers also use a period "." to indicate that the attribute is not part of the identifier.

Relationships are restricted to binaries (no unaries, ternaries, or longer relationships) and are shown as lines with a relationship name at the end from which that relationship name is to be read. This name placement overcomes the ambiguous direction problem mentioned earlier. Both forward and inverse readings may be displayed for a binary relationship, one on either end of the line. This makes the Barker notation superior to UML for verbalizing relationships.

From an ORM perspective, each end (or half) of a relationship line corresponds to a role. Like ORM, Barker treats role optionality and cardinality as distinct, orthogonal concepts instead of lumping them together into a single concept (e.g., multiplicity in UML).

A *solid half-line denotes a mandatory role*, and a *dotted half-line indicates an optional* role. For cardinality, a *crow's foot* intuitively *indicates "many"*, by its many "toes". The absence of a crow's foot intuitively indicates "one". The crow's foot notation was invented by Dr. Gordon Everest, an ORM advocate, who originally used the term "inverted arrow" (Everest, 1976) but now calls it a "fork". Figure 8.5 shows the basic correspondence with the ORM notation for simple mandatory and uniqueness constraints.

Figure 8.5 The ER diagram (a) is equivalent to the ORM diagram (b).

To enable the optionality and cardinality settings to be verbalized, Barker (1990a, p. 3–5) recommends the following *naming discipline for relationships*. Let *A R B* denote an infix relationship *R* from entity type *A* to entity type *B*. Name *R* in such a way that each of the following four patterns results in an English sentence:

Each A (**must** | **may**) **be** *R* (**one and only one** *B* | **one or more** *B-plural-form*)

Use "must" or "may" when the first role is mandatory or optional respectively. Use "one and only one" or "one or more" when the cardinality on the second role is one or many, respectively. For example, the optionality/cardinality settings in Figure 8.5(a) verbalize as: **Each** Employee **must be** an occupier of **one and only one** Room; **Each** Room **may be** occupied by **one or more** Employees.

The constraints on the left-hand role in the equivalent ORM model in Figure 8.5(b) verbalize as: **Each** Employee occupies **some** Room; **Each** Employee occupies **at most one** Room. These constraints may be combined to verbalize as **Each** Employee occupies **exactly one** Room. The lack of a uniqueness constraint on the right-hand role is verbalized as **It is possible that the same** Room is occupied by **more than one** Employee, or (if no inverse reading exists) as **It is possible that more than one** Employee occupies **the same** Room.

Regarding the lack of an explicit mandatory role constraint on the right-hand role, we don't want that verbalized explicitly, because it may well be unstable. If Room plays no other fact roles, the role is mandatory by implication (Room is not declared independent), so verbalization may well confuse here. If Room does play another fact role, and we decide that some rooms may be unoccupied, we could declare this explicitly either as **It is possible that some** Room is occupied by **no** Employee or as **It is not necessary that each** Room is occupied by **some** Employee. If no inverse reading is available, it may be verbalized thus: **It is possible that no** Employee occupies **some** Room.

To its credit, the Barker verbalization convention is good for basic mandatory and uniqueness constraints on infix binaries. But it is far less general than ORM's approach, which applies to instances as well as types, for predicates of any arity, infix or mixfix, and covers many more kinds of constraint, with no need for pluralization. As a trivial example, the fact instance "Employee '101' an occupier of Room 23" is not proper English, but "Employee '101' occupies Room 23" is good English.

If each role in a binary association is assigned one of optional/mandatory and one of many/one, there are 16 patterns. The equivalent Barker ER and ORM diagrams for these cases are shown in Figure 8.6. The last case where both roles of a many:many relationship are mandatory is considered illegal in Barker ER.

Ring associations considered illegal by Barker are shown in Figure 8.7(a). Although rare, they sometimes occur in reality, so they should be allowed at the conceptual level, as permitted in ORM.

Figure 8.6 Equivalent constraint patterns.

Figure 8.7 Illegal ring associations in Barker ER (a) that are allowed in ORM (b).

As an exercise, you may wish to invent satisfying populations for the ORM associations in Figure 8.7(b). Although considered illegal by Barker, at least some of these patterns are allowed in Oracle's CASE tools.

In the Barker notation, a bar "|" across one end of a relationship indicates that the relationship is a component of the primary identifier for the entity type at that end. For example, in Figure 8.8, Employee and Building have simple identifiers, but Room is compositely identified by combining its room number and building.

Figure 8.8 Room is identified by combining its room nr and its Building relationship.

The use of identification bars provides some of the functionality afforded by external uniqueness constraints in ORM. For example, the schemas in Figure 8.9 are equivalent. Any other attributes of Room and Building would be modeled in ORM as relationships. ORM's external uniqueness notation seems to us to convey more intuitively the idea that each RoomNr, Building combination is unique (i.e., refers to at most one room). But maybe we're biased. At any rate, this constraint (as well as any other graphic constraint) can be automatically verbalized in natural language.

Figure 8.9 Composite identification in (a) Barker ER and (b) ORM.

Figure 8.10 A room schedule model in Barker notation.

Some people misread the bar notation for composite identification as a "1", since this is what the symbol means in many other ER notations. But the main problem with the bar and "#"notations is that they cannot be used to declare uniqueness constraints not used for primary reference (see later). A second problem, arising from the practice of modeling some fact types as attributes and others as relationships, is that two very different notations are used for the same fundamental concept (uniqueness).

Section 1.2 used a room scheduling example to illustrate several fundamental differences between ORM and modeling approaches such as ER and UML. You may recall that the schema shown in Figure 8.10 was used to model the example in the Barker notation. If you skipped Section 1.2, you may wish to read it now, since it discussed this example in a lot of detail. The use of attributes and the binary-only relationship restriction in this model makes it hard to verbalize and populate the schema for validation purposes. Moreover, there is at least one constraint missing.

A populated ORM schema for this example is reproduced in Figure 8.11 (minus the counterexample rows discussed in Section 1.2). Here the facts are naturally verbalized as a ternary and a binary, and the constraints are easily checked using verbalization and sample data. With the ER model there is no way of specifying the right-hand uniqueness constraints on either fact type, since the Barker notation doesn't capture uniqueness constraints on attributes or relationships unless they are used for primary identification.

Figure 8.11 A populated ORM model for room scheduling.

Figure 8.12 A simple exclusion constraint in (a) Barker notation and (b) ORM.

In case it looks like we're just bashing attribute-based approaches such as ER, let us reiterate that we find attribute-based models useful for compact overviews and for getting closer to the implementation model. However, we generate these by mapping from ORM, which we use exclusively for the initial conceptual analysis. This makes it easier to get the model right in the first place and to modify it as the underlying domain evolves.

Unlike ER (and UML for that matter), ORM was built from a linguistic basis, and its graphic notation was carefully chosen to exploit the potential of sample populations. To reap the benefits of verbalization and population for communication with and validation by domain experts, it's better to use a language that was designed with this in mind. An added benefit of ORM is that its graphic notation can capture many more business rules than popular ER notations. Although not as rich as ORM, the Barker notation is more expressive than many ER notations. The rest of this section discusses its support for advanced constraints and subtyping.

In Barker notation, an exclusion constraint over two or more roles is shown as an "*exclusive arc*" connected to the roles with a small dot or circle. For example, Figure 8.12(a) includes the constraint that no employee may be allocated both a bus pass and a parking bay. In ORM this constraint is depicted by connecting "⊗" to the relevant roles by a dotted line, as shown in Figure 8.12(b).

To declare that two or more roles are mutually exclusive and disjunctively mandatory, the Barker notation uses the exclusive arc, but each role is shown as mandatory (solid line). For example, in Figure 8.13(a) each account is owned by a person or a company, but not both.

Figure 8.13 An exclusive-or constraint in (a) Barker notation and (b) ORM.

This notation is liable to mislead, since it violates the orthogonality principle in language design. Viewed by itself, the first role of the association Account owned by Person would appear to be mandatory, since a solid line is used. But the role is actually optional, since superimposing the exclusive arc changes the semantics of the solid line to mean the role belongs to a set of roles that are disjunctively mandatory.

Contrast this with the equivalent ORM model shown in Figure 8.13(b). Here an exclusion constraint ⊗ is orthogonally combined with a disjunctive mandatory (inclusive-or) constraint ⊙ to produce an exclusive-or constraint, shown here by the "lifebuoy" symbol formed by overlaying one constraint symbol on the other.

The ORM notation makes it clear that each role is individually optional and that the exclusive-or constraint is a combination of inclusive-or and exclusion constraints. Suppose we modified our business so that the same account could be owned by both a person and a company. Removing just the exclusion constraint from the model leaves us with the inclusive-or constraint ⊙ that each account is owned by a person or company. Like UML, the Barker ER notation doesn't even have a symbol for an inclusive-or constraint, so it is unable to diagram this or the many other cases of this nature that occur in practice.

In the Barker notation, a role may occur in at most one exclusive arc. ORM has no such restriction. For example, in Figure 8.14(a) no student can be both ethnic and aboriginal, and no student can be both an aboriginal and a migrant (these rules come from a student record system in Australia). Even if the Barker notation supported unaries (it doesn't), this situation could not be handled by exclusive arcs. Like UML, Barker ER does not provide a graphic notation for exclusion constraints over role sequences. For instance, it cannot capture the ORM pair-exclusion constraint in Figure 8.14(b), which declares that no person who wrote a book may review the same book. Such rules are very common. Moreover, the Barker notation cannot express any ORM subset or equality constraints at all, even over simple roles.

The Barker notation for ER allows simple *frequency constraints* to be specified. For any positive integer n, a constraint of the form $= n$, $< n$, $\leq n$, $> n$, $\geq n$ may be written next to a single role to indicate the number of instances that may be associated with an instance playing the other role. For example, the frequency constraint "≤ 2" in Figure 8.15 indicates that each person is a child of at most two parents. In the Barker notation, this constraint is placed on the parent role, making it easy to read the constraint as a sentence starting at the other role.

Figure 8.14 Some ORM exclusion constraints not handled by Barker's exclusive arcs.

Figure 8.15 A simple frequency constraint in (a) Barker notation and (b) ORM.

In ORM the constraint is placed on the child role, making it easy to see the impact of the constraint on the population (each person appears at most twice in the child role population). Unlike the Barker notation, ORM allows frequency constraints to include ranges (e.g., 2..5) and to apply to role sequences.

In Barker notation, *subtyping* is depicted with a version of Euler diagrams. In effect, only partitions (exclusive and exhaustive) can be displayed. For example, Figure 8.16(a) indicates that each patient is a male patient or female patent but not both. As discussed in Section 6.5, ORM displays subtyping using directed acyclic graphs (DAGs), and may use an exclusive-or constraint symbol to display a partition constraint (typically implied by subtype definitions and other constraints).

Euler diagrams are good for simple cases, since they intuitively show the subtype inside its supertype. However, unlike DAGs, they are hopeless for complex cases (e.g., many overlapping subtypes), and they make it inconvenient to attach details to the subtypes. For the latter reason, attributes are sometimes omitted from subtypes when the Barker notation is used. Another problem with Euler diagrams is in displaying multiple partitions on a single diagram (e.g., try partitioning Patient into not just MalePatient and FemalePatient, but also InPatient and OutPatient).

In the Barker notation, if the original subtype list is not exhaustive, an "Other" subtype is added to make it so, even if it plays no specific role. For example, in Figure 8.17 a vehicle is a car or truck or possibly something else, and a car is a sedan or wagon or possibly something else. In ORM, there is no need to introduce subtypes for OtherCar or OtherVehicle unless they play specific roles.

Figure 8.16 A subtype partition in (a) Barker ER and (b) ORM

Figure 8.17 Nonexhaustive, exclusive subtypes in (a) Barker ER and (b) ORM.

A major problem with the Barker notation for subtyping is that it does not depict overlapping subtypes (e.g., Manager and FemaleEmployee as subtypes of Employee) or multiple inheritance (e.g., FemaleManager as a subtype of FemaleEmployee and Manager). While it is possible to implement multiple inheritance in single inheritance systems (e.g., Java) by using some low level tricks, for conceptual modeling purposes multiple inheritance should be simply modeled directly. As a final comparison point about subtyping, Barker ER lacks ORM's capability for formal subtype definitions and context-dependent identification schemes.

In addition to its static constraints, Barker ER includes a dynamic "changeability constraint" for marking "*nontransferable relationships*". This constraint declares that once an instance of an entity type plays a role with an object, it cannot ever play this role with another object. This is indicated by an open diamond on the constrained role. For example, Figure 8.18(a) declares that the birth country of a person is nontransferable.

As indicated in Figure 8.18(b), ORM does not currently include a graphic notation for this constraint. At the time of writing, adding formal support for this and other dynamic constraints in ORM is an active research area. In practice, specification of nontransferable constraints needs to ensure that the implemented model is still open to error corrections by duly authorized users. For example, if an Australian's birth country is mistakenly entered as Austria, it should be possible to change this to Australia. We discuss this issue further in the next chapter when examining changeability properties in UML.

Figure 8.18 Nontransferability declared in Barker, but not ORM.

Well that pretty well covers the Barker notation for ER. As we've seen, it does a good job of expressing simple mandatory, uniqueness, exclusion, and frequency constraints, simple subtyping, and also nontransferable relationships. However, if a feature is modeled as an attribute instead of as a relationship, very few of these constraints can be specified for it.

Unlike ORM, the Barker notation does not support unary, *n*-ary, or objectified associations (nesting). Moreover it lacks support for most of the advanced ORM constraints (e.g., subset, multi-role exclusion, ring constraints, and join constraints). It does not include a formal textual language for specifying queries, other constraints and derivation rules at the conceptual level. Nevertheless it is better than many other ER notations and is still widely used. If you ever need to specify a model in Barker ER notation, we suggest that you first do the model in ORM, map it to the Barker notation, and make a note of any rules that can't be expressed there diagrammatically.

Rather than giving you some exercises on the Barker notation at this point, we'll wait until the end of the chapter, when we've covered the main ER notations in use as well as some techniques for mapping from ORM to ER. You can then decide which notation(s) you would like to have some practice with.

8.3 Information Engineering Notation

The *Information Engineering* (IE) approach began with the work of Clive Finkelstein in Australia and CACI in the United Kingdom, and was later adapted by James Martin. Different versions of IE exist, with no single standard. In one form or other, IE is supported by many data modeling tools and is one of the most popular notations for database design.

In the IE approach, *entity types* are shown as named rectangles, as in Figure 8.19(a). A*ttributes* are often displayed in a compartment below the entity type name, as in Figure 8.19(b), but are sometimes displayed separately (e.g., bubble charts). Some versions support basic constraints on attributes (e.g., Ma/Op/Unique).

Relationships are typically restricted to binary associations only, which are shown as named lines connecting the entity types. Relationship names are read left-to-right and top-to-bottom. As with the Barker notation, a half-line or line end corresponds to a role in ORM.

Figure 8.19 Typical IE notation for (a) entity type and (b) entity type with attributes.

Optionality and cardinality settings are indicated by annotating the line ends. To indicate that a role is *optional*, a circle "O" is placed at the other end of the line, signifying a minimum participation frequency of 0. To indicate that a role is *mandatory*, a stroke "|" is placed at the other end of the line, signifying a minimum participation frequency of 1. After experimenting with different notations for a cardinality of "*many*", Finkelstein settled on the intuitive crow's foot symbol suggested by Dr. Gordon Everest.

In conjunction with a minimum frequency of 0 or 1, a stroke "|" is often used to indicate a maximum frequency of 1. With this arrangement, the combination "O|" indicates "*at most one*" and the combination "||" indicates "*exactly one*". This is the convention used in this section. However, different IE conventions exist. For example, some assume a maximum cardinality of 1 if no crows foot is used, and hence use just a single "|" for "exactly one". Clive Finkelstein uses the combination "O|" to mean "optional but will become mandatory", which is really a dynamic rather than static constraint—this can be combined with a crow's foot. Some conventions allow a crow's foot to mean that the minimum (and hence maximum) frequency is many. So if you are using a version of IE, you should check which of these conventions applies.

Figure 8.20 shows a simple IE diagram and its equivalent ORM diagram. With IE, as you read an association from left to right, you verbalize the constraint symbols at the right-hand end. Here, each employee occupies exactly one (at least 1 and at most 1) room. Although inverse predicates are not always supported in IE, you can supply them yourself to obtain a verbalization in the other direction. For example, "Each room is occupied by zero or more employees". As with the Barker notation, a plural form of the entity type name is introduced to deal with the many case.

The IE notation is similar to the Barker notation in showing the maximum frequency of a role by marking the role at the other end. But unlike the Barker notation, the IE notation shows the optionality/mandatory setting at the other end as well. In this sense, IE is like UML (even though different symbols are used). As discussed earlier, there are 16 possible constraint patterns for optionality and cardinality on binary associations. Figure 8.21 shows these cases in IE notation together with the equivalent cases in ORM.

An example using the different notation for IE used by Finkelstein is shown in Figure 8.22. The single bar on the left end of the association indicates that each computer is located in exactly one office. The circle, bar, and crow's foot on the right-hand role collectively indicate that each office must eventually house one or more computers. Although this "optional becoming mandatory" constraint has no counterpart in ORM and is unsupported by most IE modeling tools, it could be used to refine decisions on how to group fact types into implementation structures.

Figure 8.20 The IE diagram (a) is equivalent to the ORM diagram (b).

Figure 8.21 Equivalent constraint patterns in IE and ORM.

Figure 8.22 In Finkelstein's version, circle–bar means "optional becoming mandatory".

Some modeling tools support the IE notation for *n*:1, 1:*n*, and 1:1 associations but not *m:n* (many to many) associations. For such tools, 4 of the 16 cases in Figure 8.21 can't be directly represented. In this situation, you can model the *m:n* cases indirectly by introducing an "*intersection entity type*" with mandatory *n*:1 associations to the original entity types. For example, the *m:n* case with both roles optional may be handled by introducing the object type *C* as shown in Figure 8.23.

In IE this transformation loses the constraint that each *A*-*B* combination relates to at most one *C* (unless we drop to the relational level by using two foreign keys for *C*'s primary key—see IDEF1X later). In ORM, the constraint is captured conceptually as external uniqueness, making *C* a *coreferenced* object type—the transformation is a flatten/coreference equivalence (see Chapter 14).

As an example, the *m:n* association Person plays Sport can be transformed into the uniquely constrained, mandatory *n*:1 associations: Play is by Person; Play is of Sport. However, such a transformation is often very unnatural, especially if nothing else is recorded about the coreferenced object type. So any truly conceptual approach must allow *m:n* associations to be modeled directly.

Some versions of IE support an *exclusive-or constraint*, shown as a black dot connected to the alternatives. Figure 8.24(a) depicts the situation where each employee is allocated a bus pass or parking bay, but not both. The equivalent ORM schema is shown in Figure 8.24(b). Unlike ORM, IE does not support an inclusive-or constraint. Nor does it support simple or compound exclusion constraints.

Subtyping schemes for IE vary. Sometimes Euler diagrams are used, adding a blank compartment if needed for "Other". Sometimes directed acyclic graphs are used, possibly including subtype relationship names and optionality/cardinality constraints.

Figure 8.23 An *m:n* association remodeled as an entity type with *n*:1 associations.

Figure 8.24 An exclusive-or constraint in (a) IE and (b) ORM.

Figure 8.25 shows three different subtyping notations for partitioning Patient into MalePatient and FemalePatient. There is no formal support for subtype definitions or context-dependent reference. Multiple inheritance may or may not be supported, depending on the version.

Although far less expressive than ORM, IE does a good job of covering basic constraints. Its founder Clive Finkelstein is an amiable Aussie who is still actively engaged in the information engineering discipline. He developed a set of modeling procedures to go with the notation, extended IE to Enterprise Engineering, and incorporated work with the Extensible Markup Language (XML).

Figure 8.25 Some different IE notations for subtyping.

8.4 IDEF1X

In the 1970s, the U.S. Air Force began work on a program for Integrated Computer Aided Manufacturing (ICAM). This was the genesis of a family of *IDEF modeling languages*. The acronym "IDEF" originally denoted "ICAM DEFinition", but now stands for "Integration DEFinition", reflecting its possible use for exchanging information between different modeling languages. Rather than specifying one universal modeling language, the ICAM project defined the following languages for different tasks:

- IDEF0 activity modeling
- IDEF1 conceptual data modeling
- IDEF2 simulation modeling

Later, other languages were added, including:

- IDEF1X logical data modeling
- IDEF3 process modeling
- IDEF4 object-oriented software design
- IDEF5 knowledge engineering of enterprise ontologies
- $IDEF1X_{97}$ logical data modeling with object-oriented extensions

The name "*IDEF1X*" stands for "IDEF1 eXtended". Although based on the conceptual IDEF1 language, IDEF1X was changed to focus on logical data modeling. Although regretted by some, the introduction of IDEF1X effectively spelled the end of IDEF1. Over time, IDEF3 subsumed much of IDEF2 and, more recently, IDEF5. Nowadays, IDEF0 and IDEF1X are the most popular IDEF languages. Both are supported in a variety of CASE tools and are widely used in U.S. government sectors, especially defense. The IDEF3 language is also used, although to a lesser extent.

IDEF1X is a hybrid language, combining some conceptual notions (e.g., entity, relationship) with relational database constructs (e.g., foreign keys). It was accepted as a standard by the National Institute of Standards and Technology (NIST) in 1993 (NIST 1993). A proposed successor called $IDEF1X_{97}$ was approved in June 1998 by the IEEE-SA Standards Board (IEEE 1999). Also known as $IDEF_{object}$, this extended IDEF1X with object-oriented features to facilitate implementation in object-oriented databases and programming languages, while maintaining compatibility with IDEF1X for relational database implementation. This backward compatibility gives $IDEF1X_{97}$ one advantage over UML for adoption in US government sectors.

Nevertheless the future of $IDEF1X_{97}$ is uncertain, since its object-oriented extensions are less encompassing than those in UML. Although UML is yet to be widely used for database design, it is by far the most widely used language for designing object-oriented code, it is supported by many CASE tools, and it has been adopted as a standard by the International Standards Organization (ISO). At the time of writing, few practitioners or tool vendors have expressed interest in adopting $IDEF1X_{97}$. In contrast, the original (1993) version of the IDEF1X language continues to be very widely used for database design and is supported by many modeling tools. For these reasons, we restrict our attention to the original IDEF1X in the rest of this section.

To model facts, IDEF1X uses the three main constructs of ER: entities, attributes, and relationships. Unfortunately, the IDEF1X standard uses the term "entity" to mean "entity type", not "entity instance". Even more unfortunately, this misusage has been adopted by some practitioners. So you may have to use the word "instance" more than you want to, simply to ensure that you are not misunderstood. Each entity type has one or more attributes that are based on domains. A domain is a named set of data values of the same data type. Relationships are restricted to binary associations between entity types.

Table 8.1 Three different views in IDEF1X.

ER view	Entity types: No identification scheme or keys defined No need to specify any attributes No attribute constraints (e.g., mandatory) Relationships: Many:many relationships are allowed Identifying/nonidentifying distinction is not made
Key-based view	Include at least all the key-based attributes Classify connection relationships as identifying or nonidentifying
Fully attributed view	Include all attributes

IDEF1X allows models to be developed in phases using at least *three views*: ER view, key-based view, and fully attributed view. The essential differences among these views are summarized in Table 8.1.

The *ER view* may be used early in the analysis but is very inexpressive compared with true ER. No identification schemes are specified for entity types, and no attributes need be declared. If attributes are specified, no constraints (e.g., mandatory) are declared for them.

An ER view in IDEF1X is basically an incomplete sketch of an ER model. In principle, it could be refined into a true ER model. Unfortunately, instead of carrying out the refinement at the conceptual level, IDEF1X drops down to the logical level to add the extra detail. Any ER view must be resolved into a key-based view and ultimately a fully attributed view to complete the model. Key-based views and fully-attributed views are similar to a relational model in many respects. Some CASE tools that support IDEF1X do not support its ER view, and we ignore the ER view from this point on.

In key-based and fully attributed views, IDEF1X entity types are basically relational tables, and the "relationships" are foreign key to primary key references. Recall that a foreign key is a set of one or more attributes whose values (possibly composite) must occur as a value of some primary key. The source or referencing entity (type) is called the *child* and the target or referenced entity (type) is called the *parent*. The relationships may be assigned forward and inverse readings as if they were conceptual associations or subtype links. However, they ultimately represent subset constraints. The actual facts are all stored in attributes, as in the relational model.

An entity type is classified as identifier independent or identifier dependent. An entity type is *identifier independent* if and only if its identifier does not depend on other entity types. In other words, its identification scheme does not include a conceptual relationship to another entity type. In terms of keys, this means that *an entity type is identifier independent if and only if its primary key has no component that is a (complete) foreign key*—its primary key may, however, contain an attribute that is just part of a foreign key. Otherwise the entity type is *identifier dependent*. The terms "identifier independent" and "identifier dependent" are often shortened to "independent" and "dependent". This notion of "independent" has nothing to do with the notion of independent object types in ORM.

Figure 8.26 An independent entity type (a) and a dependent entity type (b).

In IDEF1X, an independent entity (type) is depicted as a hard rectangle (square corners), with its name written above, as shown in Figure 8.26(a). A dependent entity type is depicted as a soft rectangle (rounded corners), as shown in Figure 8.26(b).

Attributes of an entity type are listed inside its rectangle. *Primary key* attributes appear in the top compartment. These provide the entity type's primary identification scheme. *Alternate keys* are marked by appending "(AK*n*)", where $n > 0$. These provide an alternate identification scheme. If an alternate key is composite, each of its components has the same value for *n*. Different alternate keys must have different values for *n*. *Foreign keys* are indicated by appending "(FK)". Attributes are *mandatory by default*. To show that an attribute is *optional*, "(O)" is appended. All attributes are single valued.

In Figure 8.27, empNr is the primary key of Employee. The bldgNr, roomNr pair provides the composite primary key for Room. The social security number attribute (ssn) is an alternate key for Employee. In Employee, the pair bldgNr, roomNr is a foreign key (referencing the primary key of Room). In Room, bldgNr is a foreign key (referencing the primary key of a Building entity type not shown here). All attributes are mandatory except for fax and phoneNr, which are marked optional.

In all views, *relationships* are binary only. In the ER view, relationships are conceptual associations and may be "non-specific" (many-to-many). In key-based and fully attributed views, each "relationship" is either a "*connection*" (*foreign key to primary key reference*) or a *categorization* (*subtype link*). Connections are "*specific*" (many to one) binary associations between a child entity type and parent entity type, where each child has at most one parent.

Figure 8.27 Key (primary, alternate, and foreign) and optionality indicators.

Figure 8.28 An identifying connection "relationship".

Connection relationships are either identifying or nonidentifying. In principle, an *identifying relationship* is an association that is conceptually used in the identification scheme of the child. In actuality, it is simply a *reference from a foreign key in the child's primary key to the parent's primary key*.

In a *nonidentifying* relationship, the child *nonkey* attributes include a foreign key that references the parent's primary key.

All connection relationships are denoted as a named line with a *dot* "•" *at the child end.* An *identifying* relationship is shown as named, *solid line*, as shown in Figure 8.28. Here *a* and *b* are attributes (possibly compound).

With an identifying connection, each child entity is associated with *exactly one* parent instance. This is because each child instance has a nonnull value for the foreign key (in the child's primary key) that references the parent. By definition, the child must be identifier dependent, as shown by the soft rectangle. The parent is identifier-independent (as shown here) unless it is made dependent by some other relationship.

IDEF1X allows both forward and inverse predicate readings for relationships. These may be written together, as shown in Figure 8.29(a), or on different sides of the relationship line, as shown later in Figure 8.30(a).

Figure 8.29 The IDEF1X schema (a) is equivalent to the ORM schema (b).

The *forward reading of a specific connection relationship is always toward the child*, and hence *toward the dot* "•". This forward or "parent-perspective" reading must be supplied. The inverse or "child-perspective" reading, if supplied, is toward the parent (away from the dot). If both readings are supplied, a slash "/" is appended to the forward reading. In Figure 8.29(a) the forward reading is Building contains Room and the inverse reading is Room is in Building. Because the child here is on the left, the forward reading is right to left. For non-specific (*m:n*) relationships however, the forward reading is always read from left to right (or top to bottom if the relationship line is vertical). Recall that *m:*n relationships exist only at the ER view level.

The equivalent ORM diagram in Figure 8.29(b) makes it clear that the association Room is in Building is part of the composite reference scheme for Room. In the IDEF1X diagram, this association is actually depicted by including bldgNr as part of the key for Room, and the connection "relationship" shows the foreign key reference. We can partially lift the IDEF1X discussion to a conceptual level by talking of the parent's primary key *migrating* to the child (see Figure 8.30).

Although not a legal IDEF1X diagram, Figure 8.30(a) helps portray Room as being partly identified by its room number and partly by its relationship to Building. Compare this with the Barker notation depiction in Figure 8.9. At this level, the association Room is in Building is conceptual.

Now imagine a copy of Building's primary key migrating to complete the primary key of Room, as in Figure 8.30(b). At this stage, we have dropped to the logical level. The conceptual relationship between Room and Building is now depicted by Room's bldgNr attribute. We still need to indicate that any value for this attribute must also occur as a primary key value of Building, and this foreign key reference is what the connection line now represents. We removed the "relationship" name to stress this point. You can now think of the connection line as a foreign key arrow.

For identifying relationships, each child (instance) is associated with exactly one parent (instance), as shown by a solid line with no adornments. By default a parent is associated with zero or more children, as shown by an unadorned dot "•" at the line end. If you are familiar with UML, you can think of this as equivalent to "*". You can override this default child cardinality by adorning the dot with a *cardinality constraint* mark. The main cases are shown in Figure 8.31 along with their ORM counterpart. Here "P" indicates a positive number (at least 1), and "Z" indicates zero or 1 (at most 1). A single number (e.g., 3) indicates exactly that number. You can also indicate ranges. For example, "2-5" indicates at least 2 and at most 5.

Figure 8.30 Building's primary key "migrates" to become a foreign key in Room.

Figure 8.31 Main child cardinality cases for identifying relationships.

To make sense of these cardinality constraints, think of them as applying to the conceptual relationship that existed before the key was migrated to the child (cardinality constraints on a foreign key reference are rather pointless). The "Z" mark is unintuitive for "at most 1", and in IDEF1X$_{97}$ an alternative notation for this constraint is to use a hollow dot "○".

A *nonidentifying relationship* is shown as a named, *dashed line*. Child and parent are independent unless made dependent by some other relationship. If the dashed line is unadorned, each child (instance) is associated with *exactly one parent* (instance), as in Figure 8.32(a). If a *diamond* is added at the parent end, each child is associated with *at most one parent*, as in Figure 8.32(b). The corresponding ORM diagrams are also shown.

In a nonidentifying relationship, the child *nonkey* attributes (those not in the primary key) include a foreign key that references the parent's primary key. If the foreign key is mandatory (each component is nonnull) then each child instance is associated with exactly one parent instance. This is referred to as a *mandatory* relationship, but actually means that the child role of the conceptual relationship that (before migration) gave rise to the foreign key reference is mandatory.

Figure 8.32 Nonidentifying connection "relationships".

Figure 8.33 "Mandatory" (a) and "optional" (b) non-identifying "relationships".

Figure 8.33(a) depicts this situation in more detail. Here *a* and *b* are simple or composite attributes. If the foreign key is optional (its components may all be null) then each child instance is associated with at most one parent instance. Figure 8.33(b) depicts this situation. In IDEF1X this is referred to as an *optional* relationship.

In both mandatory and optional cases, each instance of the child's foreign key that has no null components must occur as an instance of the parent's primary key. This is the default meaning of a foreign key constraint in SQL. For the mandatory case of course, there can be no null components.

An example of a mandatory, nonidentifying connection relationship is shown in Figure 8.34(a). Here each employee is identified by an employee number, must have a name, and must work for a department. Each department is identified by its code and must have a unique name.

The fact (conceptual relationship) that a given employee works for a given department is stored by instantiating the empNr and deptCode attributes of Employee (e.g., empNr = 101, deptCode = 'HR'). The connection "relationship" is simply a foreign key reference indicating that each department code of an employee must be a department code of a department.

Figure 8.34 Same example in (a) IDEF1X, (b) IE, (c) Relational and (d) ORM.

Figure 8.35 The room schedule example in (a) IDEF1X and (b) ORM.

The same example is shown in IE notation in Figure 8.34(b) and in ORM notation in Figure 8.34(d). Here the association Employee works for Department is depicted conceptually, rather than by including deptCode as an attribute of Employee. This would also be the case for the Barker notation. Figure 8.34(c) depicts the situation in a relational notation. Here facts about who works for what department are stored in the Employee table, which includes a deptCode attribute referencing the Department table. If you compare this with the IDEF1X notation in Figure 8.34(a), it should be clear that they are just two notational variations of the same structure. In key-based and fully-attributed views, the IDEF1X notation is essentially a logical notation rather than a conceptual one.

As another example of non-identifying relationships, the room scheduling example considered in Sections 1.2 and 8.2 is modeled in IDEF1X in Figure 8.35(a). The ORM model for this situation is reproduced in Figure 8.35(b), without the fact populations. The primary key and alternate key constraints in the IDEF1X model correspond to the uniqueness constraints in the ORM model. Although the IDEF1X schema does capture all of the constraints, it is a logical rather than a conceptual representation, and unlike ORM it does not facilitate validation by verbalization and population.

Figure 8.36 shows a more complex IDEF1X example involving a compound foreign key relationship corresponding to a pair-subset constraint in the ORM schema.

Figure 8.36 A complex example in (a) IDEF1X and (b) ORM.

Figure 8.37 An optional, nonidentifying "relationship".

A simple example of the optional, nonidentifying "relationship" pattern in Figure 8.33(b) is shown in Figure 8.37. Here an employee optionally has an office. Each office is a room with a composite identification scheme. Room's relationship with Building is omitted here.

A more complex example is shown in Figure 8.38. This is similar to the example in Figure 8.36, except that it is now optional for one to like a movie or to have a favorite movie.

In an ER view or "entity level diagram" all entity types are depicted by hard rectangles, and non-specific (*m:n*) relationships are permitted. Each non-specific relationship line must end in a dot, possibly adorned by a cardinality mark. If unadorned, a cardinality of exactly one is assumed. In moving to a key-based or fully attributed view, each *m:n relationship* must be resolved into an "*intersection entity type*" with two *n*:1 relationships. Figure 8.39 shows an example in both IDEF1X and ORM.

An intersection entity type is sometimes called an "*associative entity type*". Although conceptually this corresponds to a coreferenced object type in ORM, in IDEF1X the identifying attributes must be "migrated" to become foreign keys inside the primary key of the intersection entity type, which therefore must be identifier dependent, appearing as a soft rectangle. This foreign key depiction effectively lowers the representation to the relational level.

Although this resolution is often unnatural, it does allow attributes to be added to what was once a relationship. For example, we might wish to record a skill level for Play. In ORM and UML, this objective can also be achieved by simply permitting a relationship to be objectified (as an objectified association or association class), but this conceptual alternative is not allowed in IDEF1X.

Figure 8.38 A more complex example in (a) IDEF1X and (b) ORM.

Figure 8.39 (a) In IDEF1X, *m:n* relationships must be resolved into *n*:1 relationships. (b) The same example in ORM.

If the *m:n* relationship is a ring association, role names may be prepended to the names of the migrated foreign keys to distinguish them. For example, in Figure 8.40(a), the role names "subPartNr" and "superPartNr" are prepended to "partNr" to disambiguate the primary key attributes of Containment.

Although the IDEF1X standard uses a dot after the role name, this dot notation is not supported in some CASE tools. However the distinction can be easily achieved without the dot by using just a role name (e.g., "subPartNr" can be used instead of "subPartNr.partNr").

In either case, the role name should be shown as a noun phrase that identifies the attribute even without appending the name of the referenced primary key. In ORM, a more natural way to model this situation is to objectify the containment association, as shown in Figure 8.40(b).

In IDEF1X, an entity type may be classified into one or more clusters of mutually exclusive *categories* (subtypes). The supertype is called the *generic* entity (type). Each subtype inherits the primary key of its supertype, and hence is identifier dependent. The subtype linkage is called a "*categorization relationship*".

This version of subtyping is very restricted compared with ORM subtyping, which allows overlapping types within a "cluster" as well as context-dependent reference. In effect, the subtyping approach of IDEF1X treats entity types as tables, and the "categorization relationships" are simply foreign key references. In sharp contrast, ORM subtyping is purely conceptual, and different mapping strategies may be chosen at implementation time (see Chapter 11).

However, there are two aspects of the subtyping approach in IDEF1X that express more than simple foreign key references. These are the completeness and discriminator declarations. A subtype link is shown as a line with an underlined circle at the supertype end. A *single underline* means that the cluster is *incomplete* (i.e., the supertype is more than the union of its subtypes). This means that the population of the supertype may contain instances not present in any of its subtypes.

Figure 8.40 Role names may be used in (a) IDEF1X and (b) ORM.

For example, in Figure 8.41(a), the subtypes TallPerson and ShortPerson are mutually exclusive, but not collectively exhaustive. A person may be of medium height without being classified as either tall or short. In contrast to IDEF1X, ORM often uses derived subtypes, which must be defined in terms of roles played by their supertype(s). Figure 8.41(b) shows one way of modeling this situation in ORM.

Here a person's height is used to determine whether he or she is a member of TallPerson or ShortPerson. The subtype definitions and the lack of a value constraint on Height indicate that these subtypes are exclusive but not exhaustive. This can also be depicted explicitly by an implied exclusion constraint as shown. As an alternative to using height, you could attach the optional and exclusive unaries "is tall" and "is short" to Person and supply the obvious definitions.

In IDEF1X, a *double underline* at the supertype end means that the cluster is *complete*. This means that the supertype is the union of its subtypes. Since members of an IDEF1X cluster are also exclusive, this means that we have a partition. In Figure 8.42(a), for example, Person is partitioned into MalePerson and FemalePerson.

Figure 8.41 An incomplete and exclusive subtype cluster in (a) IDEF1X and (b) ORM.

Figure 8.42 A complete, exclusive subtype cluster in (a) IDEF1X and (b) ORM.

A mandatory attribute of a supertype may be used as a *discriminator* and written next to the circle to indicate the basis for the subtyping. In Figure 8.42(a) for example, gender is used as a discriminator to classify Person into MalePerson and FemalePerson. The corresponding ORM schema is shown in Figure 8.42(b). The partition constraint shown explicitly here is implied by the subtype definitions and other constraints.

Well that pretty much covers the IDEF1X notation. Although it is a widely used standard, especially in the defense industry, we find it less suitable than the Barker or IE notations for conceptual modeling. To capture any detail or even basic constraints in IDEF1X, you have to resolve the model into what is essentially a relational model. Moreover, the IDEF1X set of concepts and its notation are unnecessarily complex and forgettable. For conceptual analysis and validation, it also suffers from the weaknesses of an attribute-based approach.

Despite such drawbacks, IDEF1X can be used effectively by experienced modelers, who often associate connection relationships with the conceptual relationships from which they are migrated. Even for such experienced modelers, however, the best way to model in IDEF1X is to first do an ORM model and then map it to IDEF1X, expressing any additional ORM constraints as supplementary text or implementation code. Some ORM modeling tools can perform this mapping automatically.

8.5 Mapping from ORM to ER

Fully attributed IDEF1X schemas are essentially relational schemas displayed in a different notation. For this reason, Microsoft's ORM Source Model solution treats IDEF1X as simply a different relational style. If you are using that tool, you can generate IDEF1X schemas from ORM schemas by automatically generating the relational schema, setting the drawing style to IDEF1X, and then noting any extra ORM rules as textual annotations.

Table 8.2 ERmap Procedure.

Step	*Action*
1	Preprocess the ORM schema (actually or mentally) 1.1 Replace objectified associations by coreferenced object types 1.2 Binarize *n*-ary associations by co-referencing 1.3 Binarize any sets of exclusive unaries
2	Model selected object types as entity types, and map a selection of their *n*:1 and 1:1 associations as attributes
3	Map remaining unary fact types to Boolean attributes or subtypes
4	Map remaining fact types to associations
5	Map ORM constraints to ER graphic constraints, textual constraints, or notes
6	Retain subtypes, but map subtype definitions to textual constraints
7	Map derived fact types to textual derivation rules, and map semi-derived fact types to attributes/associations plus textual derivation rules

If instead you need to manually generate IDEF1X schemas from ORM, follow the relational mapping (Rmap) procedure discussed in Chapter 11 to obtain a relational schema and then redraw it in IDEF1X.

Mapping from ORM to a true ER depends on the ER dialect (e.g., does it support *n*-ary associations, multivalued attributes, any form of objectification, or derived attributes/associations?). Industrial ER dialects typically support none of these options. The *ERmap* procedure in Table 8.2 provides basic guidelines for mapping ORM to Barker ER or IE.

We now illustrate the steps in this procedure with examples using the Barker ER notation. Step 1.1 replaces objectified associations by coreferenced object types. For example, if the fact type Person wrote Paper is objectified as Writing, replace this by Writing is by Person and Writing is of Paper, as shown in Figure 8.43.

Step 1.2 binarizes *n*-ary predicates by co-referencing. For example, the ternary Room at HourSlot is booked for Activity is replaced by forming a coreferenced entity type Room-Booking, as shown in Figure 8.44.

Figure 8.43 Step 1.1: replace objectifications by coreferenced object types.

Figure 8.44 Step 1.2: replace *n*-ary fact types by coreferenced object types.

Step 1.3 binarizes any sets of exclusive unaries. For example, the Employee is male and Employee is female fact types are replaced by the single binary Employee is of Gender, with relevant constraint transforms, as shown in Figure 8.45.

Figure 8.45 Step 1.3: replace any set of exclusive unaries by a binary fact type.

In step 2, we decide which object types to model as entity types and which *n*:1 and 1:1 ORM associations to remodel as attributes. Typically, entity types that play functional fact roles are retained as entity types. In rare cases, value types that are independent or play explicit mandatory, functional fact roles map to entity types. Functional binary (*n*:1 and 1:1) associations from an entity type *A* to a value type B, or to an entity type *B* about which you never want to record details, usually map to an attribute of *A*. If you have specified role names, these can usually be used as attribute names, with the object type name becoming the attribute's domain name.

The mapping in Figure 8.46 illustrates several of these step 2 considerations, as well as step 5 (map ORM constraints to ER graphic constraints, textual constraints, or notes).

Figure 8.46 Step 2: map selected *n*:1 and 1:1 associations to attributes.

Figure 8.47 Step 3: map unaries to Boolean attributes or subtypes.

The "{M, F}" annotation is a textual note to capture the value constraint on gender codes. The "{U1}" annotation is a textual note to indicate a uniqueness constraint on the ssn attribute. Neither of these textual annotations is part of the Barker ER notation.

In step 3 we map unaries to Boolean attributes or to subtypes. The example in Figure 8.47 assumes a closed world interpretation for the unary. If this is not the case, the is-smoker attribute is made optional and an UnknownSmoking subtype is added. Open/closed world aspects are discussed in more detail in Chapter 10.

In step 4, the remaining fact types are mapped to associations. Any *m:n* associations should remain that way, unless the target notation doesn't support them (e.g., IDEF1X). In that case, you can replace the association by an intersection entity type as discussed earlier.

In the example shown in Figure 8.48, the *n*:1 fact type is retained as an association because it relates two entity types that will remain as such in the mapping. Even if the *m:n* association did not apply, we would normally retain Country as an entity type, since now or later we are likely to record details for it (e.g., country name).

If an *m:n* association involves a value type, and the ER dialect does not support multivalued attributes, map the value type to an entity type. Figure 8.49 provides an example. Otherwise, where possible, transform the *m:n* association into multiple *n*:1 associations (e.g. Employee has PhoneNr1; Employee has PhoneNr2, etc.). Conceptual schema transformations like this are discussed in depth in chapter 14.

In step 5, the simplest constraints in ORM usually map in an obvious way to multiplicity (mandatory and/or cardinality) constraints, as illustrated earlier. The more complex ORM constraints have no counterpart in ER, so you need to record these separately in textual form. Table 8.3 summarizes the main correspondences.

Figure 8.48 Step 4: map remaining fact types to associations.

Figure 8.49 Step 4: map remaining fact types to associations.

Table 8.3 Mapping main ORM graphic constraints to ER (for use in step 5).

ORM constraint	*ER*
Internal UC	Maximum multiplicity of 1, or {Un}
External UC	Composite primary id (Barker) or textual constraint
Simple mandatory	Minimum multiplicity of 1, or textual constraint
Inclusive-or	Textual constraint (unless within exclusive-or)
Frequency	Multiplicity or textual constraint
Value	Textual constraint
Subset and Equality	Textual constraints
Exclusion	Barker exclusive arc, or textual constraint
Ring constraints	Textual constraint

In step 6, subtypes are mapped to subtypes, adding relevant constraints if needed. Subtype definitions are handled with discriminators and/or textual constraints. If needed, introduce an "Other" subtype to complete a partition. If needed, use multiple diagrams (e.g., when multiple categorization schemes apply to the same supertype). Figure 8.50 shows a simple example. The braced value constraint and footnoted constraints are not part of the Barker ER notation.

In step 7, we map derived and semiderived fact types. Industrial ER typically has no graphical support for derivation, so the derivation rules are captured as textual notes. Figure 8.51 provides a simple example. With these few hints, and the examples discussed, you should now have enough background to do the mapping manually. The mapping procedure is fairly boring and is best automated with a CASE tool.

Figure 8.50 Step 6: map subtyping.

Figure 8.51 Step 7: map derivation rules.

For mapping to IDEF1X, the relational mapping procedure discussed in Chapter 11 may be used essentially as is, treating the relationships as foreign key references and making the relevant notational variations.

Exercise 8.5

1. Model the following application domain in your preferred version of ER or IDEF1X. We suggest that you do an ORM model first, but that's up to you.

   A video store has a library of videotapes that may be lent to customers. Six extracts are shown from the information system used by the store. Several aspects have been simplified or removed to reduce the size of this problem. For example, addresses are shown as a single entry, and financial aspects are ignored. Data about a movie may be recorded before the store obtains a videotape of it. Each videotape contains a copy of exactly one movie, and is either purchased or leased from another supplier. The status or condition (Good, OK, Poor, X) of each tape is noted on a regular basis. A status of "X" indicates that the tape is excluded from the list of tapes that may be borrowed.

Movie	*Title*	*Category*	*Copy Nr*	*Purchase date*	*Lease expiry*	*Status*
AP2K	Apocalypse 2000	R	1	01/10/2000		Good
BMF	Batman Forever	M				
CJ	City of Joy	MA	1	07/01/2000		OK
			2		01/01/2001	X
DS	Donovan Sings	G	1	01/03/1999		Good
			2	01/03/1999		OK
GHST	Ghost	M	1	06/06/1991		X
			2	07/10/1991		Poor
			3	07/10/1991		X
GQ	Galaxy Quest	PG	1	01/10/2000		OK
			2		07/01/2001	Good
MTX	The Matrix	R	1	11/11/1999		Good
			2	11/11/1999		Good
MTX2	The Matrix 2	R				

For excluded video tapes, a record is kept to indicate the reason for their exclusion and whether they are written off for taxation purposes. Here is a sample extract.

Movie	*CopyNr*	*Comment*	*Written off?*
CJ	2	faulty	Y
GHST	1	faulty	N
GHST	3	stolen	Y

The following table records details about the categories in which movies are classified.

Category	*Description*
G	For General exhibition
PG	Parental Guidance recommended for persons under 15
M	Recommended for Mature audiences 15 years and over
MA	For Mature Adults: restrictions apply to persons under the 15 years age
R	Restricted to adults 18 years and over

For the current calendar year, a record is kept of which movies are the best sellers.

Month	*Rank*	*Movie*	*Title*
1	1	MTX	The Matrix
	2	AP2K	Apocalypse 2000
	3	DS	Donovan Sings
	…	…	…
2	1	MTX	The Matrix
	2	GQ	Galaxy Quest
	2	DS	Donovan Sings
	4	AP2K	Apocalypse 2000
	…	…	…

Loans are charged to customers, who are identified by the number on their video store-card presented at the time of borrowing. The combination of name and address is unique for a customer. Customers who indicate a phone number must indicate when they may be contacted on that phone (D = day only; N = Night only; D&N = Day and Night). Customer details may be recorded before they take out any loan.

Customer	*Name*	*Address*	*Phone*	*Call period*
1	Frog F	5 Ribbit Rd, Bellevue	425 555 7000	N
2	Jones E	3 Sun Ave, Spokane		
3	Smith J	520 Pike St, Seattle	206 555 6789	D
4	Jones E	520 Pike St, Seattle	206 555 6701	D
5	Frog K	5 Ribbit Rd, Bellevue	425 555 7000	D&N

Each loan is identified by a loan number and may involve more than one tape. Customers may borrow tapes as often as they like and may even borrow the same tape more than once on the same day (after returning it). The rental of a video tape within a loan is called a LoanItem. Although return data for loan items may be derived from video tape returns and other data, to simplify the problem you can ignore this.

Loan date	*Loan nr*	*Customer*		*Videotape*		*Date returned*
		Nr	*Name*	*Movie*	*Copy*	
10/01/00	9001	2	Jones E	GQ	1	?
				MTX	1	10/02/2000
	9002	3	Smith J	GQ	2	10/01/2000
10/02/00	9003	3	Smith J	MTX	1	10/03/2000
				DS	2	10/03/2000
				GQ	2	10/03/2000
	9004	4	Jones E	GHST	2	?
10/03/00	9005	1	Frog F	AP2K	1	10/03/2000
				MTX	1	?

2. The following ORM schema models information about university academics. As a challenge exercise, map this to your preferred ER or IDEF1X notation, including some comments to cater for advanced constraints and multiple inheritance. You may wish to delay this until you have studied the relational mapping procedure in Chapter 11.

8.6 Summary

The *Entity Relationship* (ER) modeling approach was originated by Peter Chen in the 1970s and allows facts to be expressed via relationships (e.g., Person was born in Country) or attributes (e.g., birthdate). Of the dozens of different ER versions in existence, the most widely used are the Barker notation and Information Engineering (IE). The popular IDEF1X approach is often referred to as a version of ER, but is actually a mixture of ER and relational approaches, with the emphasis on relational.

The *Barker notation* represents entity types as named, soft rectangles with a list of one or more attributes. A "*" or "•" indicates that an attribute is mandatory, and a "○" indicates that the attribute is optional. All relationships are binary, and are shown as named lines. A solid half-line denotes a mandatory role, and a dotted half-line denotes an optional role. A crow's foot at the end of the line indicates the cardinality "many", and its absence indicates "one". A hash "#" in front of an attribute indicates it is, or is a part of, the primary identifier. A bar "|" across one end of a relationship indicates that the relationship is a component of the primary identifier for the entity type at that end.

An exclusive arc across roles indicates that the roles are mutually exclusive; if the role lines are solid, we have an exclusive-or constraint. Subtype partitions are denoted by Euler diagrams, placing the subtypes inside the supertype.

In the *Information Engineering* approach, entity types are displayed as named rectangles, with a list of attributes. Relationships are binary and are denoted by named lines. A crow's foot at the end indicates "many", a stroke "|" indicates "one", and a circle "○" indicates optional. Two strokes "||" indicates "exactly one". Some IE versions depict an exclusive-or constraint as a black dot joining relationship lines. Different subtype notations exist for IE, some using Euler diagrams and some "is a" relationship lines.

IDEF1X models may be viewed at three levels. In a high level ER view, *m:n* (or "nonspecific") relationships may be shown directly, but these must be resolved into an intersection entity type with two *n*:1 relationships as the model is refined into key-based or fully attributed views. An entity type is identifier dependent if its primary key includes a foreign key and is shown as a named, soft rectangle. Otherwise it is identifier independent and is shown as a named, hard rectangle. Attributes are listed inside the rectangle, with the primary key in the top compartment. Alternate keys are denoted by appending "(AK*n*)" and foreign keys by appending "(FK)". An attribute is mandatory unless followed by "(O)".

Connection relationships are foreign key references from the child to the parent and are shown as a named line with a dot "•" at the child end. For a specific connection relationship, its forward name is always read toward the dot. If an inverse name is added, a slash "/" is appended to the forward reading. For nonspecific relationships, the forward reading is left to right (or top-to-bottom if the line is vertical).

A line end has a cardinality of 1 unless annotated. A dot indicates "0 or more" but can be strengthened by adding "P" (1 or more), "Z" (0 or 1), or "*n*" (exactly *n*).

A foreign key reference starting from a primary key attribute is an identifying relationship and is shown as a solid line. A foreign key reference starting from a nonkey attribute is a nonidentifying relationship, and is shown as a dashed line; in this case, a diamond at the parent end indicates that each child is associated with at most one parent.

An entity type may be classified into one or more clusters of mutually exclusive categories. Subtype links are depicted as categorization relationships with a circle at the supertype end. The cluster is incomplete or complete according as to whether the circle has a single underline or double underline, respectively.

ER or IDEF1X models are best developed by mapping them from ORM models, and noting any additional ORM constraints as comments. The main guidelines for mapping from ORM to ER are summarized in Table 8.2.

Chapter Notes

There are dozens of ER notations in existence, and minor variations exist in CASE tool implementations of the main notations discussed in this chapter. Dr. Peter Chen is still active in the field, and is on the steering committee for the international Entity Relationship conferences, which have been held yearly since 1981, and have been broadened in scope to include all forms of conceptual modeling. The proceedings of the ER conferences are published by Springer (*www.springer.com*).

Clive Finkelstein, the "father of information engineering", is also still active in the field. Finkelstein (1989) discusses the basic concepts and history of IE. Finkelstein (1992) provides practical advice on using IE, as well as a treatment of entity dependency. For a short, authoritative overview of the IE method, see Finkelstein (1998). A treatment of his work on using XML for building corporate portals, can be found in Finkelstein and Aiken (2000). His Web site can be accessed at *www.ies.aust.com/~ieinfo/*. For a look at the IE approach used by James Martin, see Martin (1993). Martin's more recent books tend to use the UML notation instead. In practice, however, IE is still used far more extensively for database design than UML, which is mostly used for object-oriented code design.

An overview of the IDEF family of languages is presented in Menzel and Mayer (1998). For further details on IDEF1X, see Bruce (1992) and NIST (1993). The IDEF1X_{97} standard is described in IEEE (1999).

This page intentionally left blank

9 Data Modeling in UML

9.1 Introduction
- Overview of UML

9.2 Object-Orientation
- Class diagrams and object-orientation
- Identification schemes

9.3 Attributes
- Single-valued attributes
- Multivalued attributes

9.4 Associations
- Binary and *n*-ary associations
- Association classes
- Qualified associations

9.5 Set-Comparison Constraints
- Subset constraints
- Exclusive-or constraints

9.6 Subtyping

9.7 Other Constraints and Derivation Rules
- Other constraints
- Derivation rules

9.8 Mapping ORM to UML

9.9 Summary

9.1 Introduction

Although semantic approaches to information modeling appeared in the early seventies, no single approach has yet achieved universal adoption. By and large, the history of information systems modeling has been characterized by a plethora of techniques and notations, with occasional religious wars between proponents of different approaches. Each year, many new approaches would be proposed, leading to groans from academics who were charged with teaching the state of the art. This is referred to as the "yama" (Yet Another Modeling Approach!) or "nama" (Not Another Modeling Approach!) syndrome. Figure 9.1 pictures this as a mountain of modeling methods, piled on top of one another, which nicely ties in with the Japanese meaning of "yama" (mountain), depicted as a kanji that is high in the middle and low on the ends.

While diversity is often useful, the modeling industry would benefit if practitioners agreed to use just a few standard modeling approaches, individually suited for their modeling scope, and collectively covering the tasks needed to model a wide variety of applications. This would improve communication between modelers and reduce training costs, especially in an industry with a high turnaround of employees.

Recently, the rapid rise of the Unified Modeling Language (UML) has been accompanied by claims that UML by itself is an adequate approach for modeling any software application. Some UML proponents have even been so bold as to claim that "the modeling wars are over—UML has won". This claim has been strongly rejected by several experienced data modelers, including Dave Hay, who argues that "there is no such thing as 'object-oriented analysis'" (Hay 1999a), only object-oriented design, and that "UML is … not suitable for analyzing business requirements in cooperation with business people" (Hay 1999b).

To date, UML is mainly used in industry for designing object-oriented program code. Although it can be used for designing databases, UML has so far had little success in displacing other approaches such as ER for this purpose. Given UML's object-oriented focus, and the dominance of relational DBMSs, this is perhaps not surprising. Nevertheless, UML is a very important language that could well become popular for database design in the future.

Figure 9.1 "Yama" (Yet Another Modeling Approach) or Japanese for "Mountain".

Initially based on a combination of the Booch, Object Modeling Technique (OMT), and Object-Oriented Software Engineering (OOSE) methods, UML was further developed by a consortium of companies and individuals working within the Object Management Group (OMG). It includes adaptations of many other techniques (e.g., Harel's state charts) and is continually being refined and extended.

Version 1.1 of UML was adopted in November 1997 by the OMG as a language for object-oriented analysis and design. Versions 1.2, 1.3, 1.4, and 1.5 were approved in 1998, 1999, 2001, and 2003, respectively. Version 1.4.2 was accepted as a standard by the International Standards Organization (ISO). A major revision (2.0) was recommended in 2004, comprising Infrastructure and Superstructure specifications, plus related specifications on the Object Constraint Language (OCL) and Diagram Interchange. At the time of writing (2007), UML 2 has been updated to version 2.1.1[1]. When using a UML tool, be aware that vendor support typically lags behind the latest OMG adopted version (e.g., some tools are still at UML 1.2).

As discussed later, the UML metamodel and notation have inconsistencies, with some unresolved problems being fundamental. Despite these issues, UML is the closest thing to a de facto standard in industry for object-oriented software design, and hence is worthy of study.

The UML notation is really a set of languages rather than a single language. It includes a vast number of symbols, from which various diagrams may be constructed to model different perspectives of an application. The nine main diagram types in UML 1.5 are Use Case (use case diagram); Static Structure (class diagram, object diagram); Behavior (statechart, activity diagram); Interaction (sequence diagram, collaboration diagram); and Implementation (component diagram, deployment diagram). UML 2 extended these to 13 diagram types, as set out in Table 9.1.

Table 9.1 The 13 predefined diagram types in UML 2.

Structure	Class Object Component Deployment Package Composite Structure	
Behavior	Use case State Machine Activity	
	Interaction	Sequence Collaboration Interaction Overview Timing

[1] http://www.omg.org/technology/documents/formal/uml.htm

Some of these diagrams (e.g., collaboration diagrams) are useful only for designing object-oriented program code. Some (e.g., activity diagrams and use case diagrams) can be useful in requirements analysis, and some (e.g., class diagrams) have limited use for conceptual analysis and are best used for logical design.

The UML specification provides syntax and semantics for these diagram types, but not yet a process for developing UML models, other than to suggest that model development should be use case driven, iterative, and architecture centric. Various companies promote their own modeling process for UML, such as the Rational Unified Process (RUP).

Although all the UML diagram types are worth studying, this book focuses on information modeling for databases. This chapter addresses data modeling in UML, so considers only the static structure (class and object) diagrams. Class diagrams are used for the data schema. Object diagrams provide a limited means to discuss data populations. Some other UML diagram types are discussed in Chapter 15.

Like ER, UML uses attributes. As discussed in earlier chapters, attributes are great for logical models, but are best modeled as relationships when performing conceptual analysis, since this facilitates validation and minimizes the impact of change. For such reasons, we believe the best way to develop UML data models is to first do an ORM model and then map it to UML. Since ORM will be used to clarify the data modeling concepts in UML, to gain the full benefits of this clarification you should be familiar with the ORM concepts covered in earlier chapters.

No language is perfect, ORM included. Overall, UML provides a useful suite of notations for both data and process modeling, while ORM is currently focused on data modeling only.

9.2 Object-Orientation

UML facilitates object-oriented (OO) code design because it covers both data and behavioral modeling, and lets you drill down into physical design details relevant to OO-code. The class diagram in Figure 9.2 models a class whose instances are screen dialog boxes.

Figure 9.2 Example of a UML class.

The class shape in Figure 9.2 has three compartments. The name compartment includes the class name, as well as a tagged value naming the author of the class. The attribute compartment lists the visibility, name, and type of each attribute. The visibility settings "+", "-","#", and "~" indicate whether the attribute is public, private, protected, or package. These visibility settings are software, not conceptual, issues. The size attribute is initialized to a given area value. The operation compartment specifies what operations are encapsulated in instances of the class. In this example, the operations may be implemented by methods to display the dialog box at a specific position, and to hide the dialog box.

Figure 9.3 shows another class diagram that depicts Employee and Car classes, as well as an association corresponding to the ORM fact type Employee drives Car. The association is depicted by a line between the classes. The role name "driver" on the left end of the association clarifies the intended semantics (an association reading could also be supplied). The open arrow tip at the right end of the association, is a navigability setting indicating that fast access is required from employee instances to their car instances. This may be implemented by including pointers from employee objects (software objects) to the car objects that model the cars that they drive. Navigability settings are implementation issues related to performance, not conceptual issues about the business domain.

By omitting implementation details such as attribute visibility and association navigability, class diagrams can be used for conceptual analysis. When used in this way, class diagrams are somewhat similar to ER models. But there is a significant difference arising from the OO perspective. If you look at the classes in Figure 9.2 and Figure 9.3, what strikes you as missing?

You guessed it! No identification schemas are provided for the classes. In object-oriented programming, objects may be identified by their memory addresses or internal object identifiers (oids), so UML does not require that you provide a value-based identification scheme for use by humans in communicating about the objects. For conceptual analysis, however, such human-oriented reference schemes (e.g., dialog box numbers, employee numbers, car registration numbers) must be supplied. UML does allow you to add such attributes, but has no standard notation for declaring them to be preferred identifiers or even for declaring them to be unique. For this, we choose "{P}" for preferred reference and "{Un}" for uniqueness ($n > 0$), where the n is used to disambiguate cases where the same U constraint might apply to a combination of attributes. Various UML tool vendors choose different notations for such constraints.

Figure 9.3 The navigability setting demands fast access from Employee to Car.

Figure 9.4 Adding nonstandard notations for preferred reference and uniqueness.

In Figure 9.4, for example, employee number and car registration number attributes have been added as the primary identifiers of employee and car, respectively. This entails that they are mandatory and unique. Additionally, the combination of employee name and birthdate has been declared unique. We also dropped the navigation arrow, as it is irrelevant to the business semantics.

The requirement that each class has a value-based identification scheme distinguishes both ORM and ER from UML. ORM classifies objects into entities (non-lexical objects) and values (lexical objects) and requires each entity to be identified by a reference scheme used by humans to communicate about the entity. ORM uses "object", "entity", and "value" to mean "object instance", "entity instance", and "value instance", appending "type" for the relevant set of all possible instances. Entities may be referenced in different ways, and typically change their state over time. Glossing over some subtle points, values are constants (e.g., character strings) that basically denote themselves so do not require a reference scheme to be declared.

Figure 9.5(a) depicts explicitly a simple reference scheme in ORM. If an entity type has more than one candidate reference scheme, one may be declared preferred to assist verbalization of instances (or to reflect actual business practice). A preferred reference scheme for an entity type maps each instance of it onto a unique, identifying value (or a combination of values). In Figure 9.5(a), the reference type has a sample population shown in a reference table (one column for each role). Here icons are used to denote the real world employee entities.

Simple reference schemes may be abbreviated by enclosing the reference mode in parentheses, as in Figure 9.5(b), and an object type's reference table includes values but no icons. References verbalize as existential sentences, e.g., "There is an Employee who has the EmployeeNr 101". Entity instances are referenced elsewhere by definite descriptions, e.g., "The Employee who has the EmpNr 101".

Figure 9.5 A simple reference scheme in ORM, shown (a) explicitly and (b) implicitly.

In a relational database, we might use the preferred reference scheme to provide value-based identity or instead use system-generated row-ids. In an object-oriented implementation, we might use oids (hidden, system-generated object identifiers). Such choices can be added later as annotations to the model. For analysis and validation purposes, however, we need to ensure that humans have a way to identify objects in their normal communication. It is the responsibility of humans (not the system) to enforce constraints on preferred reference types. Assuming humans do enforce the reference type constraints, the system may be used to enforce the elementary fact type constraints.

UML classifies instances into objects and data values. UML *objects* basically correspond to ORM entities, but are assumed to be identified by oids. Although UML does not require entities to have a value-based reference scheme, we should include value-based reference in any UML class intended to capture all the conceptual semantics. UML *data values* basically correspond to ORM values: they are constants (e.g., character strings or numbers) and hence require no oids to establish their identity. Entity types in UML are called *classes*, and value types are basically *data types*. Note that "object" means "object instance", not "object type". A relationship instance in UML is called a *link*, and a relationship type is called an *association*.

9.3 Attributes

Like other ER notations, UML allows relationships to be modeled as attributes. For instance, in Figure 9.6(a) the Employee class has eight attributes. The corresponding ORM diagram is shown in Figure 9.6(b).

Figure 9.6 UML attributes (a) depicted as ORM relationship types (b).

Table 9.2 Multiplicities.

Multiplicity	*Abbreviation*	*Meaning*	*Note*
0..1		0 or 1 (at most one)	
0..*	*	0 to many (zero or more)	
1..1	1	exactly 1	Assumed by default
1..*		1 or more (at least 1)	
n..*		*n* or more (at least *n*)	$n \geq 0$
n..*m*		at least *n* and at most *m*	$m > n \geq 0$

In UML, *attributes are mandatory and single valued by default*. So the employee number, name, title, gender, and smoking status attributes are all mandatory. In the ORM model, the unary predicate "smokes" is optional (not everybody has to smoke). UML does not support unary relationships, so it models this instead as the Boolean attribute "isSmoker", with possible values True or False. In UML the domain (i.e., type) of any attribute may optionally be displayed after it (preceded by a colon). In this example, the domain is displayed only for the isSmoker attribute. By default, ORM tools usually take a closed world approach to unaries, which agrees with the isSmoker attribute being mandatory.

The ORM model also indicates that Gender and Country are identified by codes (rather than names, say). We could convey some of this detail in the UML diagram by appending domain names. For example, "Gendercode" and "Countrycode" could be appended to "gender: " and "birthcountry: " to provide syntactic domains.

In the ORM model it is *optional* whether we record birth country, social security number, or passport number. This is captured in UML by appending [0..1] to the attribute name (each employee has 0 or 1 birth country, and 0 or 1 social security number). This is an example of an *attribute multiplicity constraint*. The main multiplicity cases are shown in Table 9.2. If the multiplicity is not declared explicitly, it is assumed to be 1 (exactly one). If desired, we may indicate the default multiplicity explicitly by appending [1..1] or [1] to the attribute.

In the ORM model, the uniqueness constraints on the right-hand roles (including the EmployeeNr reference scheme shown explicitly earlier) indicate that each employee number, social security number, and passport number refer to at most one employee. As mentioned earlier, UML has no standard graphic notation for such "*attribute uniqueness constraints*", so we've added our own {P} and {U*n*} notations for preferred identifiers and uniqueness. UML 2 added the option of specifying {unique} or {nonunique} as part of a multiplicity declaration, but this is only to declare whether instances of collections for multivalued attributes or multivalued association roles may include duplicates, so it can't be used to specify that instances of single valued attributes or combinations of such attributes are unique for the class.

UML has no graphic notation for an inclusive-or constraint, so the ORM constraint that each employee has a social security number or passport number needs to be expressed textually in an attached *note*, as in Figure 9.6(a). Such *textual constraints* may be expressed informally, or in some formal language interpretable by a tool. In the latter case, the constraint is placed in *braces*.

In our example, we've chosen to code the inclusive-or constraint in SQL syntax. Although UML provides OCL for this purpose, it does not mandate its use, allowing users to pick their own language (even programming code). This of course weakens the portability of the model. Moreover, the readability of the constraint is typically poor compared with the ORM verbalization.

The ORM fact type Employee was born in Country is modeled as a birthcountry attribute in the UML class diagram of Figure 9.6(a). If we later decide to record the population of a country, then we need to introduce Country as a class, and to clarify the connection between birthcountry and Country we would probably reformulate the birthcountry attribute as an association between Employee and Country. This is a significant change to our model. Moreover, any object-based queries or code that referenced the birthcountry attribute would also need to be reformulated. ORM avoids such semantic instability by always using relationships instead of attributes.

Another reason for introducing a Country class is to enable a listing of countries to be stored, identified by their country codes, without requiring all of these countries to participate in a fact. To do this in ORM, we simply declare the Country type to be independent. The object type Country may be populated by a reference table that contains those country codes of interest (e.g., 'AU' denotes Australia).

A typical argument in support of attributes runs like this: "Good UML modelers would declare country as a class in the first place, anticipating the need to later record something about it, or to maintain a reference list; on the other hand, features such as the title and gender of a person clearly are things that will never have other properties, and hence are best modeled as attributes". This argument is flawed. In general, you can't be sure about what kinds of information you might want to record later, or about how important some model feature will become.

Even in the title and gender case, a complete model should include a relationship type to indicate which titles are restricted to which gender (e.g., "Mrs", "Miss", "Ms", and "Lady" apply only to the female sex). In ORM this kind of constraint can be captured graphically as a join-subset constraint or textually as a constraint in a formal ORM language (e.g., **If** Person$_1$ has **a** Title **that** is restricted to Gender$_1$ **then** Person$_1$ is of Gender$_1$). In contrast, attribute usage hinders expression of the relevant restriction association (try expressing and populating this rule in UML).

ORM includes algorithms for dynamically generating ER and UML diagrams as attribute views. These algorithms assign different levels of importance to object types depending on their current roles and constraints, redisplaying minor fact types as attributes of the major object types. Modeling and maintenance are iterative processes. The importance of a feature can change with time as we discover more of the global model, and the domain being modeled itself changes.

To promote semantic stability, ORM makes no commitment to relative importance in its base models, instead supporting this dynamically through views. Elementary facts are the fundamental units of information, are uniformly represented as relationships, and how they are grouped into structures is not a conceptual issue. You can have your cake and eat it too by using ORM for analysis, and if you want to work with UML class diagrams, you can use your ORM models to derive them.

Figure 9.7 (a) Multivalued UML sports attribute depicted as (b) ORM *m:n* fact type.

One way of modeling this in UML is shown in Figure 9.7(a). Here the information about who plays what sport is modeled as the *multivalued attribute* "sports". The "[0..*]" multiplicity constraint on this attribute indicates how many sports may be entered here for each employee. The "0" indicates that it is possible that no sports might be entered for some employee. UML uses a *null value* for this case, just like the relational model. The presence of nulls exposes users to implementation rather than conceptual issues and adds complexity to the semantics of queries. The "*" in "[0..*]" indicates there is *no upper bound* on the number of sports of a single employee. In other words, an employee may play many sports, and we don't care how many. If "*" is used without a lower bound, this is taken as an abbreviation for "0..*".

An equivalent ORM schema is shown in Figure 9.7(b). Here an optional, many:many fact type is used instead of the multivalued sports attribute. As discussed in the next section, this approach may also be used in UML using an *m:n* association.

To discuss *class instance populations*, UML uses *object diagrams*. These are essentially class diagrams in which each object is shown as a separate instance of a class, with data values supplied for its attributes. As a simple example, Figure 9.8(a) includes object diagrams to model three employee instances along with their attribute values. The ORM model in Figure 9.8(b) displays the same sample population, using fact tables to list the fact instances.

Figure 9.8 Populated models in (a) UML and (b) ORM.

For simple cases like this, object diagrams are useful. However, they rapidly become unwieldy if we wish to display multiple instances for more complex cases. In contrast, fact tables scale easily to handle large and complex cases.

ORM constraints are easily clarified using sample populations. For example, in Figure 9.8(b) the absence of employee 101 in the Plays fact table clearly shows that playing sport is optional, and the uniqueness constraints mark out which column or column-combination values can occur on at most one row. In the EmployeeName fact table, the first column values are unique, but the second column includes duplicates. In the Plays table, each column contains duplicates: only the whole rows are unique. Such populations are very useful for checking constraints with the subject matter experts. This validation-via-example feature of ORM holds for all its constraints, not just mandatory roles and uniqueness, since all its constraints are role-based or type-based, and each role corresponds to a fact table column.

As a final example of multivalued attributes, suppose that we wish to record the nicknames and colors of country flags. Let us agree to record at most two nicknames for any given flag and that nicknames apply to only one flag. For example, "Old Glory" and perhaps "The Star-spangled Banner" might be used as nicknames for the United States flag. Flags have at least one color.

Figure 9.9(a) shows one way to model this in UML. The "[0..2]" indicates that each flag has at most two (from zero to two) nicknames. The ["1..*] declares that a flag has one or more colors. An additional constraint is needed to ensure that each nickname refers to at most one flag. A simple attribute uniqueness constraint (e.g., {U1}) is not enough, since the nicknames attribute is set valued. Not only must each nicknames set be unique for each flag, but each element in each set must be unique (the second condition implies the former). This more complex constraint is specified informally in an attached note.

Here the attribute domains are hidden. Nickname elements would typically have a data type domain (e.g., String). If we don't store other information about countries or colors, we might choose String as the domain for country and color as well (although this is subconceptual, because real countries and colors are not character strings). However, since we might want to add information about these later, it's better to use classes for their domains (e.g., Country and Color). If we do this, we need to define the classes as well.

Figure 9.9 A flag model in (a) UML and (b) ORM.

Figure 9.9(b) shows one way to model this in ORM. For verbalization we identify each flag by its country. Since country is an entity type, the reference scheme is shown explicitly (reference modes may abbreviate reference schemes only when the referencing type is a value type). The "≤ 2" frequency constraint indicates that each flag has at most two nicknames, and the uniqueness constraint on the role of NickName indicates that each nickname refers to at most one flag.

UML gives us the choice of modeling a feature as an attribute or an association. For conceptual analysis and querying, explicit associations usually have many advantages over attributes, especially multivalued attributes. This choice helps us verbalize, visualize, and populate the associations. It also enables us to express various constraints involving the "role played by the attribute" in standard notation, rather than resorting to some nonstandard extension. This applies not only to simple uniqueness constraints (as discussed earlier) but also to other kinds of constraints (frequency, subset, exclusion, etc.) over one or more roles that include the role played by the attribute's domain (in the implicit association corresponding to the attribute).

For example, if the association Flag is of Country is depicted explicitly in UML, the constraint that each country has at most one flag can be captured by adding a multiplicity constraint of "0..1" on the left role of this association. Although country and color are naturally conceived as classes, nickname would normally be construed as a data type (e.g., a subtype of String). Although associations in UML may include data types (not just classes), this is somewhat awkward; so in UML, nicknames might best be left as a multivalued attribute. Of course, we could model it cleanly in ORM first.

Another reason for favoring associations over attributes is stability. If we ever want to talk about a relationship, it is possible in both ORM and UML to make an object out of it and simply attach the new details to it. If instead we modeled the feature as an attribute, we would need to first replace the attribute by an association. For example, consider the association Employee plays Sport in Figure 9.8(b). If we need to record a skill level for this play, we can simply objectify this association as Play,and attach the fact type: Play has SkillLevel. A similar move can be made in UML if the play feature has been modeled as an association. In Figure 9.8(a) however, this feature is modeled as the sports attribute, which needs to be replaced by the equivalent association before we can add the new details about skill level. The notion of objectified relationship types or association classes is covered in a later section.

Another problem with multivalued attributes is that queries on them need some way to extract the components, and hence complicate the query process for users. As a trivial example, compare queries Q1, Q2 expressed in ConQuer (an ORM query language) with their counterparts in OQL (the Object Query language proposed by the ODMG). Although this example is trivial, the use of multivalued attributes in more complex structures can make it harder for users to express their requirements.

(Q1) **List each** Color **that** is of Flag 'USA'.
(Q2) **List each** Flag **that** has Color 'red'.

(Q1a) **select** x.colors **from** x **in** Flag **where** x.country = "USA"
(Q2a) **select** x.country **from** x **in** Flag **where** "red" **in** x.colors

For such reasons, multivalued attributes should normally be avoided in analysis models, especially if the attributes are based on classes rather than data types. If we avoid multivalued attributes in our conceptual model, we can still use them in the actual implementation. Some UML and ORM tools allow schemas to be annotated with instructions to override the default actions of whatever mapper is used to transform the schema to an implementation. For example, the ORM schema in Figure 9.9 might be prepared for mapping by annotating the roles played by NickName and Color to map as sets inside the mapped Flag structure. Such annotations are not a conceptual issue, and can be postponed until mapping.

9.4 Associations

UML uses Boolean attributes instead of unary relationships, but allows relationships of all other arities. Optionally, each association may be given at most one name. Association names normally start with a capital letter. *Binary associations are depicted as lines* between classes. Association lines may include elbows to assist with layout or when needed (e.g., for ring relationships). Association roles appear simply as line ends instead of boxes, but may be given role names. Once added, role names may not be suppressed. Verbalization into sentences is possible only for infix binaries, and then only by naming the association with a predicate reading (e.g., "Employs") and using an optional marker "▶" to denote the direction.

Figure 9.10 depicts two binary associations in both UML and ORM. On the UML diagram, the association names, their directional markers, and some role names are displayed. In UML, association names are optional, but role names are mandatory. If a role name is not supplied, the role's name is assumed to be the name of its class (e.g., "employee"). *If two or more roles are played by the same class, the roles must be given different names to distinguish them* (e.g., "buyer", "acquisition"). In the ORM diagram, forward and inverse predicate readings are shown; at most one of these may be omitted. Role names are optional in ORM, and their display (in square brackets) may be toggled on or off.

Figure 9.10 Binary associations in (a) UML and (b) ORM.

Figure 9.11 A ternary association in (a) UML and (b) ORM.

Ternary and higher arity associations in UML are depicted as a diamond connected by lines to the classes, as shown in Figure 9.11(a). Because many lines are used with no reading direction indicator, directional verbalization is ruled out, so the diagram can't be used to communicate in terms of sentences. This non-linear layout also often makes it impractical to conveniently populate associations with multiple instances, unless we use role names for column names. Add to this the impracticality of displaying multiple populations of attributes, and it is clear that class diagrams are of little use for population checks.

As discussed earlier, UML does provide object diagrams for instantiation, but these are convenient only for populating associations with a *single* instance. Adding multiple instances leads to a mess (e.g., Blaha and Premerlani 1998, p. 31). Hence, as noted in the UML Notation Guide, "the use of object diagrams is fairly limited".

The previous section discussed how UML depicts *multiplicity constraints* on attributes. A similar notation is used for associations, where the relevant multiplicities are written next to the relevant roles. Figure 9.12(a) adds the relevant multiplicity constraints to Figure 9.10(a). A "*" abbreviates "0..*", meaning "zero or more", "1" abbreviates "1..1", meaning "exactly one", and "0..1" means "at most one". If no multiplicity is supplied for an association role, "*" is assumed by default (unlike attributes, where 1 is the default multiplicity).

Figure 9.12 UML multiplicity constraints (a) and equivalent ORM constraints (b).

UML places each multiplicity constraint on the "far role", in the direction in which the association is read. Hence the constraints in this example mean that each company employs zero or more employees, each employee is employed by exactly one company, each company acquired zero or more companies, and each company was acquired by at most one company.

The corresponding ORM constraints are depicted in Figure 9.12(b). Recall that multiplicity covers both cardinality (frequency) and optionality. Here the mandatory role constraint indicates that each employee works for at least one company, and the uniqueness constraints indicate that each employee works for at most one company, and each company was acquired by at most one company.

For comparison purposes, Figure 9.13 depicts the *n*:1 association Moon orbits Planet in various notations. The instance diagram in Figure 9.13(a) includes a sample population of moons (p = Phobos, d = Deimos, c = Callisto) and planets (v = Venus, m = Mars, j = Jupiter). For illustration purposes, the ORM diagram in Figure 9.13(e) also includes the sample object and fact populations. The population is significant with respect to multiplicity constraints. Each planet orbits exactly one moon, and the same planet may be orbited by zero or more moons.

The UML (Figure 9.13(b)) and Information Engineering (Figure 9.13(c)) approaches are similar because both express the constraints in terms of multiplicities/cardinalities. In contrast, Barker ER (Figure 9.13(d)) and ORM (Figure 9.13(e)) capture some constraints in terms of mandatory/optional roles and other constraints in terms of cardinality/uniqueness constraints. As shown later, the failure of UML to separate out these two kinds of constraint prevents it from graphically capturing various cases it might otherwise have handled.

For binary associations, there are four possible uniqueness constraint patterns (*n*:1, 1:*n*; 1:1, *m*:*n*) and four possible mandatory role patterns (only the left role mandatory, only the right role mandatory, both roles mandatory, both roles optional). Hence if we restrict ourselves to a maximum frequency of one, there are 16 possible multiplicity combinations for binary associations. The 16 cases are shown in Figure 9.14, in both UML and ORM.

Figure 9.13 A mandatory:optional, *n*:1 association in various notations.

Figure 9.14 Equivalent constraint patterns in UML and ORM.

Figure 9.15 Constraints on a ternary in (a) UML and (b) ORM.

UML allows multiplicity constraints with whole numbers other than zero or one, and also supports multiplicity lists or ranges (e.g., "1..7, 10"). For such cases, ORM uses frequency constraints instead of uniqueness constraints. ORM is more expressive in this regard, since it can apply such constraints to arbitrary collections of roles, not just single roles.

For an elementary *n*-ary association, each internal uniqueness constraint in ORM must span at least *n* - 1 roles. In UML, a multiplicity constraint on a role of an *n*-ary association effectively constrains the population of the other roles combined. For example, Figure 9.15(a) is a UML diagram for a ternary association in which both Room-HourSlot and HourSlot-Activity pairs are unique. For simplicity, reference schemes are omitted.

An ORM depiction of the same association is shown in Figure 9.15(b). The left-hand uniqueness constraint indicates that Room-HourSlot is unique (i.e., for any given room and hour slot, at most one activity is booked). The right-hand uniqueness constraint indicates that HourSlot-Activity is unique (i.e., for any given hour slot and activity, at most one room is booked). An extended version of this example was discussed in Section 1.2, where the ORM diagram better facilitated constraint checking by verbalization and population.

Because it covers some *n*-ary cases like this, UML's multiplicity constraint notation is richer than the optionality/cardinality notation of typical ER. However there are many cases with *n*-ary associations where the multiplicity notation of UML is incapable of capturing even a simple mandatory role constraint, or a minimum frequency constraint above 1. In contrast, the mandatory, uniqueness, and frequency constraint notation of ORM can capture any possible constraint of this nature, on roles or role sequences, on predicates of any arity. So ORM is far richer in this regard.

For example, suppose we modify our room booking example to indicate that all activities have a room-hourslot booking and also have unique names as well as their identifying codes. The modified example, including reference schemes, is shown in Figure 9.16 in both UML and ORM. Because UML bundles both mandatory and uniqueness into a single notion of multiplicity, it cannot capture the constraint that each activity has a booking graphically. The best we can do is add a note, as shown in Figure 9.16(a). This constraint may be expressed graphically in ORM using a mandatory role constraint, as shown in Figure 9.16(b).

Figure 9.16 (a) UML resorts to a note to capture (b) a mandatory constraint in ORM.

This deficiency in UML is a direct consequence of choosing to attach minimum multiplicity to a role other than the immediate role. For the same reason, UML multiplicity constraints are also unable to capture various ORM frequency constraints. In general, *given any n-ary (n > 2) association, if an ORM mandatory or frequency constraint applies to at least 1 and at most n - 2 roles, this cannot be captured by a UML multiplicity constraint*. Some examples of such cases are shown in Figure 9.17. Further discussion on such cases may be found in Halpin (2000c).

Unlike many ER versions, both UML and ORM allow associations to be objectified as first class object types, *called association classes* in UML and *objectified associations* or *nested object types* in ORM. UML requires the same name to be used for the original association and the association class, impeding natural verbalization of at least one of these constructs. In contrast, ORM nesting is based on linguistic *nominalization* (a verb phrase is objectified by a noun phrase), thus allowing both to be verbalized naturally, with different readings for each.

Although UML identifies an association class with its underlying association, it displays them separately, connected by a dashed line (see Figure 9.18(a)). Each person may write many papers, and each paper is written by at least one person. Since authorship is *m*:*n*, the association class Writing has a primary reference scheme based on the combination of person and paper (e.g., the writing by person 'Norma Jones' of paper 33). The optional period attribute stores how long that person took to write that paper.

Figure 9.18(b) shows an ORM schema for this domain. The objectified association Writing is marked independent (by the "!") to indicate that a writing object may exist, independently of whether we record its period. ORM displays Period as an object type, not an attribute, and includes its unit.

Figure 9.17 Some ORM constraints that can't be captured by UML multiplicities.

Figure 9.18 Writing depicted as an objectified association in (a) UML and (b) ORM.

UML allows any association (binary and above) to be objectified into a class, regardless of its multiplicity constraints. In particular UML allows objectification of *n*:1 associations, as shown in Figure 9.19 (a). While this is allowed in ORM 2, it is often a case of poor modeling. For example, given that a moon orbits only one planet, an orbital period may be related directly to the moon without including the planet. So instead of objectifying, we could model the orbital period in UML as an attribute of Moon, or in ORM as an association between Moon and Period. Chapter 10 provides heuristic guidelines for applying objectification in modeling.

Earlier we saw that UML has no graphic notation to capture ORM external uniqueness constraints across roles that are modeled as attributes in UML. Hence we introduced a {U*n*} notation to append textual constraints to the constrained attributes. Simple cases where ORM uses an external uniqueness constraint for *coreferencing* can also be modeled in UML using *qualified associations*. Here, instead of depicting the relevant ORM roles or object types as attributes, UML uses a class, adjacent to a *qualifier*, through which connection is made to the relevant association role. A qualifier in UML is a set of one or more attributes whose values can be used to partition the class, and is depicted as a rectangular box enclosing its attributes. Figure 9.20 is based on an example from the official UML specification, along with the ORM counterpart.

Here each bank account is used by at most one client, and each client may use many accounts. In the UML model, the attribute accountNr is a qualifier on the association, effectively partitioning each bank into different accounts. In the ORM model, an Account object type is explicitly introduced and referenced by combining its bank with its (local) account number.

Figure 9.19 Objectification of an *n*:1 association in (a) UML and (b) ORM.

Figure 9.20 (a) Qualified association in UML, and (b) co-referenced type in ORM.

The UML notation is less clear and less adaptable. For example, if we now want to record something about the account (e.g., its balance) we need to introduce an Account class, and the connection to accountNr is unclear. For a similar example, see Fowler (1997, p. 92), where product is used with Order to qualify an order line association; again, this is unfortunate, since we would normally introduce a Product class to record data about products, and relevant connections are then lost.

As a complicated example of this deficiency, see Blaha and Premerlani (1998, p. 51) where the semantic connection between Node and nodeName is lost. The problem can be solved in UML by using an association class instead, although this is not always natural. The use of qualified associations in UML is hard to motivate, but may be partly explained by their ability to capture some external uniqueness constraints in the standard notation, rather than relying on nonstandard textual notations (such as our {U*n*} notation).

ORM's concept of an external uniqueness constraint that may be applied to a set of roles in one or more predicates provides a simple, uniform way to capture all of UML's qualified associations and unique attribute combinations, as well as other cases not expressible in UML graphical notation (e.g., cases with *m:n* predicates or long join paths). As always, the ORM notation has the further advantage of facilitating validation through verbalization and multiple instantiation.

9.5 Set-Comparison Constraints

Set-comparison constraints declare a subset, equality, or exclusion relationship between the populations of role sequences. This section compares support for these constraints in UML and ORM. A detailed discussion of these constraints in ORM can be found in Section 6.4.

As an extension mechanism, UML allows subset constraints to be specified between *whole associations* by attaching the constraint label "{subset}" next to a dashed arrow between the associations. For example, the subset constraint in Figure 9.21(a) indicates that any person who chairs a committee must be a member of that committee. Figure 9.21(b) shows the same example in ORM.

Figure 9.21 A subset constraint in (a) UML and (b) ORM.

ORM has a mature formalization, including a rigorous theory of schema consistency, equivalence, and implication. Since formal guidelines for working with UML are somewhat immature, care is needed to avoid logical problems. As a simple example, consider the modified version of our committee example shown in Figure 9.22(a), which comes directly from an earlier version of the UML specification, with reference schemes added. Do you spot anything confusing about the constraints?

You probably noticed the problem. The multiplicity constraint of 1 on the chair association indicates that each committee must have at least one chair. The subset constraint tells us that a chair of a committee must also be a member of that committee. Taken together, these constraints imply that each committee must have a member. Hence we would expect to see a multiplicity constraint of "1..*" (one or more) on the Person end of the membership association. However, we see a constraint of "*" (zero or more) instead, which at best is misleading. An equivalent, misleading ORM schema is shown in Figure 9.22(b), where the upper role played by Committee appears optional when in fact it is mandatory.

One might argue that it's okay to leave these schemas unchanged, as the constraint that each committee includes at least one person is implied by other constraints. However, while display options for implied constraints may sometimes be a matter of taste, practical experience has shown that in cases like this it is better to show implied constraints explicitly, as in Figure 9.23, rather than expect modelers or domain experts to figure them out for themselves.

Figure 9.22 (a) A misleading UML diagram and (b) a misleading ORM diagram.

Figure 9.23 All constraints are now shown explicitly in (a) UML and (b) ORM.

Figure 9.24 The relational schema mapped from Figure 9.23.

Some ORM tools can detect the misleading nature of constraint patterns like that of Figure 9.22(b) and ask you to resolve the problem. Human interaction is the best policy here, since there is more than one possible mistake (e.g., is the subset constraint correct leading to Figure 9.23, or is the optional role correct resulting in Figure 9.21?).

If a schema in Figure 9.23 is mapped to a relational database, it generates a referential cycle, since the mandatory fact types for Committee map to different tables (so each committee must appear in both tables). The relational schema is shown in Figure 9.24 (arrows show the foreign key references, one simple and one composite, which correspond to the subset constraints).

Although referential cycles are sometimes unavoidable, they are awkward to implement. In this case, the cycle arose from applying a mandatory role constraint to a nonfunctional role. Unless the business requires it, this should be avoided at the conceptual level (e.g. by leaving the upper role of committee optional, as in Figure 9.21). Relational mapping is covered in detail in Chapter 11.

Since UML does not allow unary relationships, subset constraints between ORM unaries need to be captured textually, using a note to specify an equivalent constraint between Boolean attributes. For example, the ORM subset constraint in Figure 9.25(b), which verbalizes as **Each** Patient **who** smokes is cancer prone, may be captured textually in UML by the note in Figure 9.25(a).

UML 2 introduced a *subsets property* to indicate that the population (extension) of an attribute or association role must be a subset of the population of another compatible attribute or association role respectively. For example, adorning the citizen role in Figure 9.26(a) with {subset resident} means that all citizens are residents (not necessarily of the same country). Figure 9.26 (b) shows the equivalent ORM schema.

Figure 9.25 (a) UML note for (b) ORM subset constraint between unaries.

Figure 9.26 A single role subset constraint in (a) UML and (b) ORM.

However there are still many subset constraint cases in ORM that cannot be represented graphically as a subset constraint in UML. For example, the subset constraint in Figure 9.27(b) that each student with a second given name must have a first given name is captured as a note in Figure 9.27(a) because the relevant ORM fact types are modeled as attributes in UML, and the required subset constraint applies between student sets not name sets. The subset constraint in Figure 9.25(b) is another example.

Moreover, UML does not support subset constraints over arguments that are just parts of relationships, such as the subset constraint in Figure 9.27(b) that students may pass tests in a course only if they enrolled in that course. Figure 9.27(a) models this constraint in UML by transforming the ternary into a binary association class (Enrollment) that has a binary association to Test. Although in this situation an association class provides a good way to cater for a compound subset constraint, sometimes this nesting transformation leads to a very unnatural view of the world. Ideally the modeler should be able to view the world naturally, while having any optimization transformations that reduce the clarity of the conceptual schema performed under the covers.

Figure 9.27 (a) UML model capturing (b) some subset constraints in ORM.

Figure 9.28 Spot anything wrong?

As another constraint example in UML, consider Figure 9.28, which is the UML version of an OMT diagram used in Blaha and Premerlani (1998, p. 68) to illustrate a subset constraint (if a column is a primary key column of a table, it must belong to that table). Can you spot any problems with the constraints?

One obvious problem is that the "1" on the primary key association should be "0..1" (not all columns belong to primary keys), as in Figure 9.29(a). If we allow tables to have no columns (e.g., the schema is to cater for cases where the table is under construction), then the "*" on the define association is fine; otherwise it should be "1..*". Assuming that tables and columns are identified by oids or artificial identifiers, the subset constraint makes sense, but the model is arguably sub-optimal since the primary key association and subset constraint could be replaced by a Boolean isaPKfield attribute on Column.

From an ORM perspective, heuristics lead us to initially model the situation using natural reference schemes as shown in Figure 9.29(b). Here ColumnName denotes the local name of the column in the table. We've simplified reality by assuming that tables may be identified just by their name. The external uniqueness constraints suggest two natural reference schemes for Column: name plus table, or position plus table. We chose the first of these as preferred, but could have introduced an artificial identifier. The unary predicate indicates whether a column is, or is part of, a primary key. If desired, we could derive the association Column is a primary key field of Table from the path: Column is in Table **and** Column is a primary key column (the subset constraint in the UML model is then implied).

What is interesting about this example is the difference in modeling approaches. Most UML modelers seem to assume that oids will be used as identifiers in their initial modeling, whereas ORM modelers like to expose natural reference schemes right from the start and populate their fact types accordingly. These different approaches often lead to different solutions.

Figure 9.29 A corrected UML schema (a) remodeled in (b) ORM.

The main thing is to first come up with a solution that is natural and understandable to the domain expert, because here is where the most critical phase of model validation should take place. Once a correct model has been determined, optimization guidelines can be used to enhance it.

One other feature of the example is worth mentioning. The UML solution in Figure 9.29(a) uses the annotation "{ordered, unique}" to indicate that a table is composed of an *ordered set* (i.e., a sequence with no duplicates) of columns. UML 2 allows the *unique property* to be specified with or without the *ordered property*. By default, ordered = false and unique = true. So either of the settings {ordered} or {ordered, unique} may be used to indicate an ordered set. Either no setting, or the single setting {unique}, indicates a set (the default). If {nonunique} is allowed in this context (this in unclear in the UML specification), one could specify a bag or sequence with the settings {nonunique} or {ordered, nonunique}, respectively

In the ORM community, a debate has been going on for many years regarding the best way to deal with constructors for collection types (e.g., set, ordered set, bag, sequence) at the conceptual level. Our view is that such constructors should not appear in the base conceptual model. Hence the use of Position in Figure 9.29(b) to convey column order (the uniqueness of the order is conveyed by the uniqueness constraint on Column has Position). Keeping fact types elementary has so many advantages (e.g., validation, constraint expression, flexibility, and simplicity) that it seems best to relegate constructors to derived views. We have more to say about collection types in Chapter 10.

In ORM, an *equality constraint* between two compatible role sequences is shorthand for two subset constraints (one in either direction) and is shown as circled "=". Such a constraint indicates that the populations of the role-sequences must always be equal. If two roles played by an object type are mandatory, then an equality constraint between them is implied (and hence not shown). UML has no graphic notation for equality constraints. For whole associations we could use two separate subset constraints, but this would be very messy. In general, equality constraints in UML may be specified as textual constraints in notes.

As a simple example, the equality constraint in Figure 9.30(b) indicates that if a patient's systolic blood pressure is measured, so is his/her diastolic blood pressure (and vice versa). In other words, either both measurements are taken, or neither.

Figure 9.30 A simple equality constraint in (a) UML and (b) ORM.

Figure 9.31 Exclusive-or: each account is used by a person or corporation but not both.

This kind of constraint is fairly common. Less common are equality constraints between sequences of two or more roles. Figure 9.30(a) models this in UML as a textual constraint between two attributes for blood pressure readings.

Subset and equality constraints enable various classes of schema transformations to be stated in their most general form, and ORM's more general support for these constraints allows more transformations to be easily visualized (e.g., see the equivalence theorem PSG2 in Section 14.2).

Although UML does not include a graphic notation for a pure exclusion constraint, it does include an *exclusive-or constraint* to indicate that each instance of a class plays *exactly one* association role from a specified set of alternatives. To indicate the constraint, "{xor}" is placed beside a dashed line connecting the relevant associations. Figure 9.31(a), which is based on an example from the UML specification, indicates that each account is used by a person or corporation but not both. For simplicity, reference schemes and other constraints are omitted.

Prior to version 1.3 of UML, "{or}" was used for this constraint, which was misleading since "or" is typically interpreted in the inclusive sense. The equivalent ORM model is shown in Figure 9.31(b), where the exclusive-or constraint is simply an orthogonal combination of a disjunctive mandatory role (inclusive-or) constraint (circled dot) and an exclusion constraint (circled "X").

Although the current UML specification describes the xor constraint as applying to a set of associations, we need to apply the constraint to a set of roles (association-ends) to avoid ambiguity in cases with multiple common classes. Visually this could be shown by attaching the dashed line near the relevant ends of the associations, as shown in Figure 9.32(a). Unfortunately, UML attaches no significance to such positioning, so the xor constraint could be misinterpreted to mean that each company must lease or purchase some vehicle rather than the intended constraint that each vehicle is either leased or purchased, a constraint captured unambiguously by the ORM schema in Figure 9.32(b).

Figure 9.32 The exclusive-or constraint should apply between association roles.

Figure 9.33 (a) Nobody wrote and reviewed a book; (b) nobody wrote and reviewed *the same* book; (c) UML version of (b).

UML has no symbols for exclusion or inclusive-or constraints. If UML symbols for these constraints are ever considered, then "{x}" and "{or}" respectively seem appropriate—this choice also exposes the composite nature of "{xor}".

UML xor-constraints are intended to apply between single roles. The current UML specification seems to imply that these roles must belong to different associations. If so, UML cannot use an xor-constraint between roles of a ring fact type (e.g., between the husband and the wife roles of a marriage association). ORM exclusion constraints cover this case, as well as many other cases not expressible in UML graphic notation. As a trivial example, consider the difference between the following two constraints: no person both wrote a book and reviewed a book, and no person wrote and reviewed the same book. ORM clearly distinguishes these by noting the precise arguments of the constraint (compare Figure 9.33(a) with Figure 9.33 (b)).

The pair-exclusion constraint in Figure 9.33(b) can be expressed in UML by a note connected by dotted lines to the two associations, as shown in Figure 9.33(c). Alternatively, one could attach a textual constraint to either the Person class (e.g., "bookAuthored and bookReviewed are disjoint sets") or the Book class (e.g. "author and reviewer are disjoint sets"), but the choice of class would be arbitrary.

UML has no graphic notation for exclusion between attributes, or between attributes and association roles. An exclusion constraint in such cases may often be captured as a textual constraint. For example, in Figure 9.34(a), the exclusion constraint that each employee is either tenured or is contracted until some date may be captured by the textual constraint shown.

Figure 9.34 An exclusion constraint modeled in (a) UML and (b) ORM.

Here the constraint is specified in OCL. The expressions "-> isEmpty()" and "-> notEmpty()" are equivalent to "is null" and "is not null" in SQL. Figure 9.34(b) depicts the exclusion constraint graphically in ORM. There are other ways to model this case in UML (e.g., using subtypes) that offer more chance to capture the constraints graphically.

9.6 Subtyping

Both UML and ORM support *subtyping*, using substitutability ("is-a") semantics, where each instance of a subtype is also an instance of its supertype(s). For example, declaring Woman to be a subtype of Person entails that each woman is a person, and hence Woman inherits all the properties of Person. Given two object types, *A* and *B*, we say that *A* is a *subtype* of *B* if, for each database state, the population of *A* is included in the population of *B*. For data modeling, the only subtypes of interest are *proper* subtypes. We say that *A* is a proper subtype of *B* if and only if *A* is a subtype of *B*, and there is a possible state where the population of *B* includes an instance not in *A*. From now on, we'll use "subtype" as shorthand for "proper subtype".

In both UML and ORM, *specialization* is the process of introducing subtypes, and *generalization* is the inverse procedure of introducing a supertype. Both UML and ORM allow single *inheritance*, as well as multiple inheritance (where a subtype has more than one direct supertype). For example, AsianWoman may be a subtype of both AsianPerson and Woman. In UML, "subclass" and "superclass" are synonyms of "subtype" and "supertype", respectively, and generalization may also be applied to things other than classes (e.g., interfaces, use case actors, and packages). This section restricts its attention to subtyping between object types (classes).

In ORM, a subtype inherits all the roles of its supertypes. In UML, a subclass inherits all the attributes, associations, and operations/methods of its supertype(s). An operation implements a service and has a signature (name and formal parameters) and visibility, but may be realized in different ways. A method is an implementation of an operation, and hence includes both a signature and a body detailing an executable algorithm to perform the operation. In an inheritance graph, there may be many methods for the same operation (*polymorphism*), and scoping rules are used to determine which method is actually used for a given class. If a subclass has a method with the same signature as a method of one of its supertypes, this is used instead for that subclass (*overriding*). For example, if Rectangle and Triangle are subclasses of Shape, all three classes may have different methods for display(). This section focuses on data modeling, not behavior modeling, covering inheritance of static properties (attributes and associations), but ignoring inheritance of operations or methods.

Subtypes are used in data modeling to assert typing constraints, encourage reuse of model components, and show a classification scheme (taxonomy). In this context, typing constraints ensure that subtype-specific roles are played only by the relevant subtype.

Figure 9.35 Subtyping displayed by directed acyclic graphs in (a) UML and (b) ORM.

Since a subtype inherits the properties of its supertype(s), only its specific roles need to be declared when it is introduced. Apart from reducing code duplication, the more generic supertypes are likely to find reuse in other applications. At the coding level, inheritance of operations/methods augments the reuse gained by inheritance of attributes and association roles. Using subtypes to show taxonomy is of limited use, since taxonomy is often more efficiently captured by predicates. For example, the fact type Person is of Gender {male, female} implicitly provides the taxonomy for the subtypes MalePerson and FemalePerson.

Both UML and ORM display subtyping using *directed acyclic graphs*. A directed graph is a graph of nodes with directed connections, and "acyclic" means that there are no cycles (a consequence of proper subtyping). Figure 9.35 shows a subtype pattern in UML and ORM. An arrow from one node to another shows that the first is a subtype of the second. UML uses a thin arrow shaft with an open arrowhead, while ORM uses a solid shaft and arrowhead. As an alternative notation, UML also allows separate shafts to merge into one, with one arrowhead acting for all (e.g., *E* and *F* are subtypes of *C*). Since subtypehood is transitive, indirect connections are omitted (e.g., since *E* is a subtype of *C*, and *C* is a subtype of *A*, it follows that *E* is a subtype of *A*, so there is no need to display this implied connection).

UML includes four predefined constraints to indicate whether subtypes are exclusive or exhaustive. If subtype connections are shown with separate arrowheads, the constraints are placed in braces next to a dotted line connecting the subtype links, as in Figure 9.35(a) (top). We assume that this line may include elbows, as shown for the disjoint constraint, to enable such cases to be specified. If the subtype connections are shared, the constraints are placed near the shared arrowhead, as in Figure 9.35(a) (bottom). The {overlapping} and {disjoint} options, respectively, indicate that the subtypes overlap or are mutually exclusive. Originally {complete} simply meant that all subtypes were shown, but this was redefined to mean exhaustive (i.e., the supertype equals the union of its subtypes). The {incomplete} option means that the supertype is more than the union of its subtypes. The default is {disjoint, incomplete}. Users may add other keywords.

By default, ORM subtypes may overlap, and subtypes need not collectively exhaust their supertype. ORM allows graphic constraints to indicate that subtypes are mutually exclusive (a circled "X" connected to the relevant subtype links), collectively exhaustive (a circled dot), or both (a circled, crossed dot), as shown in Figure 9.35(b). ORM's approach is that exclusion and totality constraints are enforced on populations, not types. An overlapping "constraint" does not mean that the populations must overlap, just that they may overlap. Hence from an ORM viewpoint, this is not really a constraint at all, so there is no need to depict it. In ORM, subtype exclusion and totality constraints are often implied by other constraints in conjunction with formal subtype definitions.

For any subtype graph, the top supertype is called the *root*, and the bottom subtypes (those with no descendants) are called *leaves*. In UML this can be made explicit by adding "{root}" or "{leaf}" below the relevant class name. If we know the whole subtype graph is shown, there is little point in doing this, but if we were to display only part of a subtype graph, this notation makes it clear whether or not the local top and bottom nodes are also like that in the global schema. For example, from Figure 9.36 we know that globally Party has no supertype and that MalePerson and FemalePerson have no subtypes. Since Party is not marked as a leaf node, it may have other subtypes not shown here.

UML also allows an ellipsis "…" in place of a subclass to indicate that at least one subclass of the parent exists in the global schema, but its display is suppressed on the diagram. Currently ORM does not include a root/leaf notation or an ellipsis notation for subtypes. Such notations could be a useful extension to ORM diagrams.

UML distinguishes between *abstract* and *concrete* classes. An abstract class cannot have any direct instances and is shown by writing its name in italics or by adding "{abstract}" below the class name. Abstract classes are realized only through their descendants. Concrete classes may be directly instantiated. This distinction seems to have little relevance at the conceptual level and is not depicted explicitly in ORM. For code design, however, the distinction is important (e.g., abstract classes provide one way of declaring interfaces, and in C++ abstract operations correspond to pure virtual operations, while leaf operations map to nonvirtual operations). For further discussion of this topic, see Fowler (1997, pp. 85–88) and Booch et al. (1999, pp. 125–126).

Figure 9.36 Party may have other subtypes not shown here.

Like other ER notations, UML provides only weak support for defining subtypes. A *discriminator* label may be placed near a subtype arrow to indicate the basis for the classification. For example, Figure 9.37 includes a "gender" discriminator to specialize Person into MalePerson and FemalePerson.

The UML specification says that the discriminator names "a partition of the subtypes of the superclass". In formal work, the term "partition" usually implies the division is both exclusive and exhaustive. In UML, the use of a discriminator does not imply that the subtypes are exhaustive or complete, but at least some authors argue that they must be exclusive (Fowler 1997, p. 78). If that is the case, there does not appear to be any way in UML of declaring a discriminator for a set of overlapping subtypes.

The same discriminator name may be repeated for multiple subclass arrows to show that each subclass belongs to the same classification scheme. This repetition can be avoided by merging the arrow shafts to end in a single arrowhead, as in Figure 9.37.

In Figure 9.37, the gender attribute of Patient is used as a discriminator. This attribute is based on the enumerated type Gendercode, which is defined using the stereotype «enumeration», and listing its values as attributes. The notes at the bottom are needed to ensure that instances populating these subtypes have the correct gender. For example, without these notes there is nothing to stop us populating MalePatient with patients that have the value 'f' for their gendercode.

As discussed in Section 6.5, ORM overcomes this problem by requiring that if a taxonomy is captured both by subtyping and a classifying fact type, these two representations must be synchronized, either by deriving the subtypes from *formal subtype definitions*, or by deriving the classification fact type from asserted subtypes. For example, the populated ORM schema in Figure 9.38 adopts the first approach. The ORM partition (exclusion and totality) constraint is now implied by the combination of the subtype definitions and the three constraints on the fact type Patient is of Gender.

Figure 9.37 Gender is used as a discriminator to partition Patient.

Figure 9.38 With formal subtype definitions, subtype constraints are implied.

While the subtype definitions in Figure 9.38 are trivial, in practice more complicated subtype definitions are sometimes required. As a basic example, consider a schema with the fact types City is in Country, City has Population, and now define LargeUScity as follows: **Each** LargeUScity **is a** City **that** is in Country 'US' **and** has Population > 1000000. There does not seem to be any convenient way of doing this in UML, at least not with discriminators. We could perhaps add a derived Boolean isLarge attribute, with an associated derivation rule in OCL, and then add a final subtype definition in OCL, but this would be less readable than the ORM definition just given. For a more detailed discussion of subtyping in ORM, including the notion of context-dependent reference, see Sections 6.5 and 6.6. Mapping of subtypes is discussed in Chapter 11.

9.7 Other Constraints and Derivation Rules

A *value constraint* restricts the population of a value type to a finite set of values specified either in full (*enumeration*), by start and end values (*range*), or some combination of both (*mixture*). The values themselves are primitive data values, typically character strings or numbers.

In UML, enumeration types may be modeled as classes, stereotyped as enumerations, with their values listed (somewhat unintuitively) as attributes. Ranges and mixtures may be specified by declaring a textual constraint in braces, using any formal or informal language. For example, see Figure 9.39(a).

Figure 9.39 Data value restrictions declared as enumerations or textual constraints.

Figure 9.40 Ring constraints expressed in (a) UML, (b) UML, and (c) ORM.

Figure 9.39(b) depicts the same value constraints in ORM. Value constraints other than enumeration, range, and mixture may be declared in UML or ORM as textual constraints, for example, {committeeSize must be an odd number}. For further UML examples, see Rumbaugh et al. (1999, pp. 236, 268).

A ring fact type has at least two roles played by the same object type (either directly, or indirectly via a supertype). A *ring constraint* applies a logical restriction on the role pair. For example, the association Person is a parent of Person might be declared acyclic and intransitive.

UML does not provide ring constraints built in, so the modeler needs to specify these as a textual constraint in some chosen language or as a note. In UML, if a textual constraint applies to just one model element (e.g., an association), it may be added in braces next to that element, as in Figure 9.40(a). Here the {acyclic, intransitive} notation is nonstandard but is assumed to be user supported.

It is the responsibility of the modeling tool to ensure that the constraint is linked internally to the relevant model element and to interpret any textual constraint expressions. If the tool cannot interpret the constraint, it should be placed inside a note (dog-eared rectangle), without braces, showing that it is merely a comment, and explicitly linked to the relevant model element(s), as shown in Figure 9.40(b). Figure 9.40(c) displays the ring constraints graphically in ORM (see Section 7.3 for a detailed discussion of ring constraints in ORM).

A *join constraint* applies to one or more role sequences, at least one of which is projected from a path from one predicate through an object type to another predicate. The act of passing from one role through an object type to another role invokes a conceptual join, since the same object instance is asserted to play both the roles. Although join constraints arise frequently in real applications, UML has no graphic symbol for them. To declare them on a UML diagram, write a constraint or comment in a note attached to the model elements involved.

For example, Figure 9.41 links a comment to three associations. This example is based on a room scheduling application at a university with built-in facilities in various lecture and tutorial rooms. Example facility codes are PA = Personal Address system, DP = Data Projection facility, INT = Internet access.

As discussed in Section 10.1, ORM provides deep support for join constraints. Role sequences featuring as arguments in set comparison constraints may arise from projections over a join path.

Figure 9.41 Join constraint specified as a comment in UML.

For example, in Figure 9.42, the subset constraint runs from the Room-Facility role-pair projected from the path: Room at **an** HourSlot is booked for **an** Activity **that** requires **a** Facility. This path includes a conceptual join on Activity. The constraint may be formally verbalized as: **If a** Room at **an** HourSlot is booked for **an** Activity **that** requires **a** Facility **then that** Room provides **that** Facility. Figure 9.42 includes a satisfying population for the three fact types. This again illustrates how ORM facilitates validation constraints via sample populations. The UML associations in Figure 9.41 are not so easily populated on the diagram. Other join constraint examples are discussed in Section 10.1.

Figure 9.42 A join-subset constraint in ORM.

In UML, the term "*aggregation*" is used to describe a *whole/part relationship*. For example, a team of people is an aggregate of its members, so this membership may be modeled as an aggregation association between Team and Person. Several different forms of aggregation might be distinguished in real world cases. For example, Odell and Bock (Odell 1998, pp. 137–165) discuss six varieties of aggregation (component-integral, material-object, portion-object, place-area, member-bunch, and member-partnership), and Henderson-Sellers (Barbier et al., 2003) also distinguishes several kinds of aggregation.

UML 2 associations are classified into one of three kinds: ordinary association (no aggregation), shared (or simple) aggregation, or composite (or strong) aggregation. Hence UML recognizes only two varieties of aggregation: shared and composite. Some versions of ER include an aggregation symbol (typically only one kind). ORM and popular ER approaches currently include no special symbols for aggregation.

These different stances with respect to aggregation are somewhat reminiscent of the different modeling positions with respect to null values. Although over 20 kinds of null have been distinguished in the literature, the relational model recognizes only 1 kind of null. Codd's version 2 of the relational model includes 2 kinds of null, and ORM argues that nulls have no place in base conceptual models (because all its asserted facts are atomic). But let's return to the topic at hand.

Shared aggregation is denoted in UML as a binary association, with a *hollow diamond* at the "whole" or "aggregate" end of the association. *Composition* (*composite aggregation*) is depicted with a *filled diamond*. For example, Figure 9.43(a) depicts a composition association from Club to Team and a shared aggregation association from Team to Person.

In ORM, which currently has no special notation for aggregation, this situation would be modeled as shown in Figure 9.43(b). Does Figure 9.43(a) convey any extra semantics, not captured in Figure 9.43(b)? At the conceptual level, it is doubtful whether there is any additional useful semantics. At the implementation level, however, there is additional semantics. Let's discuss this in more detail.

The UML specification declares that "both kinds of aggregation define a transitive … relationship". The use of "transitive" here is somewhat misleading, since it refers to indirect aggregation associations rather than base aggregation associations. For example, if Club is an aggregate of Team, and Team is an aggregate of Person, it follows that Club is an aggregate of Person.

Figure 9.43 Composition (composite aggregation) and shared aggregation in UML.

Figure 9.44 A derived aggregation in (a) UML and (b) ORM.

However, if we wanted to discuss this result, it should be exposed as a *derived association*. In UML, derived associations are indicated by prefixing their names with a *slash* "/". The *derivation rule* can be expressed as a constraint, either connected to the association by a dependency arrow or simply placed beside the association as in Figure 9.44(a).

In ORM, derived fact types are marked with a trailing asterisk, with their derivation rules specified in an ORM textual language (see Figure 9.44(b). In many cases, derivation rules may also be diagrammed as a join-subset or join-equality constraint. As this example illustrates, the derived transitivity of aggregations can be captured in ORM without needing a special notation for aggregation.

The UML specification declares that "both kinds of aggregation define a transitive, antisymmetric relationship (i.e., the instances form a directed, noncyclic graph)". Recall that a relation R is antisymmetric if and only if, for all x and y, if x is not equal to y then xRy implies that not yRx. It would have been better to simply state that paths of aggregations must be acyclic.

At any rate, this rule is designed to stop errors such as that shown in Figure 9.45. If a person is part of a team, and a team is part of a club, it doesn't make sense to say that a club is part of a person. Since ORM does not specify whether an association is an aggregation, illegal diagrams like this can't occur in ORM.

Figure 9.45 Illegal UML model. Aggregations should not form a cycle.

Of course, it is possible for an ORM modeler to make a silly mistake by adding an association such as Club is part of Person, where "is part of" is informally understood in the aggregation sense, and this would not be formally detectable. But avoidance of such a bizarre occurrence doesn't seem to be a compelling reason to add aggregation to ORM's formal notation. There are plenty of associations between Club and Person that do make sense, and plenty that don't. In some cases, however, it is important to assert constraints such as acyclicity, and this is handled in ORM by ring constraints. That said, there have been some recent proposals to add formal semantics for various forms of the part-of relationship to ORM. For example, Keet (2006) proposes adding several different mereological part-of predicates as well as four kinds of meronymic relations.

Composition does add some important semantics to shared aggregation. To begin with, it requires that each part belongs to at most one whole at a time. In ORM, this is captured by adding a uniqueness constraint to the role played by the part (e.g., see the role played by Team in Figure 9.43(b)). In UML, the multiplicity at the whole end of the association must be 1 or 0..1. If the multiplicity is 1, as in Figure 9.43(a), the role played by the part is both unique and mandatory, as in Figure 9.43(b).

As an example where the multiplicity is 0..1 (i.e., where a part optionally belongs to a whole), consider the ring fact type of Figure 9.46: Package contains Package. Here "contains" is used in the sense of "directly contains". The UML specification notes that "composition instances form a strict tree (or rather a forest)". This strengthening from directed acyclic graph to tree is an immediate consequence of the functional nature of the association (each part belongs to at most one whole), and hence ORM requires no additional notation for this. In this example, the ORM schema explicitly includes an acyclic constraint. This direct containment association is intransitive by implication (acyclicity implies irreflexivity, and any functional, irreflexive association is intransitive).

UML allows some alternative notations for aggregation. If a class is an aggregate of more than one class, the association lines may be shown joined to a single diamond, as in Figure 9.47(a). For composition, the part classes may be shown nested inside the whole by using role names, and multiplicities of components may be shown in the top right-hand corners, as in Figure 9.47(b).

Some authors list kinds of association that are easily confused with aggregation but should not be modeled as such (e.g., topological inclusion, classification inclusion, attribution, attachment, and ownership (Martin and Odell 1998; Odell 1998)).

Figure 9.46 Direct containment modeled in (a) UML and (b) ORM.

Figure 9.47 Alternative UML notations for aggregation.

For example, Finger belongs to Hand is an aggregation, but Ring belongs to Finger is not. There is some disagreement among authors about what should be included on this list. For example, some treat attribution as a special case of aggregation, namely, a composition between a class and the classes of its attributes (Rumbaugh et al. 1999).

For conceptual modeling purposes, agonizing over such distinctions might not be worth the trouble. Obviously there are different stances that you could take about how, if at all, aggregation should be included in the conceptual modeling phase. You can decide what's best for you. The chapter notes provide further discussion on this issue.

Let's now look at the notion of *initial values*. The basic syntax of an attribute specification in UML includes six components as shown. Square and curly brackets are used literally here as delimiters (not as BNF symbols to indicate optional components).

visibililty name [*multiplicity*] : *type-expression* = *initial-value* {*property string*}

If an attribute is displayed at all, its name is the only thing that must be shown. The visibility marker (+, #, –, ~ denote public, protected, private, package respectively) is an implementation concern and will be ignored in our discussion. Multiplicity has been discussed earlier and is specified for attributes in square brackets, e.g., [1..*].

For attributes, the default multiplicity is 1, that is, [1..1]. The type expression indicates the domain on which the attribute is based (e.g., String, Date). Initial-value and property string declarations may optionally be declared. Property strings may be used to specify aspects such as changeability (see later).

An attribute may be assigned an initial value by including the value in the attribute declaration after "=" (e.g., diskSize = 9; country = USA; priority = normal). The language in which the value is written is an implementation concern.

In Figure 9.48(a), the nrColors attribute is based on a simple domain (e.g., PositiveInteger) and has been given an initial value of 1. The resolution attribute is based on a composite domain (e.g., PixelArea) and has been assigned an initial value of (640,480).

Unless overridden by another initialization procedure (e.g., a constructor), declared initial values are assigned when an object of that class is created. This is similar to the database notion of *default value*s, where during the insertion of a tuple an attribute may be assigned a predeclared default value if a value is not supplied by the user.

Figure 9.48 Attributes assigned initial values in (a) UML and (b) ORM extension.

However, UML uses the term "default value" in other contexts only (e.g., template and operation parameters), and some authors claim that default values are not part of UML models (Rumbaugh et al. 1999, p. 249).

The SQL standard treats **null** as a special instance of a default value, and this is supported in UML, since the specification notes that "a multiplicity of 0..1 provides for the possibility of null values: the absence of a value". So an optional attribute in UML can be used to model a feature that will appear as a column with the default value of null, when mapped to a relational database. Presumably a multiplicity of [0..*] or [0..n] for any $n > 1$ also allows nulls for multivalued attributes, even though an empty collection could be used instead.

Currently, ORM has no explicit support for initial/default values. However, UML initial values and relational default values could be supported by allowing default values to be specified for ORM roles. At the meta-level, we add the fact type: Role has default- Value. At the external level, instances of this could be specified on a predicate properties sheet, or entered on the diagram (e.g., by attaching an annotation such as d: *value* to the role, and preferably allowing this display to be toggled on/off). For example, the role played by NrColors in Figure 9.48(b) is allocated a default value of 1. When mapped to SQL, this should add the declaration "default 1" to the column definition for ClipArt.nrColors.

To support the composite initial values allowed in UML, composite default values could be specified for ORM roles played by compositely identified object types (coreferenced or nested). When coreferencing involves at least two roles played by the same or compatible object types, an order is needed to disambiguate the meaning of the composite value. For example, in Figure 9.48(b) the role played by Resolution is assigned a default composite value of (640,480). To ensure that the 640 applies to the horizontal pixelcount and the 480 applies to the vertical pixelcount (rather than the other way round), this ordering needs to be applied to the defining roles of the external uniqueness constraint. ORM tools often determine this ordering from the order in which the roles are selected when entering this constraint.

If all or most roles played by an object type have the same default, it may be useful to allow a default value to be specified for the object type itself. This could be supported in ORM by adding the meta fact type ObjectType has default- Value and providing some notation for instantiating it (e.g., by an entry in an Object Type Properties sheet

or by annotating the object type shape with d: *value*). This corresponds to the default clause permitted in a create-domain statement in the SQL standard. Note that an object-type default can always be expressed instead by role-based defaults, but not conversely (since the default may vary with the role).

Specification of default values does not cover all the cases that can arise with regard to default information in general. A proposal for providing greater support for default information in ORM is discussed in Halpin and Vermeir (1997), but this goes beyond the built-in support for defaults in either UML or SQL. Default information can be modeled informally by using a predicate to convey this intention to a human. For example, we might specify default medium (e.g., 'CD', 'DVD') preferences for delivery of soft products (e.g., music, video, software) using the 1:n fact type: Medium is default preference for SoftProduct.

In cases like this where default values overlap with actual values, we may also wish to classify instances of relevant fact types as actual or default (e.g., Shipment used Medium). For the typical case where the uniqueness constraint on the fact type spans *n* - 1 roles, this can be achieved by including fact types to indicate the default status (e.g., Shipment was based on Choice {actual, default}), resulting in extra columns in the database to record the status. While this approach is generic, it requires the modeler and user to take full responsibility for distinguishing between actual and default values.

In UML, restrictions may be placed on the *changeability* of attributes, as well as the roles (ends) of binary associations. It is unclear whether changeability may be applied to the ends of *n*-ary associations. UML 2 recognizes the following four values for changeability, only one of which can apply at a given time:

- unrestricted
- readOnly
- addOnly
- removeOnly

The default changeability is "unrestricted" (any change is permitted). The value "unrestricted" was formerly called "changeable", which itself was formerly called "none". The other settings may be explicitly declared in braces. For an attribute, the braces are placed at the end of the attribute declaration. For an association, the braces are placed at the opposite end of the association from the object instance to which the constraint applies.

Recall that in UML a "link" is an instance of an association. The term "*readOnly*" (formerly called "frozen") means that once an attribute value or link has been inserted, it cannot be updated or deleted, and no additional values/links may be added to the attribute/association (for the constrained object instance).

The term "*addOnly*" means that although the original value/link cannot be deleted or updated, others values/links may be added to the attribute/association (for the constrained object instance). Clearly, addOnly is only meaningful if the maximum multiplicity of the attribute/association-role exceeds its minimum multiplicity. The term "*removeOnly*" means that the only change permitted for an existing attribute value or link is to delete it.

Figure 9.49 Changeability of attributes and association roles.

As a simple if unrealistic example, see Figure 9.49. Here employee number, birthdate, and country of birth are readOnly for Employee, so they cannot be changed from their original value. For instance, if we assign an employee the employee number 007, and enter his/her birthdate as 02/15/1946 and birth country as 'Australia', then we can never make any changes or additions to that.

Notice also that for a given employee, the set of languages and the set of countries visited are addOnly. Suppose that when facts about employee 007 are initially entered, we set his/her languages to {Latin, Japanese} and countries visited to {Japan}. As long as employee 007 is referenced in the database, these facts may never be deleted. However we may add to these (e.g., later we might add the facts that employee 007 speaks German and visited India).

By default, the other properties are changeable. For example, employee 007 might legally change his name from 'Terry Hagar' to 'Hari Seldon', and the countries he wants to visit might change over time from {Ireland, USA} to {Greece, Ireland}.

Some traditional data modeling approaches also note some restrictions on changeability. As discussed in the previous chapter, the Barker ER notation includes a diamond to mark a relationship as nontransferable (once an instance of an entity type plays a role with an object, it cannot ever play this role with another object). Although changeability restrictions can be useful, in practice their application in database settings is limited.

One reason for this is that we almost always want to allow facts entered into a database to be changed. With snapshot data, this is the norm, but even with historical data changes can occur. The most common occurrence of this is to allow for corrections of mistakes, which might be because we were told the wrong information originally or because we carelessly made a misspelling or typo when entering the data.

In exceptional cases, we might require that mistakes of a certain kind be retained in the database (e.g., for auditing purposes) but be corrected by entering later facts to compensate for the error. This kind of approach makes sense for bank transactions (see Figure 9.50). For example, if a deposit transaction for $100 was mistakenly entered as $1000, the record of this error is kept, but once the error is detected it can be compensated for by a bank withdrawal of $900. As a minor point, the balance is both derived and stored, and its readOnly status is typically implied by the readOnly settings on the base attributes, together with a rule for deriving balance.

Figure 9.50 All attributes of Transaction are read only.

Some authors allow changeability to be specified for a class, as an abbreviation for declaring this for all its attributes and opposite association ends (Booch et al. 1999, p. 184). For instance, all the {readOnly} constraints in Figure 9.50 might be replaced by a single {readOnly} constraint below the name "Transaction". While this notation is neater, it would be rarely used. Even in this example, we would probably want to allow for the possibility of adding nonfrozen information later (e.g., a transaction might be audited by zero or more auditors).

Changeability settings are useful in the design of program code. Although changeability settings are not currently supported in ORM, which focuses on static constraints, they are being considered in extensions to support dynamic constraints (see next chapter). In the wider picture, being able to completely model security issues (e.g., who has the authority to change what) would provide extra value.

As discussed earlier, UML allows {ordered} and {unique} properties to be specified for multivalued attributes and association ends. Since {unique} is true by default, the use of {ordered} alone indicates an ordered set (a sequence with no duplicates). For example, Figure 9.51(a) shows one way of modeling authorship of papers in UML. Each paper has a list or sequence of authors, each of whom may appear at most once on the list.

This may be modeled in flat ORM by introducing a Position object type to store the sequential position of any author on the list, as shown in Figure 9.51(b). The uniqueness constraint on the first two roles declares that for each paper an author occupies at most one position; the constraint covering the first and third roles indicates that for any paper, each position is occupied by at most one author. The textual constraint indicates that the list positions are numbered sequentially from 1.

Figure 9.51 An ordered set modeled in (a) UML and (b) ORM.

Although this ternary representation may appear awkward, it is easy to populate and it facilitates any discussion involving position (e.g., who is the second author for paper 21?). From an implementation perspective, an ordered set structure could still be chosen.

An ordered set is an example of a collection type. As discussed in Chapter 10, some versions of ORM allow collections to be specified as mapping annotations in a similar way to UML, and some ORM versions allow collections to be modeled directly as first class objects.

UML 2 introduced the notion of *association redefinition*. This concept is complex and applies to generalizations as well as associations. One main use of it is to specify stronger constraints on an association role that specializes a role played by a supertype. For example, in Figure 9.52(a) the executiveCar role redefines the assignedCar role, applying a stronger multiplicity constraint on it that applies only to executives. Effectively, the association Executive is assigned CompanyCar is treated as a specialization of the Employee is assigned CompanyCar association. Although some versions of ORM support a similar notion, most ORM versions require the stronger multiplicity to be asserted in a textual constraint, as shown in Figure 9.52 (b).

Now let's consider *derived data*. In UML, derived elements (e.g., attributes, associations, or association-roles) are indicated by prefixing their names with "/". Optionally, a *derivation rule* may be specified as well. The derivation rule can be expressed as a constraint or note, connected to the derived element by a dashed line. This line is actually shorthand for a dependency arrow, optionally annotated with the stereotype name «derive». Since a constraint or note is involved, the arrow-tip may be omitted (the constraint or note is assumed to be the source). For example, Figure 9.53(a) includes area as a derived attribute. Figure 9.53(b) shows the ORM schema.

The UML dependency line may also be omitted entirely, with the constraint shown in braces next to the derived element (in this case, it is the modeling tool's responsibility to maintain the graphical linkage implicitly). A club-membership example of this was included earlier.

As another example, Figure 9.54(a) expresses uncle information as a derived association. For illustration purposes, role names are included for all association ends. The corresponding ORM schema is shown in Figure 9.54(b), where the derivation rule is specified in relational style.

Figure 9.52 Association redefinition in (a) UML and (b) ORM.

Figure 9.53 Derivation of area in (a) UML and (b) ORM.

Figure 9.54 Derived uncle association in (a) UML and (b) ORM.

Although precise role names are not always elegant, the use of role names in derivation rules involving a path projection can facilitate concise expression of rules, as shown here in the UML model. By adding role names to the ORM schema, the derivation rule may be specified compactly in attribute style thus: *** define** uncle **of** Person **as** brother **of** parent **of** Person. More complex derivation rules can be stated informally in English or formally in a language such as OCL.

One advantage of ORM's approach to derivation rules is that it is more stable, since it is not impacted by schema changes such as attributes being later remodeled as associations.

9.8 Mapping from ORM to UML

The *UMLmap procedure* in Table 9.3 provides basic guidelines for mapping ORM schemas to UML class diagrams. Selected entity types and value types map to object classes and data types, including attribute domains when associations are replaced by attributes. We now illustrate this procedure. As a preparatory move, step 1 binarizes any sets of exclusive binaries, as shown in Figure 9.55.

Table 9.3 UMLmap procedure.

Step	*Action*
1	Binarize any sets of exclusive unaries
2	Model selected object types as classes, and map a selection of their *n*:1 and 1:1 associations as attributes. To store facts about a value type, make it a class
3	Map remaining unary fact types to Boolean attributes or subclasses
4	Map *m:n* and *n*-ary fact types to associations or association classes. Map objectified associations to association classes
5	Map ORM constraints to UML graphic constraints, textual constraints, or notes
6	Map subtypes to subclasses, and if needed, subtype definitions to textual constraints
7	Map derived fact types to derived attributes/associations, and map semi-derived fact types to attributes/associations plus derivation rules

In step 2, we decide which object types to model as classes and which *n*:1 and 1:1 ORM associations to remodel as attributes. Typically, entity types that play functional fact roles become classes. Functional binary (*n*:1 and 1:1) associations from an entity type *A* to a value type B, or to an entity type *B* about which you never want to record details, usually map to an attribute of *A*. If you have specified role names, these can usually be used as attribute names, with the object type name becoming the attribute's domain name.

The mapping in Figure 9.56 illustrates several of these step 2 considerations, as well as step 6 (map ORM constraints to UML graphic constraints, textual constraints, or notes). The {P} and {U1} annotations for preferred identifier and uniqueness are not standard UML. The value constraint on gender codes is captured using an enumeration type.

In rare cases, value types that are independent, play an explicit mandatory role, or play a functional fact role in an 1:*n* fact type map to classes. The example in Figure 9.57(a) deals with cases where we store title–gender restrictions (e.g., the title 'Mr' is restricted to the male gender). The example in Figure 9.57(b) uses a multivalued attribute to store all the genders applicable to a title (e.g., the title 'Dr.' applies to both male and female genders). The Title class gives fast access from title to applicable gender, but slow access from gender to title. As discussed earlier, multivalued attributes should be used sparingly.

Figure 9.55 Step 1: replace any set of exclusive binaries by a binary fact type.

Figure 9.56 Step 2: map selected *n*:1 and 1:1 associations to attributes.

Figure 9.57 Step 2: rare cases of value types mapping to classes.

In step 3 we map unaries to Boolean attributes or to subclasses. The example in Figure 9.58 assumes a closed world interpretation for the unary. With an open world approach, the isSmoker attribute is assigned a multiplicity of [0..1] and the {complete} constraint is removed from the subclassing. Open/closed world aspects are discussed in more detail in Chapter 10.

Figure 9.58 Step 3: map unaries to Boolean attributes or subclasses.

Figure 9.59 Step 4: map remaining fact types to associations.

In step 4, the remaining fact types are mapped to associations. Any *m:n* associations should normally remain that way. In the example in Figure 9.59, the *n*:1 fact type is retained as an association because it relates two entity types that become classes in the mapping. Even if the *m:n* association did not apply, we would normally retain Country as a class, since now or later we are likely to record details for it (e.g., country name).

If an *m:n* association involves a value type (e.g., Employee has PhoneNr) instead of using a multivalued attribute, see if it is possible to transform the *m:n* association into multiple *n*:1 associations (e.g. Employee has PhoneNr1; Employee has PhoneNr2, etc.). Conceptual schema transformations like this are discussed in depth in Chapter 14.

If each object type in an *n*-ary fact type should map to a class (e.g., it plays other functional roles), then map the *n*-ary fact type to an *n*-ary association. Figure 9.60 provides an example.

If an object type in a ternary fact type should not map to a class (typically an *m:n:1* uniqueness pattern with it outside the uniqueness constraint), then objectify the rest of the association as an association class and map its role as an attribute. Figure 9.61 shows an example.

Figure 9.60 Step 4: map some *n*-ary fact types to *n*-ary associations.

Figure 9.61 Step 4: map some *n*-ary fact types to association classes.

Objectified associations map to association classes, as shown earlier in Figure 9.18. Some cases of coreference could be mapped into qualified associations, but mapping to separate attributes or associations supplemented by a textual composite uniqueness constraint offers a more general solution.

In step 5, the simplest constraints in ORM usually map in an obvious way to multiplicity constraints, as illustrated earlier. The more complex ORM constraints have no graphic counterpart in UML, so you need to record these separately in textual form. Table 9.4 summarizes the main correspondences. Join constraints are considered in the next chapter.

In step 6, subtypes are mapped to subclasses, adding relevant subclassing constraints. Subtype definitions are handled with discriminators and/or textual constraints. For example, the ORM schema considered earlier in Figure 9.38 maps to the UML schema in Figure 9.37.

Table 9.4 Mapping main ORM graphic constraints to UML (for use in step 5).

ORM constraint	*UML*
Internal UC	Maximum multiplicity of 1, or {Un}
External UC	Qualified association or textual constraint
Simple mandatory	Minimum multiplicity of 1, or textual constraint
Inclusive-or	Textual constraint (unless within exclusive-or)
Frequency	Multiplicity or textual constraint
Value	Enumeration or textual constraint
Subset and Equality	Subset(s) or textual constraint
Exclusion	Textual constraint (unless within exclusive-or)
Ring constraints	Textual constraint
Join constraints	Textual constraint
Object cardinality	Class multiplicity

In step 7, we map derived and semi-derived fact types. The schemas in Figure 9.53 and Figure 9.54 provide simple examples.

With these hints, and the examples discussed earlier, you should now have enough background to do the mapping manually for yourself. The following short exercise will give you some practice at this. If you are keen on using UML for data modeling, you may wish to use UML to model some of the many modeling exercises in other chapters.

Exercise 9.8

1. Model the following application in UML. We suggest that you do an ORM model first, but that's up to you. The universe of discourse is based on a simplified fragment of a database application used by an electrical utility to help manage the delivery of electricity to consumers in Australia. A line link is a section of an electrical feeder or mains power line that connects two adjacent nodes (e.g., power poles, pillar-boxes).

   The figure shown illustrates a tiny part of the electrical power network. Here three power poles (also called telephone poles) carry power lines to supply electricity. The house is one of many receiving electricity from the power company.

The following output report provides an extract of sample data about line links. The link types listed are not exhaustive.

Line link	*Link type*	*Linktype description*	*Voltage (V)*	*Current (A)*	*Power (kW)*	*Length (m)*
33	OW	open wire	11000	400	4400	200
40	UGC	underground cable	11000	400	4400	150
55	ABC	aerial bundled conductor	415	300	124.5	170
56	ABC	aerial bundled conductor	240	200	48	180

The following report shows details about line links to help with fault detection and correction. Each power line link has a switch at both ends. The switches can be remotely toggled open (to break the circuit) or closed. Line status has only two values as shown.

Line link	*From node*	*To node*	*Status*	*Start switch*	*End switch*
33	B	D	OK	Closed	Closed
40	D	E	OK	Closed	Open
55	F	G	Faulty	Open	Open
56	F	G	OK	Closed	Closed

The following report extract lists details about nodes and any consumers serviced with electricity by those nodes. Only nodes with a transformer (i.e., nodes for which we record a transformer type) can service a consumer. A consumer may be serviced by many nodes and has exactly one, two, or three phases supplied (regardless of which node supplies the power). The transformer types shown here (PT = Pole Transformer, PMT = Pad-mounted transformer, GT = Ground Transformer) are not meant to be exhaustive. Codes of transformer types are stored but descriptions are not.

Note: In reality a consumer has only one node for normal service, and possibly other nodes for backup in the case of a fault—however, we ignore this refinement since its modeling requires subtypes, which we avoid in this simple exercise. Data models of electrical applications usually include many subtypes.

Node	*Nr fuses*	*Transformer type*	*Consumer serviced*	*Nr phases supplied*
B	3	PT	?	?
D	0	?	?	?
E	1	PMT	3001	2
			3005	3
…	…	…	…	…
F	2	GT	5678	2
			5700	1
…	…	…	…	…
G	2	GT	5678	2
…	…	…	…	…

The following report extract lists details about electrical faults suffered by consumers. Timestamps may be treated as a single value (including both date and time of day).

Consumer	*Fault type*	*Reported*	*Fixed*	*Fault cause*
3005	Dim lights	2006-04-29 09:30	2006-04-30 10:45	?
	Dim lights	2007-07-12 17:25	?	Wires down
5678	Blackout	2006-04-30 11:20	2006-05-01 6:30	Lightning strike
	Cold water	2006-04-30 11:20	?	?

The following report extract indicates electrical energy consumption (in kilowatt hours) by consumers on a quarterly basis. Some consumers (e.g., recent ones) might not yet appear in the report. Although consumption can be derived from other data not shown here, assume that it is simply stored for this exercise.

Consumer	*Quarter*		*Consumption*
	Year	*QtrNr*	*(kW.h)*
3001	200	4	1622
	6	1	1500
	200	2	2001
	7	…	…
5678	200	2	2001
…	7	…	…
	…		
	200		
	7		
	…		

(a) Specify a UML schema for this UoD, including constraints and derivation rules.
(b) Suppose that in the model, Fault is an association class to objectify the association Consumer reported FaultType at Instant. Suggest a new identification scheme for Fault to improve the usability of the model.

2. Model the academic schema in Exercise 8.5, Question 2 in UML.

9.9 Summary

The Unified Modeling Language (UML) has been adopted by the OMG as a method for object-oriented analysis and design. Although mainly focused on the design of object-oriented programming code, it can be used for modeling database applications by supplementing its predefined notations with user-defined constraints.

UML 2 includes 13 main diagram types, comprising 6 for structure (class, object, component, deployment, package, and composite structure), and 7 for behavior (use case, state machine, activity, sequence, collaboration, interaction overview, and timing). When stripped of implementation details, class diagrams are essentially an extended form of ER diagrams minus a standard notation for value-based identification.

The basic correspondence between data structures and instances in UML and ORM is summarized in Table 9.5. Classes are basically entity types and are depicted as named rectangles, with compartments for attributes and operations, etc. In UML, facts are stored either in attributes of classes or in associations among two or more classes. Binary associations are depicted as lines. Ternary and longer associations include a diamond. Role names may be placed at association ends, and an association may be given a name. An association may be objectified as an association class, corresponding to nesting in ORM. Associations may be qualified to provide a weak form of co-reference.

Attributes and association ends may be annotated with multiplicity constraints that indicate both optionality and cardinality (e.g., 0..1 = at most one, 1 = exactly one, * = zero or more, 1..* = one or more). Attributes have a default multiplicity of 1, and association ends have a default multiplicity of *. Refer back to Table 9.4 for the main correspondences between constraints in UML and ORM.

Table 9.5 Correspondence between ORM and UML data instances and structures.

ORM	*UML*
Entity	Object
Value	Data value
Object	Object or Data value
Entity type	Class
Value type	Data type
Object type	Class or Data type
— { use relationship type }	Attribute
Unary relationship type	— { use Boolean attribute }
2^+-ary relationship type	Association
2^+-ary relationship instance	Link
Nested object type	Association class
Co-reference	Qualified association §

§ = incomplete coverage of corresponding concept

Subset constraints are allowed only between whole associations and are denoted by "{subset}" next to a dashed arrow. An exclusive-or constraint is depicted by "{xor}" next to a dashed line connecting the relevant associations.

Subclasses are connected to their superclasses by a line with an open arrowhead at the superclass end. Subclassing may be annotated using the keywords "{complete}", "{incomplete}", "{disjoint}" and "{overlapping}", "{root}", "{leaf}". A discriminator (e.g., gender) may be used to indicate the basis for a subclass graph.

Whole/part associations may be displayed as aggregations using a small diamond at the whole end. A hollow diamond denotes shared aggregation (a part may belong to more than one whole), and a filled diamond indicates composition or composite aggregation (a part may belong to at most one whole at a time).

Attributes may be assigned initial (default) values. Derived attributes and associations are indicated by prepending "/" to their name. Attributes and binary association roles may be assigned a changeability setting: unrestricted, readOnly, addOnly, or removeOnly. ReadOnly means that once an attribute value or link has been inserted, it cannot be updated or deleted, and no additional values/links may be added to the attribute/association (for the constrained object instance). AddOnly means that although the original value/link cannot be deleted or updated, other values/links may be added to the attribute/association (for the constrained object instance).

A multivalued attribute or multivalued association end may be adorned with "{ordered}" to indicate implementation as an ordered set. One way of modeling this in ORM is to explicitly introduce a Position type to indicate the order.

An association may be redefined by declaring an association role to be a special case of a compatible role played by a superclass. One use of this is to strengthen the constraints on the specialized association roles.

UML models are best developed by mapping them from ORM models and noting any additional ORM constraints as comments.

Chapter Notes

The UML specification is accessible online at www.omg.org/uml/. For a detailed discussion of UML by "the three amigos" (Booch, Rumbaugh, and Jacobson), see Booch et al. (1999) and Rumbaugh et al. (1999). Their suggested modeling process for using the language is discussed in Jacobson et al. (1999). Martin and Odell (1998) provide a general coverage of object-oriented modeling using the UML notation. Muller (1999) provides a detailed treatment of UML for the purposes of database modeling. A thorough discussion of OMT for database applications is given in Blaha and Premerlani (1998), although its notation for multiplicity constraints differs from the UML standard. The Object Constraint Language is covered in detail in Warmer and Kleppe (2003). Bennett, McRobb, and Farmer (2006) provide a detailed discussion of how to use UML 2.0 for object-oriented systems analysis and design.

On the topic of aggregation, Rumbaugh et al. (1999, p. 148) argue:

> Aggregation conveys the thought that the aggregate is inherently the sum of its parts. In fact, the only real semantics that it adds to association is the constraint that chains of aggregate links may not form cycles … Some authors have distinguished several kinds of aggregation, but the distinctions are fairly subtle and probably unnecessary for general modeling.

There are plenty of other distinctions (apart from aggregation) we could make about associations, but don't feel compelled to do so. For a very detailed discussion arguing for an even more thorough treatment of aggregation in UML, see Barbier et al. (2000).

The view that security issues have priority over changeability settings is nicely captured by the following comment of John Harris, in a thread on the InConcept Web site:

> Rather than talk of "immutable" data I think it is better to talk of a privilege requirement. For instance, *you* can't change your recorded salary but your boss can, whether it's because you've had a pay rise or because there's been a typing error. Privileges can be as complicated or as simple as they need to be, whereas "immutable" can only be on or off. Also, privileges can be applied to the insertion of new data and removal of old data, not just to updates.

A collection of readings critiquing UML is contained in Siau and Halpin (2000). The Precise UML group, comprised largely of European academics, has published several papers mainly aimed at providing a more rigorous semantic basis for UML. A useful collection of their papers is accessible from their Web site *www.puml.org*.

This page intentionally left blank

10 Advanced Modeling Issues

10.1 Join Constraints

10.2 Deontic Rules

10.3 Temporality

- Modeling temporal aspects
- Dynamic constraints
- Rigid subtypes and role subtypes

10.4 Collection Types

10.5 Nominalization and Objectification

10.6 Open/Closed World Semantics

10.7 Higher-Order Types

10.8 Summary

10.1 Join Constraints

This chapter considers several advanced modeling topics and assumes familiarity with the basics of ORM, Barker ER, and UML class diagrams. While the main focus is on ORM, some attention is also given to the other approaches. Even where ER and UML fail to provide built-in support for the features discussed, many of the ideas could be applied, at least as extensions, to these approaches.

This section examines *join constraints*, first considering join constraints over a single join path, then set-comparison constraints involving joins. Although consideration of these constraints is really part of CSDP step 7, we have postponed a detailed discussion of them until now because of their advanced nature.

The notion of conceptual joins was introduced in Section 4.4 and further illustrated in Section 9.7. With a *conceptual inner join* over two or more roles, the same object plays all the roles. In Figure 10.1(a), there is a single join path Building contains Room **that** has RoomNr. Linguistically, the pronoun "**that**" in this path expression declares the join, indicating it is the same room playing both the join roles (shaded here for discussion purposes).

The external uniqueness constraint declares that each building, roomNr combination applies to at most one room. This is equivalent to an internal uniqueness constraint on building, roomNr in the set of (building, room, roomNr) tuples instantiating the conceptual join path.

Figure 10.1(b) shows the same schema in Barker ER notation. Here the external uniqueness constraint is captured by combining the "#" and "|". Figure 10.1(c) shows the UML version, with the external uniqueness constraint specified in a note.

In the room example, the two roles participating in the conceptual join are both mandatory for the join object type (Room). This ensures that the conceptual join is an inner join. However, if one or more roles involved in the conceptual join are optional, the meaning of the external uniqueness constraint may be ambiguous. As a simple example, consider the two ORM schemas shown in Figure 10.2.

Figure 10.1 A simple conceptual join in (a) ORM, (b) Barker ER, and (c) UML.

Figure 10.2 Join type ambiguity may arise with optional roles.

In Figure 10.2(a), each course must have a title, but may or may not be offered by a department (e.g., it could be a department course, a general course, or a course offered by a visiting lecturer). The external uniqueness constraint indicates that if a course is offered by a department, then the combination of the department and course title is unique. So, within a given department each course has a different title. But if two courses are not offered by a department, must they have different titles? There are two reasonable answers to this question: yes or no. In principle, we could respond unknown, but in practice this is not an option because we require the semantics to be well defined. At any rate, the constraint is ambiguous unless we agree to always answer such a question in the same way.

If we answer yes to the question, then the same title may apply to at most one course with no department. The same title may, however, apply to many courses in different departments. In terms of the underlying join path for the constraint, this interpretation applies conceptual outer joins to optional roles, with the added proviso that null values are treated as ordinary values. The join of the title and department fact types results in a compound ternary fact type equivalent to the following relational table scheme.

Course(courseCode, courseTitle, [departmentCode])

C1	Mechanics	PY	
C2	Mechanics	MA	
C3	Mechanics	?	
C4	*Mechanics*	*?*	*-- violates constraint*

This relational notation underlines uniqueness constraints; if more than one exists, the primary key is doubly-underlined. Optional columns are marked with square brackets. The external uniqueness constraint appears as the internal uniqueness constraint spanning courseTitle and departmentCode.

The sample population shows three courses, C1, C2, and C3, with the same title, one offered by the physics department, one by the mathematics department, and one offered by no department. The fourth row (C4, Mechanics, ?) is rejected because in the presence of row 3 it violates the constraint that courses with no department must have different titles. For the purpose of comparison, the nulls on rows 3 and 4 are treated like ordinary values, and thus are equated. This effectively applies *outer join semantics*.

To simplify the discussion, each course is identified by a course code (e.g., 'C1'), but this is not essential, since the semantics may apply even if the disjunctive reference scheme is the only means of identifying courses. The outer join interpretation is reasonable for the course example and is required in many practical situations.

Now consider the ORM schema in Figure 10.2 (b), which is a fragment of an ORM metaschema. ORM requires each predicate to have at least one reading, but it is optional whether roles are given names. In the figure, the role played by RoleName is named "name", but the other roles are not named. The external uniqueness constraint indicates that role names are unique within a given predicate. However, many roles within the same predicate may have no name.

Unlike the course example, this situation requires an *inner join interpretation* for the join path underlying the external uniqueness constraint. Let us identify the predicate of the association Role is in Predicate as P1 and its left and right roles as r1 and r2, respectively. Let us also identify the predicate of the association Role has RoleName as P2 and its left and right roles as r3 and r4, respectively. The join of the two fact types results in a compound ternary fact type equivalent to the following relational table scheme.

Role(roleId, predicateId, [roleName])

R4	P2	name

The external uniqueness constraint appears as the internal uniqueness constraint. Since we only require roles to have unique names in a predicate if they are named, the conceptual join between the roles of Role in Figure 10.2 (b) is an inner join, resulting in the single row population shown. For comparison, the outer join result would include four tuples : (r1, P1, ?), (r2, P1, ?), (r3, P2, ?), and (r4, P2, name). If we were to take the outer join interpretation as we did for the course model, the first two tuples would violate the compound uniqueness constraint.

Although this example might seem unusual, we have encountered several real world cases involving this inner join semantics. In practice, *uniqueness constraints involving optional roles may have either inner join or outer join semantics. So we need a way to distinguish them.* This can be done graphically, textually, or both. It can also be done on a role basis or a constraint basis.

A complete solution to the problem theoretically requires that each join on the path be identified in some way as an inner or outer join. The NORMA tool for ORM 2 is currently being extended to support this. For attribute-association approaches such as ER and UML, an ORM-like notation could be added for linking association roles only, association roles and attributes, as well as just attributes.

In the previous examples, the join path underlying the external uniqueness constraint involves only one join. But in practice, the join path may involve many joins. We consider a complex example of this in a later exercise. Figure 10.3 summarizes the problem of *join path ambiguity*. To find the join path underlying the external uniqueness constraint, we need to find a path connecting roles r2 and r9, moving right from r2 and left from r9.

Figure 10.3 Join paths may require explicit declaration of the join roles.

Our choice is shaded, but there are three possible paths between *B* and *C*. We could traverse the top predicate, performing the joins r2 = r3 and r4 = r9. We could traverse the middle predicate that implicitly underlies the subtype connection, performing the joins r2 = r5 and r6 = r9, or we could traverse the bottom predicate, performing the joins r2 = r7 and r8 = r9.

Currently ORM does not graphically highlight the relevant joins (as we have done here using shading). In general, shading is not adequate since a path may traverse the same predicate more than once. A general graphical solution may be provided by numbering the roles used to perform the joins, but this can lead to messy diagrams. A tool could prompt the user to select the list of join roles and then toggle the display of the joins on/off under user control. An alternative is to use a textual language to formulate constraints with multiple, candidate join paths.

For non-ORM approaches, first basic support for external uniqueness is required, and then the join paths may be disambiguated graphically as for ORM or textually if the method includes a sufficiently powerful formal query language. UML could use the Object Constraint Language (OCL) for such specifications, although OCL expressions often appear cryptic to nontechnical domain experts.

As for external uniqueness, external frequency constraints have an underlying join path (for an example, refer back to Figure 7.6). If the joins involve optional roles, then the join type (inner or outer) is potentially ambiguous. However we have never encountered a practical example of this, so we recommend that inner joins be assumed and that a formal textual language be used instead of a graphic notation if ever the outer join interpretation is required.

Recall that ORM *set-comparison constraints* apply a subset, equality, or exclusion constraint between compatible role sequences. Sometimes one or more of these role sequences are projected from a join path. As a simple example, the join subset constraint in Figure 10.4(a) runs from the role pair comprising the Advisor serves in Country fact type to the role pair comprising the first roles of the Advisor speaks Language and Country uses Language fact types.

The constraint means that the set of advisor–country pairs populating the serves-in predicate must be a subset of the set of advisor–country pairs obtained by projecting on the advisor and country roles of the join path Advisor speaks Language **that** is used by Country.

Figure 10.4 A join subset constraint in (a) ORM and (b) UML.

This join path involves a conceptual join on Language, equating instances playing its two roles. In other words, if an advisor serves in a country, then that advisor must speak at least one language used by that country. This constraint may be verbalized formally in ORM as: **Each** Advisor **who** serves in **a** Country **also** speaks **a** Language **that** is used by **that** Country.

Like ER, UML has no graphic notation for expressing a join subset constraint, but does allow the constraint to be expressed textually in a note attached to the relevant model elements (in this case the three associations), as shown in Figure 10.4(b). ER could also be adapted to allow such usage of notes.

Figure 10.5(a) shows another example discussed earlier in the book. The constraint may be verbalized in ORM thus: **If a** Person has **a** PersonTitle **that** is restricted to **a** Gender **then that** Person is of **that** Gender. A sample population illustrates the title–gender restriction.

Figure 10.5 Another join subset constraint in (a) ORM and (b) UML.

Figure 10.6 A join equality constraint.

Figure 10.5(b) shows one way to model this in UML. The title–gender restriction fact type can't be expressed because title and gender are modeled as attributes. An informal note has been added as a reminder for humans to look after the subset constraint themselves.

Join subset constraints often occur in practical applications. For a more complex join subset involving a ternary, look back at the room facility example discussed in Figure 9.42.

Join equality constraints are less common. The ORM schema in Figure 10.6 includes an equality constraint between role triples, where the first triple involves a join on an objectified association. Role numbers are displayed here to clarify the constraint arguments (the NORMA tool displays these numbers only when the constraint is entered or selected). The constraint captures the rule that tips must be reported for, and only for, those services that earn a tip.

The ORM schema in Figure 10.7 includes a *join exclusion constraint* between two role pairs. Each role pair is projected from a path involving a join on Person. The role pairs are numbered 1.1, 1.2 and 2.1, 2.2. Since the role played by Institute occurs in both role pairs, it is numbered twice. The constraint captures the rule that no person can review a paper written by someone from their own institute, a rule commonly enforced in academic conferences. To avoid cluttered diagrams, the NORMA tool displays the role numbers only when the constraint is entered or selected. The constraint verbalization is also available to clarify the meaning of the constraint.

Figure 10.7 A join exclusion constraint.

The same considerations given to join paths underlying external uniqueness and frequency constraints may be raised for each join path involved in a set-comparison constraint. Although we have modeled numerous business domains involving join set-comparison constraints, the join types have always been inner joins. External ring constraints, as well as value-comparison constraints, can also involve a join. For value-comparison constraints, the join type is always inner.

Rather than complicating the graphic notation for rare cases, we recommend that any set-comparison or ring constraints involving outer joins be specified textually using a formal language.

The mapping of join constraints depends on the mapping target, typically a DBMS or 3GL. Commercial SQLs often diverge from standard SQL, and special care is needed when mapping uniqueness constraints over nullable columns. For example, suppose we want to implement the composite uniqueness constraint on the last two columns of *Role* (roleId, predicateId, [roleName]) in SQL Server. SQL Server allows only one null if a unique constraint is declared, but we want to allow multiple nulls. To implement inner join semantics for this composite uniqueness constraint, we need to code it up explicitly. The logic involved may be set out thus:

```
check( not exists
   (select predicateId, roleName from Role
    where roleName is not null
    group by predicateId, roleName
    having count(*) > 1) )
```

If this code is unintelligible to you, don't worry. Chapters 12 and 13 provide all you need to know about SQL to handle cases like this.

Exercise 10.1

1. A sample population is included for each of the following ORM schemas, using surrogate identifiers to save space. In case (A), but not case (B), the following constraint applies: for each person, at most one employment end date is unknown (we have a complete record of their employments). The relational schema is also shown, using underlines to indicate uniqueness constraints.
   (a) Complete the population of the relational table for each situation.

Employment_A

enr	*pnr*	*startdate*	*enddate*

Employment_B

enr	*pnr*	*startdate*	*enddate*

   (b) What kind of join semantics (inner or outer) applies to the uniqueness constraint over (pnr, enddate) for schema A?

(c) What kind of join semantics (inner or outer) applies to the uniqueness constraint over (pnr, enddate) for schema B?
(d) Specify a check clause in SQL to check this uniqueness constraint for schema A.
(e) Specify a check clause in SQL to check this uniqueness constraint for schema B.

Employment (enr, pnr, startdate, [enddate])

Employment (enr, pnr, startdate, [enddate])

2. Add a graphical constraint to the following ORM schema to capture the rule that if a modeling approach is used on a task that requires a specific modeling level, then the modeling approach must support that level.

3. Add a graphical constraint to the following ORM schema to ensure that nobody reviews any book that they author.

10.2 Deontic Rules

Business domains are constrained by various *business rules*, which specify required or desirable states of affairs or behavior. Business rules may be of different *modalities* (e.g,. alethic and deontic). *Alethic rules* impose necessities, which cannot, even in principle, be violated by the business, typically because of some physical or logical law. For example, each employee was born on at most one date; no product is a component of itself.

All the constraints we've considered so far are alethic. The point of including them in an information system is not to prevent the rules being violated in the business domain (since they cannot be violated there), but to ensure that the information model accurately reflects the business domain. Alethic rules simply prevent data representing impossibilities from being entered into the system. For example, if the information system enforces the uniqueness constraint on the employee birth fact type this ensures that we don't record two birthdates for the same employee.

Deontic rules impose obligations, which may be violated, even though they ought not. For example, in most cultures it is obligatory that each employee is married to at most one person, and in most businesses it is forbidden that any person smokes in any office.

A business should have a clear understanding of all its rules, including deontic ones, whether or not it chooses to enforce these rules, or monitor violations of them, by means of an automated system. In recognition of this need, both ORM 2 and the Object Management Group's Semantics of Business Vocabulary and Rules (SBVR) specification support the formulation of deontic rules (OMG, 2007). Although ER and UML currently confine their specification of rules to those of an alethic modality, in principle these approaches could be extended to cater for deontic rules.

Rule formulations may use any of the basic alethic or deontic modal operators from modal logic, as shown in Table 10.1. These modal operators are treated as proposition-forming operators on propositions (rather than actions). Equivalent readings in other concrete syntaxes may be used (e.g., "necessary" might be replaced by "required", and "is obligatory" might be replaced by "ought to be the case").

The following *modal negation rules* apply: it is not necessary that ≡ it is possible that not ($\sim\Box p \equiv \Diamond\sim p$); it is not possible that ≡ it is necessary that not ($\sim\Diamond p \equiv \Box\sim p$); it is not obligatory that ≡ it is permitted that it is not the case that ($\sim\boldsymbol{O}p \equiv \boldsymbol{P}\sim p$); it is not permitted that ≡ it is obligatory that it is not the case that ($\sim\boldsymbol{P}p \equiv \boldsymbol{O}\sim p$).

Table 10.1 Alethic and deontic modal operators.

Alethic		*Deontic*	
Reading	*Symbol*	*Reading*	*Symbol*
It is necessary that	□	It is obligatory that	***O***
It is possible that	◊	It is permitted that	***P***
It is impossible that	~◊	It is forbidden that	***F*** or ***~P***

In ORM 2, each constraint or rule has an associated modality, determined by the logical modal operator that functions explicitly or implicitly as its main operator. In positive verbalizations, an alethic modality of necessity is often assumed (if no modality is explicitly specified), but may be explicitly prepended. For example, the constraint **Each** Person was born in **at most one** Country may be explicitly verbalized with an alethic modality thus: **It is necessary that each** Person was born in **at most one** Country.

We interpret this constraint in terms of possible world semantics, as introduced by Saul Kripke and other logicians in the 1950s. A proposition is necessarily true if and only if it is true in all possible worlds. With respect to a static constraint declared for a given business domain, a possible world corresponds to a state of the fact model that might exist at some point in time. The aforementioned constraint declares that for each state of the fact model, each instance in the population of Person is modeled to be born in at most one country.

A proposition is possible if and only if it is true in some possible world. A proposition is impossible if and only if it is true in no possible world (i.e., it is false in all possible worlds). In ORM, the above constraint may be reformulated as the negative verbalization: **It is impossible that the same** Person was born in **more than one** Country.

To avoid confusion, when declaring a deontic constraint, the deontic modality should always be explicitly included. Consider the following static, deontic constraint, verbalized in both positive and negative forms:

(+ve) **It is obligatory that each** Person is a husband of **at most one** Person.
(-ve) **It is forbidden that the same** Person is a husband of **more than one** Person.

This deontic rule indicates a condition that *ought* to be satisfied, while recognizing that the condition *might* not be satisfied. Including the obligation operator makes the rule much weaker than a necessity claim, since it allows that there could be some states of the fact model where a person is a husband of more than one wife (excluding same-sex unions from instances of the husband relationship). For such cases of polygamy, it is important to know the facts indicating that the person has multiple wives. Rather than reject this possibility, we allow it and then typically perform an action that is designed to minimize the chance of such a situation arising again (e.g. send a message to inform legal authorities about the situation).

Figure 10.8(a) shows an ORM schema with some alethic and deontic constraints. When displayed in color, alethic constraints are violet, and deontic constraints are blue. Deontic constraint markers typically include a small "o" (for obligatory).

Figure 10.8 Some ORM schemas with alethic and deontic constraints.

The citizenship and marriage fact types have spanning uniqueness constraints, and hence are alethically many-to-many associations. However, each role of the marriage association has a deontic uniqueness constraint, so it is obligatory that each person has at most one spouse. The mandatory role dot on the citizenship role is an open "o" rather than solid, so that mandatory constraint is deontic, indicating that it is obligatory that each person is a citizen of some country.

Figure 10.8(b) shows an ORM schema with a deontic uniqueness constraint spanning the first two roles of a ternary fact type. This verbalizes in positive form as **It is obligatory that for each** Room **and** HourSlot, **that** Room at **that** HourSlot is booked for **at most one** Activity. In negative form, we have: **It is forbidden that the same** Room at **the same** HourSlot is booked for **more than one** Activity. Notice that in both schemas *the deontic uniqueness constraints span a subset of the roles spanned by the relevant alethic constraints (explicit or implicit)*. If an alethic constraint already applies to a role set, a deontic constraint over the same roles is pointless, as it can't be violated.

In practice, most business rules include only one modal operator, and this operator is the main operator of the whole rule expression. While ORM is restricted to rules of this nature, SBVR allows modal operators to be embedded anywhere in a rule formulation. The use of embedded modal operators, especially deontic operators, can dramatically increase the complexity of rule formalization and execution. For a detailed discussion of such issues, including formalization aspects and much more complex examples, see Halpin (2007).

The only impact of tagging a rule as a necessity or obligation is on the rule enforcement policy. Enforcement of a necessity rule should never allow the rule to be violated in the model. Enforcement of an obligation rule may allow model states that do not satisfy the rule condition, and take some other remedial action: the precise action to be taken is not specified here, but a simple default is to generate a message when an update violates the rule. At any rate, a business person should be able to specify a deontic rule first at a high level, without committing at that time to the precise action to be taken if the condition is not satisfied; of course, the action still needs to be specified later in refining the rule to make it fully operational.

The ORM glossary at the back of this book lists the symbols for all the alethic and deontic constraints supported in ORM 2.

Exercise 10.2

1. Which of the following statement pairs is/are a pair of *equivalent* rule statements?
   (a) It is necessary that each Person died on at most one Date.
   It is forbidden that the same Person died on more than one Date.
   (b) It is necessary that each Country is of at most one Area.
   It is impossible that the same Country is of more than one Area.
   (c) It is obligatory that each Person has at most one SocialSecurityNumber.
   It is forbidden that the same Person has more than one SocialSecurityNumber.
   (d) It is not permitted that any Person is a husband of more than one Person.
   It is obligatory that no Person is a husband of more than one Person.

2. Identify any errors in the following schema and explain why they are errors. Ignore the missing ring constraints.

3. A given business identifies employees by employee numbers. All employees have names, not necessarily unique. All employees should have a social security number, but it is possible that some do not. Safeguards make it impossible for the same employee to be issued more than one social security number. Although no two employees ought to have the same social security number, it is possible for this to happen. Model this UoD.

10.3 Temporality

In the database community, various proposals and products have been developed to address temporal data. For example, SQL/Temporal was considered as a temporal extension to SQL (on hold since late 2001, but might be revived) and many DBMS products have incorporated specific support for temporal features. Three basic *temporal data types* may be distinguished (Snodgrass 2000):

- *Instant* (point in time, e.g., 2008 June 3, 2:50 p.m. UTC)
- *Interval* (duration of time, e.g., 3 weeks)
- *Period* (anchored duration of time, e.g., 2008 June 3 ... 2006 June 23 MST)

For each of these data types, various temporal operators may be defined. For example, date subtraction (–) applied to an ordered pair of dates (e.g. today and your birthdate) results in an interval (e.g., your age), and the overlaps operator applied to two periods yields a Boolean value (True or False) indicating whether those periods overlap in time. Many such temporal operators (e.g. before, after, during) have been defined, and are useful for specifying temporal queries and dynamic rules.

Currently, the only data type with good support from the standard version of most DBMSs is Instant (e.g. Date, Time, Timestamp). SQL-92 added support for Interval, but most commercial DBMSs have yet to provide this support. Period was initially included in the SQL/Temporal proposal for SQL:1999, but this temporal extension for SQL is on-hold and is not even included in SQL:2008. In the meantime, periods may be specified in the SQL standard either by start and end times, or by a start time and an interval. However, sophisticated support for temporal data is available in extensions (e.g., spatio-temporal data cartridges/blades) to some of the major DBMSs and in specific temporal database products.

In database work, two kinds of time are often distinguished. *Valid time* denotes the time period during which a database fact was, is, or will be valid in the business domain being modeled. In contrast, *transaction time* denotes the time period during which a database fact is/was recorded in the database. In a database table, valid time and transaction time may often be captured by extra columns (e.g., a mandatory startVT and an optional endVT column, or a mandatory startTT and an optional endTT). If the end transaction time is unknown, it is sometimes recorded as the largest time stamp supported.

Different facts may be recorded with different *granularities of time* (e.g., second, day). Time may be measured using different *calendars* (e.g., Gregorian, Julian) and different *time zones* (e.g., UTC, Mountain Standard Time). For most modeling tasks, calendar and time zone choices are taken to be implicitly understood.

Temporal extensions have been proposed for the relational model, as well as higher level data modeling approaches such as ER, ORM, and UML. Basic *temporal object types* (e.g., Date or Period) may be used just like other object types in the business model, with basic temporal operators (e.g., –) assumed to be predefined. For the rest of this section, time is understood as valid time, not transaction time.

Temporal object types may be classified into two kinds: once only and repeatable. Each *once-only temporal object* is either a single *instant* or a single *period* of time, measured to a specified temporal granularity (relative to an assumed calendar and time zone). Some simple examples are depicted in Figure 10.9(a), using ORM notation. For example, Albert Einstein was born in the year 1879 CE. In this sense, a year is a 12-month period in the time stream. Likewise, a date is a period of 1 day in the time stream (e.g., 2008 June 5), a month is a period of approximately 30 days in the time stream (e.g., 2008 June), an hour is a period of 60 minutes (e.g., 2008 June 5, 9–10 a.m.). A time is an instant in the time stream measured to a specified granularity, such as a second, e.g., 2008-06-05 14:35:27 (i.e. 2008 June 5, 14 h 35 m 27 s). Object types like these are useful for recording when an individual event happened or will happen (e.g. the birth of Einstein, or the next Olympic Games).

Each *repeatable temporal object* corresponds to a set of instants or time periods, measured to a specified granularity (relative to an assumed time zone). Some simple examples in ORM notation are depicted in Figure 10.9(b). Here a weekday is a day slot of any week (e.g., Sunday), a monthOfyear is a month slot of any year (e.g., January, or Month 1), and an hour slot is a 1-hour slot of a weekday (e.g., Monday starting at 9 a.m. and ending at 10 a.m.). Object types like these are useful when modeling schedules (e.g., a workout routine or a conference program).

Figure 10.9 Examples of (a) once-only and (b) repeatable temporal object types.

Figure 10.10 Closed periods modeled in (a) ORM and (b) UML.

Sometimes periods are modeled in more detail by explicitly specifying their start and end times to the desired temporal granularity (year, month, day, hour, etc.). For example, if periods are closed (we know both start and end) and the granularity is day, then they may be modeled in ORM as shown in Figure 10.10(a) and in UML as Figure 10.10(b).

Intervals (unanchored durations of time) may be modeled as a simple object type with a unit for the chosen granularity. As a simple example, if we measure ages of fossils in year units, this may be modeled in ORM as shown in Figure 10.11(a) and in UML as shown in Figure 10.11(b). In ORM, the colon ":" indicates that "y" is a unit of a given type (display here suppressed, but can be expanded to "y: Interval").

As a business domain changes over time, the set of facts describing it also changes. Regardless of how we represent fact types in a model (e.g., as relationships or as attributes), for the purpose of making decisions about how they are impacted by time it is convenient to distinguish at least *three kinds of fact type*:

- *Definitional* (truth of instances is a matter of definition, independent of events)
- *Once-only* (instances correspond to a single event)
- *Repeatable* (instances may correspond to multiple events)

Definitional facts have no temporal aspect. They are necessarily true by definition, rather than depending on any event in the business domain. Figure 10.12(a) shows an ORM diagram for the definitional fact type PolygonShape has NrSides along with a sample fact population. Figure 10.12 (b) shows the same fact type as a UML class diagram

Each *once-only fact* corresponds to a single event. Its truth derives from an event that can never be repeated in that business domain. For example, Einstein was born in Germany. This is true because of his birth event, a state of affairs that happened in 1879 CE.

Figure 10.11 Modeling an interval in (a) ORM and (b) UML.

Figure 10.12 Example of a definitional fact type in (a) ORM and (b) UML.

Whether or not reincarnation is possible, let us ignore such possibilities for the business domain modeled in Figure 10.13, where each employee has exactly one birth event. Figure 10.13(a) depicts the fact type Employee was born in Country in ORM, along with a sample population. Figure 10.13(b) depicts this as a many:1 association in UML, and Figure 10.13(c) depicts this as a single-valued attribute in UML.

Figure 10.13 Example of a once-only fact type in (a) ORM and (b and c) UML.

Each *repeatable fact* corresponds to a set of events, in the sense that its truth is determined by any one of a set of repeatable events. For example, suppose employee 101 visited Belgium many times. The fact that he/she visited Belgium is made true by any one of those visits. Because the same employee may visit the same country repeatedly, the fact type Employee visited Country is said to be repeatable. Figure 10.14(a) depicts the fact type Employee visited Country in ORM, along with a sample population. Figure 10.14 (b) depicts this as a many:many association in UML, and Figure 10.14 (c) depicts this as a multivalued attribute in UML.

Figure 10.14 Example of a repeatable fact type in (a) ORM and (b and c) UML.

A *proposition* (e.g., Employee 101 visited Belgium) must be true or false (and hence is a *truth bearer*). A state of affairs (e.g., Employee 101 visiting Belgium) is actual (occurs or exists in the actual world) or not. A state of affairs may be possible or impossible. Some possible states of affairs may be actual (occur in the actual world). *States of affairs* are thus *truth makers*, in that true propositions are about actual states of affairs. In sympathy with the correspondence theory of truth, we consider the relationship between propositions and states of affairs to be one of correspondence rather than identity.

Definitional facts may be viewed as corresponding to states of affairs only in a trivial sense (if at all); since they have no temporal aspect, they need no special attention in this regard. In contrast, once-only and repeatable facts correspond to states of affairs that are either single events (once only) or event sets (repeatable). *For each once-only or repeatable fact type in a model, we need to determine what (if any) temporal information is needed.*

Let's first consider once-only fact types. If a fact type is once-only and we wish to record for at least some of its fact instances when its underlying event (that made the fact instance true) occurred, then (if not already present in the model) add a functional fact type to record when the event occurred. To do this, relate a "key" of the original fact type to a temporal object type of the desired temporal granularity. Here "key of a fact type" means an object type that plays a mandatory, functional role in that fact type. A functional role is constrained by a simple uniqueness constraint. If we wish to record the event for each of the original fact instances, the key's role in that temporal fact type should normally be mandatory.

For example, consider the once-only fact type Employee was born in Country. Suppose that for each instance of this fact type we want to know when the underlying birth event occurred to a granularity of 1 day. The key object type is Employee (its role is mandatory and functional), so add the mandatory, functional fact type Employee was born on Date. Figure 10.15(a) depicts this in ORM showing a sample population. Figure 10.15 (b) depicts this in UML using a birthdate attribute. If we instead chose the temporal granularity to be a year, then we would replace Date(ymd) by Year(CE), or birthdate by birthyear, and replace the date values by year values (1946, 1980, 1980).

Figure 10.15 Add a temporal fact type to record when once-only facts are "made true".

Figure 10.16 A subset constraint between optional roles in (a) ORM and (b) UML.

The example just discussed includes one temporal fact type (involving a temporal object type such as Date) and one nontemporal fact type (in this case a spatial fact type). Both these fact types are mandatory for employees. In other situations, either or both of these fact types might be optional for employees. If both fact types are optional for employees, but one is known only if the other is known, the relevant subset constraint should be added. For example, Figure 10.16 models the case where we know an employee's birth date only if we know the employee's birth country. The subset constraint depicted graphically in ORM is captured textually in UML using OCL.

We use the term "event" to cover both *point events* (which occur at a single instant of time, e.g., a horse winning a race) and *period events* (which occur over a nonzero period of time, e.g., a rock concert). If the event described in the once-only fact is a period event and we wish to record the start and/or end of that period, then add temporal fact types of the desired temporal granularity.

As an example, suppose the fact type Employee first visited Country is used to record the event when people first visit various countries, and suppose we always know the date when such visits started but only optionally know when they ended. For example, this could record details about your first visit to Belgium, your first visit to Japan, and so on. Figure 10.17 shows one way to model this in ORM and UML, objectifying the *m:n* visit association and attaching fact types for the start date (mandatory) and end date (optional).

Figure 10.17 Recording an open period for a period event in (a) ORM and (b) UML.

Figure 10.18 A coreferencing alternative in (a) ORM and (b) UML.

Alternatively, these two temporal fact types could be replaced by a single fact type linking to Period as a compositely identified object type or class. Yet another alternative is to replace the FirstCountryVisit objectified association by a coreferenced entity type or class, as in Figure 10.18. This alternative approach is suitable for industrial versions of ER that do not support objectification.

Now suppose the fact type Employee lived first in Country is used to record where people lived in the first part of their lifetime (e.g., Einstein lived first in Germany). This fact type is *n:1* rather than *m:n*. In ORM and UML, such associations may be objectified, and we may attach the temporal information as previously (see Figure 10.19(a) and (b)). In modeling approaches that do not allow such objectification, the start and end date information may be attached directly to Employee (see Figure 10.19(c) and (d) for ORM and Barker ER versions of this alternative).

Figure 10.19 Period events modeled by an *n:1* association.

Figure 10.20 A simple patient schema in (a) ORM, (b) UML, and (c) Barker ER.

Now let's discuss how to record *snapshots of states that can change over time*. Consider a medical clinic where patients are identified by patient numbers, have their birthdate recorded, and optionally have their weight, temperature, and allergies (if any) recorded. Figure 10.20 models this domain in ORM, UML, and Barker ER. The ORM and UML schemas include a derived fact type to compute a patient's age. A textual derivation rule may be added to compute age from birthdate and current date (in ORM, the latter is obtained from the implicit, predefined fact type Date is today).

The fact type Patient was born on Date is a once-only fact type, and hence is *unchangeable* (ignoring the possibility of error correction). Each of the other fact types is *individually changeable* (its individual instances may change over time). The fact type Patient has Weight is *functional* for Patient (at any time, a patient has at most one weight) and is repeatable (patients may gain or lose weight and later return to their previous weight).

The temperature fact type is also functional and repeatable. The allergy fact type is *m:n* (many-to-many) and repeatable (patients may cease to be allergic to a given drug, but later become allergic to it again). The derived age fact type is functional and changeable even though it is not repeatable.

Being functional, the age and weight fact types are changeable only at the instance level—an old instance can change only by being replaced by a new instance. The nonfunctional allergy fact type is also *collectively changeable* (its population of instances may change over time by adding new instances or deleting old instances).

The individual changeability of the age, weight, temperature, and allergy fact types follows from their *time-deictic* nature. Deixis (Greek for indicating, or point of reference) is the process whereby the referent of a linguistic structure depends on its communication (utterance or inscription) context and may be classified into five basic categories: time, place, person, discourse, and social. For example, the sentence "You

are now reading this book" is person deictic ("you"), time deictic ("are", "now"), and discourse deictic ("this book").

In time (or temporal) deixis, the meaning of a communication act depends on the time in which it was performed. Temporal deictic references may appear via temporal adverbs (e.g., "now", "currently", "today", "tomorrow", "yesterday"), *tense* (e.g., "is eating", "was eating", "will eat"), combinations of deictic modifiers and time units (e.g., "last year", "this week", "next month"), or other temporal phrases (e.g., "two days ago", "within a month"). In information modeling, time-deictic aspects of predicates are usually restricted to tense and temporal adverbs.

In Figure 10.20(a), the birthdate predicate is in past tense (was born on), while the other four predicates are in present tense (has, has, has, is allergic to). Since one cannot change the past, instances of the birthdate fact type are unchangeable. If we wish our information model to remember instances of birthdate facts, these facts also cannot be deleted. In contrast, instances of the four present tense fact types are changeable and may be replaced by other instances. Consider the following fact instance:

Patient 101 has Weight 80 kg.

This is shorthand for "Patient 101 *currently* has Weight 80 kg", which itself is shorthand for "Patient 101 has Weight 80 kg at time *t*, where *t* is the time at which the sentence was communicated". Originally *t* is a valid time (e.g., the time at which the patient's weight was actually measured). Typically, the fact is then entered into the information model at some later transaction time and remains there until it either is updated by a more recent measurement or is deleted.

From the viewpoint of the information model, the fact really means "*When last inspected*, Patient 101 had Weight 80 kg". For such reasons, the fact is known as a *snapshot* fact. In practice, if the delay between valid and transaction times is minor, and the property (e.g., weight, allergy) is reevaluated on a suitably regular basis (which may be very infrequent if the property changes only slowly over time), then business users are willing to treat snapshot facts as if they were current.

Now let's discuss how to record *history of functional, changeable fact types.* If a fact type is individually changeable, and we need to maintain a *history* (full or partial) of its instances, then we should add or adapt a fact type to achieve this, using a temporal object type of the desired temporal granularity. In ORM and UML, one simple way to do this is to insert into a "key" of the original, snapshot fact type a role that is played by the temporal object type, rephrasing the predicate as appropriate. A key of a fact type is a set of its roles that is exactly spanned by a uniqueness constraint. For example, in Figure 10.20(a) the key of the weight fact type is the role played there by Patient.

Suppose we wish to maintain a history of patient weights where for each patient at most one weight measurement is performed per day. Since our temporal granularity is 1 day, we choose Date as the object type and insert its relevant role into the binary fact type to produce the ternary fact type Patient on Date had Weight and expand the key uniqueness constraint to cover this role as shown in Figure 10.21(a).

Figure 10.21 Different options for maintaining history of patient weights in ORM.

The uniqueness constraint verbalizes as: **For each** Patient **and** Date, **that** Patient on **that** Date had **at most one** Weight. If now or later you may wish to add details about the weight measurement (e.g., who performed the measurement), it is better to introduce a WeightMeasurement type either by objectification (Figure 10.21(b)) or by co-referencing (Figure 10.21(c)). For easy reference, you could also introduce a measurement number to provide a simple identifier, as in Figure 10.21(d), which includes an external uniqueness constraint to ensure that the patient and date combination provides a secondary identifier. If you wish to change the rules to allow the same patient to have more than one weight measurement on the same date, the external uniqueness constraint should be removed from Figure 10.21(d).

In UML and Barker ER, it would be unusual to include Date as a class or entity type, so a WeightMeasurement type would normally be used. If no measurement number is introduced, the schemas shown in Figure 10.22 may be used. As UML has no graphic constraint for external uniqueness, this constraint is declared here informally in a note. It could also be declared formally in OCL.

Figure 10.22 Figure 10.21(c) remodeled in (a) Barker ER and (b) UML.

UML assumes that each object is identified by an internal surrogate. If desired, we could add a measurement number identifier for WeightMeasurement for Barker ER and UML. If the combination of date and patient still provides a secondary identifier for WeightMeasurement, the Barker ER identifier notation used in Figure 10.22(a) to capture this constraint can no longer be used, since it applies only to primary identifiers; so this constraint should be noted elsewhere.

To maintain a history of patient temperatures, we might choose a different temporal granularity. For example, if our granularity is 1 hour, we could use the fact type Patient at Hour had Temperature.

Note that while the aforementioned approaches allow full history (at the desired granularity), they do not demand full history. For example, we are not constrained to record a patient's weight on a daily basis or a patient's temperature on an hourly basis. Note that given date arithmetic, there is no need to add any asserted fact types to Figure 10.20 to provide a history of how patients aged over time (this is derivable).

For comparison purposes, we sometimes require a well-defined *partial history* (e.g., current and previous values only). The basics for such a situation can be modeled easily by using multiple fact types, as shown for patient weights in Figure 10.23. A complete approach requires additional textual rules to assign the old current weight to the new previous weight on update.

Maintaining a *history of nonfunctional fact types* is a little more complicated, since they may be changeable both collectively and individually. Figure 10.24 shows some attempts to model visits of employees to countries, assuming that we know the start date for each visit. Since some visits might still be in progress, an end date for a visit is optional. Figure 10.24(a) models Visit as an objectified association in ORM, and Figure 10.24(b) takes a similar approach in UML, modeling Visit as an association class. Figure 10.24(c) models Visit as an entity type in the Barker ER notation. All of these schemas have a potential problem. You might like to identify this problem for yourself before reading on.

In Figure 10.24(a) the fact type Employee visited Country is non-functional (in this case *m:n*) and *repeatable*. For example, suppose you have visited Belgium several times and your employee number is 1001. The fact instance Employee '1001' visited Country 'BE' may be repeated, with one occurrence for each of your visits to Belgium. But the uniqueness constraint spanning the roles of the fact type Employee visited Country requires that such a fact instance appears at most once in any population of the fact type.

Figure 10.23 Modeling consecutive weights in (a) ORM, (b) UML, and (c) Barker ER.

Figure 10.24 Some attempts to model visits in (a) ORM, (b) UML, and (c) Barker ER.

In other words, the schema in Figure 10.24(a) has *no way to record multiple visits to the same country by the same employee*. The same is true of all the other variations in Figure 10.24—they assume that a visit may be identified simply by combining the visitor with the country visited. If we are not interested in recording repeated visits, then no change is needed. But suppose we do want to record a full history of visits. How do we do this? Try answering this yourself before reading on.

There are at least three ways to resolve this problem. One approach is to *include a distinguishing temporal role as part of the identifier*. For example, assuming that each employee starts to visit any given country at most once on any given date, we may identify a visit by combining the visitor, the country visited, and the start date of the visit. Figure 10.25 shows two ORM schemas using this approach. The additional external uniqueness constraint (circled bar) indicates that where a visit end date exists, each combination of visitor, country visited, and end date applies to at most one visit.

Figure 10.25 ORM schemas including start date as part of the identifier for Visit.

Figure 10.26 Figure 10.25 remodeled in (a) UML and (b) Barker ER

UML and Barker ER versions of this approach are shown in Figure 10.26. UML has no graphic for either external uniqueness constraint so these are captured in a note. The Barker ER version uses "#" to include start-date in the primary identifier, but has no way to specify the external uniqueness constraint involving end-date.

If an employee may start to visit more than one country on the same date, we need to refine the temporal granularity (e.g., to hour or minute, instead of date) to provide an appropriate visit identifier. For the rest of the discussion we assume that a temporal granularity of 1 day is sufficient.

A second approach is to *introduce a simple, visible identifier* for Visit, as shown in Figure 10.27. Of course, both of the former external uniqueness constraints still apply, but neither provides the preferred identifier.

Figure 10.27 Introducing a simple identifier in (a) ORM, (b) UML, and (c) ER.

In UML, hidden surrogate identifiers are assumed. However, we use id here as a visible identifier used by humans in the business domain to communicate about visits. This requires the addition of an id attribute to the UML diagram as shown in Figure 10.27(b). In the Barker ER solution (Figure 10.27(c)), a visit-id attribute is added, with "#" marking it as the primary identifier, and the stroke is removed from the association lines. Barker ER has no notation for alternate identifiers, so now both external uniqueness constraints are lost.

A third approach is to *introduce an ordinal number as part of the identifier*. Here the number is used to count the number of times the same employee visited the same country. For example, your first visit to Belgium is distinguished from your second visit to Belgium simply by including "first" and "second" in the definite descriptions. This visit number is included in the models in Figure 10.28. For example, the first and second visits of employee 1001 to Belgium and Norway map to the tuples ('1001', 'BE', 1), ('1001', 'BE', 2), ('1001', 'NO', 1), and ('1001', 'NO', 2), respectively.

A fourth approach in UML is to tag the association ends in Figure 10.24(b) with {nonunique}, so the population of the association is a bag of (employee, country) pairs. A surrogate id then provides a visit identifier. While allowed, this approach is subconceptual since the visit identifiers are no longer part of human communication.

While we have now covered the most common temporal data model patterns, in practice other complexities can arise that require further analysis. So far, our constraints have been *static*—they apply to each state of the information system that models the business domain and may be checked by examining each state individually (e.g., each person was born on at most one date). *Dynamic constraints* reference at least two states, which may be successive (e.g., no employee may be demoted in rank) or separated by a period (e.g., invoices ought to be paid within 30 days of issue).

Figure 10.28 Adding ordinals to help identify visits in (a) ORM, (b) UML, and (c) ER.

UML supports dynamic constraints via both diagrams (e.g. state machines and activity diagrams, as discussed in later chapters) and textual formulation of dynamic rules in OCL. Apart from the nontransferable setting in Barker ER, industrial ER provides no support for dynamic constraints. While ORM allows dynamic rules to be added as textual notes, it currently provides little formal support for dynamic constraints. Various dynamic rule extensions to both ER and ORM have been proposed, but so far these have not made their way into popular modeling tools. We now briefly sketch some ways to add dynamic rule support to ORM, based largely on a recent proposal (Balsters et al. 2006).

Constraints over *successive states* may be declared in *textual rules* using the words "**old**" and "**new**" respectively to reference the states immediately before and after an elementary transaction (involving the addition, deletion, or update of an object or fact). For example, given the fact type Employee has Salary with the second role named "salary", we might define a rule to prevent salary decreases thus:

For each Employee,
  new salary >= **old** salary.

The example just given involves a simple numeric comparison over an ordered quantity. When the relationship between successive states is more complicated, it may often be specified using a *state-transition table or a state transition diagram*. Table 10.2 shows a marital state example considered earlier in Chapter 2. Here "1" indicates a possible transition and "0" an impossible transition.

Figure 10.29 lists the possible marital state changes in a state-transition diagram. Two commonly used notations are shown. In Figure 10.29(a) each state is shown as a named, horizontal bar and allowed transitions are depicted as vertical arrows. In Figure 10.29(b), each state is shown as a named, soft rectangle.

Table 10.2 A state transition table for marital states.

From \ To	*Single*	*Married*	*Widowed*	*Divorced*
Single	0	1	0	0
Married	0	0	1	1
Widowed	0	1	0	0
Divorced	0	1	0	0

Figure 10.29 Two state transition diagram notations indicating possible transitions.

Figure 10.30 A transition constraint captured as data using a reserved predicate.

Given the ORM fact type Employee has MaritalStatus in Figure 10.30, the transition constraints may also be specified textually as a dynamic rule thus:

```
Context: Employee
in case old maritalStatus =
  'single':    new maritalStatus = 'married'
  'married':   new maritalStatus in ('widowed', 'divorced')
  'widowed':   new maritalStatus = 'married'
  'divorced':  new maritalStatus = 'married'
end cases
```

One proposal currently under consideration is to allow ORM extensions using *reserved predicates* with predefined semantics. For example, the reserved predicate "may change to" could be applied to various object types to constrain their changes from one state to the next. Using this idea, the fact table in Figure 10.30 could be used to declare the possible marital state transitions. This effectively demotes metadata (in this case a constraint) to object data, thus simplifying schema evolution, since changes to the transition rule are now data changes rather than schema changes. Notice that the salary rule considered earlier cannot be specified in this way because its formula involves an operator, not just values.

A *guarded transition diagram* allows a guard condition expressed as a formula to be applied to a transition. In practice, many different diagrams and formalisms are used to specify various kinds of dynamic constraints. These include statecharts, activity diagrams, state nets, colored Petri nets, process algebra based task structures, and *Event-Condition-Action* (*ECA*) declarations. ECA languages allow statements of the form: **If** *event* **then on** *condition* **then** *action1* [**else** *action2*]. This provides a general framework for *controlling the action to be taken on violation of a constraint*. An update is an example of an event, and a constraint violation is an example of a condition.

An extreme version of a transition constraint is to make a fact *immutable* (unchangeable). In other words, once an elementary fact is entered it cannot be deleted. This kind of constraint is best considered in connection with dynamic aspects of subtyping, which we soon discuss. For a detailed discussion of more complex examples of dynamic rules for state transitions expressed in a proposed dynamic rule language for ORM, see Balsters et al. (2006).

As mentioned before, some dynamic rules reference states that are separated by a specified period rather than being successive. In practice, many of these rules are deontic in nature. For example, consider the rule that invoices ought to be paid within 30 days of being issued. This rule may be formally expressed as follows, assuming that the relevant fact types are declared in the schema.

It is obligatory that
  each Invoice **that** was issued on $Date_1$
  is paid on **some** $Date_2$
  where $Date_2 <= Date_1 + 30$ days.

As the final segment of this section on temporal issues in modeling, we now turn to *dynamic aspects of subtyping*. This assumes that you are familiar with the static aspects of subtyping in ORM and UML discussed in Sections 6.5 and 9.6.

Recent proposals from the ontology engineering community have employed type metaproperties to ensure that subtyping schemes are well formed from an ontological perspective. Guarino and Welty (2002) argue that every property in an ontology should be labeled as rigid, nonrigid, or antirigid.

Rigid properties (e.g., being a person) necessarily apply to all their instances for their entire existence. *Nonrigid* properties (e.g., being hard) necessarily apply to some but not all their instances. *Antirigid* properties (e.g., being a patient) apply contingently to all their instances. One may then apply a meta constraint (e.g., antirigid properties cannot subsume rigid properties) to impose restrictions on subtyping (e.g., Patient cannot be a supertype of Person).

Guizzardi, Wagner, Guarino, and van Sinderen (2004) proposed a UML profile to stereotype classes into kinds, subkinds, phases, roles, categories, roleMixins, and mixins, together with a set of meta constraints, to help ensure that UML class models are ontologically well formed. This modeling profile is used by Guizzardi (2005) in his doctoral thesis on ontological foundations for conceptual information models.

While the aforementioned research provides valuable contributions, we have some reservations about its use in industrial information systems modeling. Our experience with industrial data modelers suggests that the seven-stereotype scheme would seem overly burdensome to the majority of them. To be fair, we've also had pushback on the expressive detail of ORM, to which we've replied "Well, the world you are modeling is that complex—do you want to get it right or not?" Perhaps the same response could be made in defense of the seven stereotypes.

At any rate, a simpler alternative that we are currently considering for ORM 2 *classifies each subtype as either a rigid subtype or a role subtype*. A *type is rigid* if and only if each instance of that type must belong to that type for its whole lifetime (in the business domain being modeled). Examples include Person, Animal, and Book. In contrast, an instance of a *role type* need not always be an instance of that type during its lifetime (in the business domain). Here we use "role" liberally to include a role played by an object (e.g., Manager, Student, Patient—assuming that these are changeable in the business domain) as well as a phase or state of the object (e.g., Child, Adult, FaultyProduct—assuming changeability).

Figure 10.31 Rigid subtypes depicted in (a) ORM and (b) UML.

Although this rigid/role classification scheme applies to any type, we typically require this distinction to be made only for subtypes (to control subtype migration, as discussed shortly, and to reduce the classification burden for modelers). As a simple example, Figure 10.31 depicts Dog and Cat as rigid subtypes in ORM and UML. The rigidity notation tentatively being considered for ORM is square bracketing of the subtype name (violet for alethic; blue with "°" for deontic, e.g., changing from male to female might be possible but forbidden). For UML we have chosen a rigid stereotype. Rigidity is a *dynamic constraint* rather than a static constraint since it restricts state changes (e.g., no dog may change into a cat). Our next example identifies a case where the rigidity of a root type (here Animal) should also be declared.

In the example just given, the subtypes are asserted. If instead they are derived, the relevant fact type/attribute used in their definition may be constrained by an appropriate changeability setting with an impact on subtype rigidity. In Figure 10.32(a) the fact type Animal is of AnimalKind is made unchangeable (an animal can't change its kind), as indicated by the square brackets (this notation is tentative). In Figure 10.32 (b) the defining animal kind attribute is constrained to be read only.

The unchangeability of animal kind combined with the rigidity of Animal implies that the subtypes are rigid. If we were instead to assert the subtypes and derive animal kind from subtype membership, the changeability/rigidity settings would still need to be kept in sync. UML 2 recognizes four changeability settings: unrestricted, readOnly, addOnly, and removeOnly. ORM 2 is currently being extended to enable declaration of fact type changeability (updateability and deleteability).

Figure 10.32 Rigidity of subtypes is now derived (given that Animal is rigid).

Figure 10.33 Migration between role subtypes is allowed.

To avoid explicitly declaring role subtypes as such, ORM subtypes may be assumed to be role subtypes by default. Unlike rigid subtypes, *migration between role subtypes is often permitted*. As a simple example, a person may play the role of child, teenager, and adult at different times in his/her lifetime (see Figure 10.33).

Various data model patterns cater for such *historical subtype migration*. The simplest pattern deals with linear state transitions. For example, in Figure 10.34(a) each role has specific details, and we wish to maintain these details of a person (e.g., favorite toy, favorite pop group) as he/she passes from one role to another.

An appropriate pattern for this case is to start with a supertype that disjoins all the roles and then successively subtype to smaller disjunctions, as shown in Figure 10.34(c). We call this the *decreasing disjunctions pattern*. Depending on the business domain, a simple name may be available for the top supertype (e.g., Person).

An alternative approach using what we call the *once-only role playing pattern* creates subtypes for the actual playing of the roles, as shown in Figure 10.35. Here the subtypes PersonAsChild, PersonAsTeenager, and PersonAsAdult are subtypes not of Person but of the objectified association PersonInLifeRole, instances of which correspond to the playing of one of those roles by a person. The temporal order can be added in a number of ways. This role-playing pattern can be adapted to a *repeated role playing pattern* that allows cycles in the role state graph, for example by including a start time in the identifier for the role playing object type, as in Figure 10.25.

Figure 10.34 Retaining history of subtype-specific details as a person changes roles.

Figure 10.35 An alternative approach using subtypes for role playing.

Exercise 10.3

1. (a) The following tiny extract is from a report about patient blood pressures. Blood pressure (BP) is determined from two readings, systolic pressure and diastolic pressure, each measured in millimeters of mercury (mmHg). For example a BP of 150/95 has a systolic pressure of 150 mmHg and a diastolic pressure of 95 mmHg. The information system is required to explicitly capture the semantics of BP readings. Schematize this UoD in ORM, including any derivation rules. Do *not* introduce a BP measurement object type.

PatientNr	*Patient Name*	*BP*	*Time*
101	John Smith	150/95	2007-05-01 08:30:00
101	John Smith	130/90*	2007-05-01 15:30:00
102	Ann Jones	120/80*	2007-05-01 08:30:00

* = latest BP (i.e., most recently measured BP for that patient)

   (b) Remodel this situation using the object type BPmeasurement, introducing a simple identifier for such measurements.

2. For each of the following sentences, state which parts are time deictic.
   (a) George W. Bush is our current president.
   (b) We will graduate next quarter.
   (c) Today's invoice should be paid within 30 days.
   (d) Australia just now won the World Cup in cricket.

3. The following ORM schema is designed to capture details about when people lived in various countries. A person may live in the same country many times. We know each time a person starts to live in a country but might not know when they end that habitation (e.g., they might still live there).
   (a) Explain why this design is poor.
   (b) Provide an improved ORM schema that removes the problem.

4. Each academic at an Australian university has exactly one of the following possible ranks listed in ascending order: AL = Associate Lecturer, L = Lecturer, SL = Senior Lecturer, R = Reader, P = Professor. Academics at any of the four lower ranks may be promoted to the next highest rank. Senior lecturers may also be promoted directly to Professor. No other rank changes are possible. Specify the relevant transition constraint using (a) a state chart, and (b) a transition table.

5. (a) Explain what is wrong with each of the following subtyping arrangements.

(b) The following schema was designed to record details about people who apply for an employment position at a given university, are employed at the university, or are past employees of the university. A person may be employed more than once at the university. Past employees may optionally join the past employee association—although they may do this more than once, only their most recent joining is of interest. However, the university wishes to retain a history of such details about people (e.g., employee and applicant details are retained for past employees). Modify the schema to enable this using (a) the decreasing disjunctions pattern and (b) the role-playing pattern.

10.4 Collection Types

Data modeling approaches vary considerably in their support for *collection types*. While implementation targets often support such types (e.g., SQL:2003 supports arrays, row types, and bags), high level data modeling approaches typically provide either no direct support or only very limited direct support. Even when no direct support for fact collections is provided, it is possible to model the same information in terms of elementary facts, so the issue of collection type support is one of convenience rather than expressibility. The following collection types are the most commonly encountered in conceptual modeling:

- Set (unordered, no duplicates)
- Bag or multiset (unordered, duplicates allowed)
- Ordered set or unique sequence (ordered, no duplicates)
- Sequence (ordered bag)
- Array (indexed sequence)
- Schema

Industrial ER provides no direct support for these collections. UML enables many of these collection types to be specified as stereotyped classes, multivalued attributes, or constrained association ends. Although most versions of ORM, including ORM 2, are "flat" in the sense that they provide no direct support for collections, the Predicator Set Model (PSM) variant of ORM provides deep support for them as first class types (ter Hofstede et al., 1993).

While the use of collection types in logical and physical data structures makes sense, whether collection types should be used at all in conceptual analysis is still a disputed issue. The addition of constructors for building various collection types requires additional language features to not only declare such structures, but also manipulate them. With collection options, the modeler now has many more choices as to how to formulate a model, update, or query.

Our experience indicates that collection types should be used with extreme care, if at all, in developing a conceptual model. Undisciplined use of constructors often leads to incorrect or incomplete solutions or to overlooking a simpler solution that does not use collections. Recall our earlier advice in Section 9.3 against using multivalued attributes in conceptual models.

The following points have been proposed in favor of conceptual constructor usage: collections lead to more compact models; they better reveal the connection between conceptual models and physical models that use collections; and they allow unnamed structures to be modeled directly.

The following points have been proposed against using collection types as first class types in conceptual models: collections complicate verbalization and population; collections often complicate constraints and hide other details; choosing the best constructor is often difficult; unnamed structures should be avoided because they are awkward to reference; and one can still transform to collections in the physical model simply by annotating the mapping process.

Figure 10.36 Convoy modeled as (a) an unnamed set, and (b) a named simple object.

The main point of disputation is whether it is a good idea to include *unnamed collections* in conceptual models. As a classic example (Hammer and McLeod 1981), consider a convoy as an *unnamed set* of ships. This may be modeled in PSM using a set collection type, as shown in Figure 10.36(a). While various notations exist for collection types, here we depict them as soft rectangles, with the collection kind (set, bag, etc.) indicated in bold.

Figure 10.36(b) shows an alternative flat schema in both ORM and UML using an association, and a name to identify each convoy. In practice, convoy names are very useful, making it easy to verbalize facts about convoys and to check constraints. The uniqueness constraint on the ORM model asserts that each Ship is in at most one Convoy. This constraint is captured in the UML diagram also, since the solid diamond indicates composition (strong aggregation). This constraint is missing from Figure 10.36(a). In most cases, the use of collection types makes it harder to express constraints on collection members. The association in the flat model can also be populated conveniently with fact instances to help validate the constraint.

For snapshot purposes, it is convenient to identify a convoy by its name (or by its flagship). The convoy membership fact type can be temporalized to maintain an historical record of membership either by using the ternaries Ship joined Convoy on Date and Ship left Convoy on Date or by nesting the historical *m:n* association Ship was in Convoy as ConvoyMembership, and adding ConvoyMembership began on Date; ConvoyMembership ended on Date. In UML, this history may be modeled using ConvoyMembership as an association class with appropriate date attributes (possibly multivalued) or associations. Whether it is philosophically correct to use the word "convoy" to allow a convoy to be the same convoy after a ship leaves is of little practical interest.

As a related example, consider the notion of team membership, where it is possible for a person to be a member of many teams and vice versa. Figure 10.37 shows how to model this in ORM with and without the set constructor. In Figure 10.37(b) the circled "{_}" depicts an *extensional uniqueness constraint* (ter Hofstede and van der Weide, 1994) to declare that teams may be defined by their membership (i.e., different teams cannot have exactly the same set of members). A sample population for two teams is included. The underline (like a uniqueness bar) inside the set braces intuitively suggests that the set is unique. An alternative notation for this constraint is a circled "eu". This constraint is implicit in Figure 10.37(a). Its use in Figure 10.37(b) allows unnamed structures without introducing collection types.

Figure 10.37 Three ways of modeling team membership.

Bakema, Zwart, and van der Lek (1994) provide yet another notation for this constraint, and argue that if teams are unnamed in the real world, this must be reflected in the conceptual model, to obey the conceptualization principle by modeling only conceptually relevant features. We adopt a relaxed reading of this principle, encouraging the introduction of identifiers that improve model usability. In Figure 10.37(c) we add a team code as a simple identifier for teams to facilitate talk about teams. The extensional uniqueness constraint is still required if we definitely don't want two different teams to have the same membership, as illustrated by the counterexample provided. This constraint is very expensive to enforce, so should be used only when necessary.

A complex example cited in the ORM literature to justify the use of constructors deals with chemical reactions (Falkenberg, 1993). This models reactions such as $2H_2 + O_2 \leftrightarrows 2H_2O$. One way to model this with a set collection type is shown in Figure 10.38(a). This schema, based on a solution by ter Hofstede, Proper and van der Weide (1992), treats each side of a reaction equation as a set of chemical quantities, e.g., $\{(H_2, 2), (O_2, 1)\}$. The reaction itself is then captured in the binary input/output association. An alternative flat model is shown in Figure 10.38(b). Here a reaction number is introduced as a convenient way of referring to reactions. This model also shows a complex example of the extensional uniqueness constraint.

Figure 10.38 Chemical reactions modeled (a) with and (b) without unnamed sets.

Although Figure 10.38(a) is more compact, its use of a set constructor makes it harder to think about. For example, it lacks a simple constraint captured in Figure 10.38(b). Can you spot it? The uniqueness constraints in Figure 10.38(b) ensure that each chemical occurs at most once in each reaction side (e.g., to exclude equations such as 5H ⇆ 3H + 2H). This is not enforced in Figure 10.38(a). For example, we may populate the input/output association with the tuple ⟨{(H, 5)}, {(H, 3), (H, 2)}⟩. The set-based model makes this constraint harder to conceive and express.

In Figure 10.38(a) the reaction association is *m:n* (presumably other conditions such as catalysts and temperature could be varied to have the same input reagents produce different results). If we instead demand that the same input always gives the same output, this association becomes 1:1. This is easy to model in Figure 10.38(a), but harder to model in Figure 10.38(b). This is one of those rare cases when we encounter a constraint on a collection that cannot be modeled easily as a constraint on the members. The opposite situation (a constraint on members that is hard to model on collections) is far more common.

Falkenberg (1993) models a chemical reaction instead as a set of sets of object-role pairs. Although the structure of this model is difficult to comprehend, even for modelers much less the domain experts, Falkenberg argues that it is the only correct way to model the situation, based on a deterministic principle that one birth transition instance must correspond to exactly one object instance. Our viewpoint is that communication is more important than determinism, and there are often many correct ways to model the same business domain.

While there seems to be no compelling case to introduce collection types as first class citizens in the conceptual model, we still need some way to specify collections for implementation in targets that support such structures. While this might be done directly on the logical schema, it is desirable that there be a two-way transformation between conceptual and logical levels so that changes in one can be communicated to the other to keep the models synchronized. One simple way to do this is to specify collection type mappings as *annotations* to the pure conceptual schema, resulting in an annotated conceptual schema used only by the developer, not the domain expert, for the sole purpose of controlling the mapping between conceptual and lower levels. Different annotations could be chosen for different implementation targets. At the time of writing, no standard annotations for this purpose have yet been finalized. For some further discussion on this notion, see Halpin (2000b).

We conclude this section by showing how to model a variety of collection types indirectly using a flat approach. Figure 10.39(a) uses a unique sequence constructor to model an *ordered set* of authors. Figure 10.39(b) models this in UML applying the {ordered} property to an association end ({unique} is assumed by default for this collection). This may be modeled in flat ORM by introducing a Position object type to store the sequential position of any author on the list, as shown in Figure 10.39(c). The uniqueness constraint on the first two roles declares that an author occupies at most one position for each paper; the constraint on the first and third roles indicates that for any paper, each position is occupied by at most one author. The textual constraint indicates that list positions are numbered sequentially from 1.

Figure 10.39 Some different ways to model ordered sets.

Although this ternary representation may appear awkward, it is easy to populate and it facilitates any discussion involving position (e.g., who is the second author for paper 21?). From an implementation perspective, a sequence structure could still be chosen: this can simplify updates by localizing their impact. However, the update overhead of the positional structure is not onerous, given set-at-a-time processing (e.g., to delete author n, simply set position to position -1 for position $> n$).

Care is required when modeling collection types. Figure 10.40(a) associates each fact type with an ordered set of roles, and Figure 10.40(b) flattens this into a ternary. See if you can spot something wrong with these schemas before reading on.

Both schemas omit the constraint that each role belongs to only one fact type. Figure 10.40(a) requires each role list to belong to at most one fact type, but that's not strong enough; the use of a collection type here prevents us from capturing the required uniqueness constraint graphically. The ternary in Figure 10.40(b) is not elementary (a uniqueness constraint applies to its second role). Figure 10.40(c) provides a correct solution.

Figure 10.40 Can you spot what is wrong with schemas (a) and (b)?

Figure 10.41 Modeling a bag of grades for each student in (a) ORM and (b) UML.

Recall that *bags* (multisets) are unordered collections with duplicates allowed. These may be implemented by ternary associations using a frequency object type to count the number of occurrences for each entry. For example, suppose student 1001 obtains the bag of grades [A, A, B, B, B] in five tests, and student 1002 obtains the bag of grades [B, B, B] in three tests. This may be modeled in flat ORM as shown in Figure 10.41(a). By default, multivalued association ends in UML denote sets (unique and unordered). Assuming that the nonunique setting is allowed here (the UML specification is unclear on this issue), a bag setting may be specified by the {nonunique} property as shown in Figure 10.41 (b).

Recall that a *sequence* is an ordered bag (ordered collection with duplicates allowed). For example, a paragraph is a sequence of sentences. Sequences may be modeled in flat ORM using a ternary association with a "Position" object type in the key. Suppose a lottery presents prizes to a sequence of people, in descending order of prize value, with no ties. For example, the prizes for lottery L1 might be awarded to the sequence (Ann Smith, Tom Jones, Ann Smith). This may be modeled in flat ORM using Rank as a position type as shown in Figure 10.42(a), along with a sample population. Figure 10.42(b) models this in UML using the {ordered, nonunique} setting, assuming this is allowed.

Arrays are indexed sequences. They may be modeled in flat ORM using a ternary with a "Position" type for the index as part of the key. For example, the ORM model in Figure 10.43(a) uses Place as a position type to record the first three places in sporting events, assuming no ties are allowed.

Figure 10.42 Modeling a sequence in (a) flat ORM and (b) UML.

Figure 10.43 Modeling arrays of sporting event places with no ties in flat ORM.

Using a suitable mapping annotation, this could be mapped to an indexed array of countries inside sport events, leading to an object-relational database model such as

SportEvent (eventName, results; array[1..3] of Country)
'Mens 100 m freestyle' [1: 'US'
2: 'AU'
3: 'US']

Figure 10.43(b) shows a similar example involving an additional uniqueness constraint (each country has at most one rank in any given sport). Again, no ties are allowed. This could be mapped to a unique array structure (with uniqueness indicated by underlining) as follows:

Sport (sportName, results; array[1..3] of Country)
'Swimming' [1: 'US'
2: 'AU'
3: 'CN']

Although direct use of collection types in a conceptual model can lead to compactness, an alternative flat model (no constructors) of comparable convenience is generally available. The use of constructors often makes it harder to express constraints (which typically occur on members, not collections).

Because constructors involve compound facts, they are awkward to populate or check for validation, and they require more sophisticated modeling skills. Constructors are sometimes used even when there is there is a simpler modeling solution without them. It seems better to relegate constructor usage to annotations that guide the later mapping phase.

Exercise 10.4

1. In Terry's Railway, each train consists of exactly one locomotive and exactly one carriage train (sequence of railway cars). Each carriage is classified as first class, second class, or third class. It is possible that a carriage is not currently part of a train. Trains, carriages, and locomotives are identified by numbers. The same locomotive may power many trains.

(a) The same carriage may belong to more than one train (e.g., a carriage used on a morning train may be used on a different train in the afternoon). However the same carriage train can be used in only one train. Model this in ORM using a collection type constructor.
(b) Model the domain described in (a) without using a collection type constructor.
(c) Suppose now that each train is uniquely determined by its composition (i.e., its locomotive and carriage train). Is any change is needed to your model for (a) to capture this? Explain.
(d) Can this additional constraint be captured graphically in your solution to (b)? Answer yes or no.
(e) The domain is changed so that each carriage belongs to at most one train. Can this rule be added as a graphic constraint (without adding a fact type) to your solution to (a)? Answer yes or no.
(f) Remodel your solution to (b) to ensure that the rule in (e) is expressed graphically.
(g) If the composition constraint in (c) also applies to your solution to (f), does any further constraint need to be added to that solution? Answer yes or no.

2. In cricket, a player may score runs in singles, fours, or sixes. Some players may score a duck (no runs). The following report extract lists a bag of runs (ignore the order) for each cricketer. Each cricketer belongs to exactly one team. Model this in flat ORM.

Cricketer	*Team*	*Runs*
Don Superman	A	6, 6, 6, 4, 4, 6, 4, 1, 6, 1, 6
Fred Bloggs	A	
Mark Hitter	B	1, 1, 1, 1, 4, 1, 1

10.5 Nominalization and Objectification

Roughly speaking, objectification or nesting treats a relationship instance as an object in its own right, so it may be talked about like any other object. In UML this is called reification, where instances of an association are objectified as instances of an association class. The basic notion of objectification in ORM and UML was discussed in Chapter 4 and Chapter 9, respectively. Industrial ER typically provides no support for objectification. Some academic versions of ER support objectification, but typically with restrictions (e.g., objectified relationships may have attributes but cannot play in other relationships). This section examines objectification in greater depth, first distinguishing two kinds of nominalization and then considering objectification of unaries and associations with non-spanning uniqueness constraints.

We treat *nominalization* as the recasting of a declarative sentence using a noun phrase morphologically related to a corresponding verb in the original sentence. Declarative sentences may be nominalized in different ways. One common way is to use a gerund (verbal noun) derived from the original verb or verb phrase. For example, "Elvis sang the song 'Hound Dog'" may be nominalized as "Elvis's singing of the song 'HoundDog'". Another way is to introduce a pronoun or description to refer back to the original (e.g., "that Elvis sang the song 'Hound Dog'" or "the fact that Elvis sang the song 'Hound Dog'").

In philosophy, it is usual to interpret the resulting nominalizations as naming either corresponding *states of affairs* or corresponding *propositions*. In linguistics, further alternatives are sometimes included. For example, states of affairs might be distinguished into events and situations. For information modeling purposes, we adopt the philosophical approach, ignoring finer linguistic distinctions, and classify all nominalizations into just two categories.

A *situational nominalization* refers to a state of affairs or situation in the business domain being modeled. A *propositional nominalization* refers to a proposition. We treat events (instantaneous) and activities (of short or long duration) to be special cases of a state of affairs.

The relationships among states of affairs, propositions, sentences, and communication acts have long been matters of philosophical dispute. At one extreme, states of affairs and propositions are sometimes argued to be identical. Some view logic as essentially concerned with the connection between sentences and states of affairs, while others view its focus to be propositions as abstract structures. Our viewpoint on some of these issues is motivated pragmatically by the need to model information systems, and is now summarized.

We define a proposition as that which is asserted when a sentence is uttered or inscribed. A proposition (e.g., Elvis sang Hound Dog) must be true or false (and hence is a *truth bearer*).

Intuitively it seems wrong to say that a state of affairs (e.g., Elvis's singing of Hound Dog) is true or false. Rather, a state of affairs is actual (occurs in the actual world, is the case) or not. A state of affairs may be possible or impossible. Some possible states of affairs may be actual (occur in the actual world).

States of affairs are thus *truth makers*, in that true propositions are about actual states of affairs. In sympathy with the correspondence theory of truth, we treat the relationship between propositions and states of affairs as one of *correspondence* rather than identity.

Although natural language may be ambiguous as to what a given usage of a nominalization phrase denotes (a state of affairs or a proposition), the intended meaning can usually be determined from the context in which the nominalization is used (i.e., the logical predicate applied to talk about it). For example:

Elvis sang the song 'Hound Dog'.	-- original proposition
Elvis's singing of the song 'Hound Dog' is popular.	-- actual state of affairs
That Elvis sang the song 'Hound Dog' is well known.	-- true proposition
That Elvis sang the song 'Hound Dog' is a false belief.	-- false proposition

It's snowing outside.	-- original proposition
It's true that it's snowing outside.	-- proposition
That snowing is beautiful.	-- state of affairs

The first three uses of the demonstrative pronoun "that" result in propositional nominalization. In the final example, "that" is used in combination with the gerund "snowing" to refer to a state of affairs (propositions aren't beautiful). In the previous two sentences, "snowing" is a present participle, not a gerund.

Figure 10.44 Objectification of Country plays Sport as Playing in (a) ORM and (b) UML.

As indicated earlier, states of affairs may be actual or not, and propositions may be true or false. In ordinary speech, the term "fact" is often used to mean a true proposition, but when modeling information in ORM, the term "fact" is used in the weaker sense of "proposition taken to be true" (epistemic commitment) by users of the business domain.

Figure 10.44(a) displays a typical case of objectification in information modeling in ORM. Here the fact type Country plays Sport is objectified as the object type Playing, which itself plays in another fact type Playing is at Rank. The exclamation indicates that Playing is independent, so Playing instances may exist without participating in other fact types. Figure 10.44(b) depicts the same example in UML, with the association between Country and Sport reified into the association class Playing, and the rank fact type modeled as an optional attribute.

Now consider the question: *are the objects resulting from objectification identical to the relationships that they objectify?* The UML metamodel answers yes to this question by treating AssociationClass as a subclass of both Association and Class. Since relationships are typically formalized in terms of propositions, this affirmative choice may be appropriate for propositional nominalization. However, we believe that *the objectification process used in modeling information systems is typically situational nominalization*, where the object described by the nominalization is a state of affairs rather than a proposition. For situational nominalization, we answer this question in the negative, treating fact instances and the object instances resulting from their objectification as nonidentical. An intuitive argument in support of this position follows, based on the information model in Figure 10.44.

Consider the relationship instance expressed by the sentence: "Australia plays Cricket". Clearly this relationship is a proposition, which is either true or false. Now consider the object described by the definite description: "The Playing by Australia of Cricket" or, more strictly, "The Playing by the Country that has CountryCode 'AU' of the Sport named 'Cricket'". Clearly, this Playing object is a state of affairs (e.g., an activity). It makes sense to say that Australia's playing of cricket is at rank 1, but it makes no sense to say that Australia's playing of cricket is true or false.

So the Playing instance (The Playing by Australia of Cricket) is an object that is ontologically distinct from the fact/relationship that Australia plays Cricket. Our experience is that the same may be said of any typical objectification example that one

finds in information system models. In this case, so-called *"objectified relationships" are in 1:1 correspondence with the relationships that they objectify, but they are not identical to those relationships*. Compare this with first order logic, where predicate formulae are often tested for equivalence (≡) but not identity (=). Terms or individuals may be identical, but not equivalent.

In information models, one may sometimes encounter propositional nominalizations, where the noun phrase refers to a proposition rather than a state of affairs (e.g., [the fact] that Australia plays cricket is well known; the proposition that Australia plays cricket is true). A related although different case is where the noun phrase refers to a communication act rather than a proposition (e.g., the assertion that Australia plays cricket was made by Don Bradman). We delay discussion of such cases until the end of this section.

To facilitate high level declaration of rules and queries on information models that make use of objectification, ORM includes (implicitly or explicitly) *link fact types* that link or relate the objectification result to the objects in the relationship that has been objectified. For example, the definite description "**The** Playing **that**: is by **the** Country **that** has CountryCode 'AU'; **and** is of the Sport **that** has SportName 'Cricket'" makes use of the linking fact types Playing is by Country and Playing is of Sport, as shown in Figure 10.45(a). The external uniqueness constraint indicates that each (Country, Sport) pair projected from the attached roles relates to at most one Playing object.

If the modeler does not supply readings for the linking predicates, default predicate readings are assigned, such as "involves", appended by numbers if needed to distinguish linking fact types that link back to the same object type (which plays more than one role in the fact type being objectified).

The diagram in Figure 10.45(a) is best understood as an abbreviation of the diagram in Figure 10.45(b). Here Playing is a normal object type that is related back to the Country and Sport object types using the linking fact types mentioned earlier. The equality constraint depicted by a circled "=" indicates that the (Country, Sport) pairs in the population of the Country plays Sport fact type must be identical to the population of the (Country, Sport) pairs projected from the Country and Sport roles in the join path Playing is by Country **and** is of Sport. For clarification purposes, the role sequence numbers for the equality constraint are displayed here.

Figure 10.45 Objectification in ORM uses linking fact types for relational navigation.

A similar analysis applies to the objectification of ternary and longer facts. For example, we might objectify Country plays Sport in Year as an object type Playing that has a third link fact type Playing is in Year whose year role adds a third component to the composite reference scheme for Playing.

The result of objectifying a binary or longer relationship type may now be viewed as an entity type with a composite reference scheme whose reference projection bears an equality constraint to the objectified fact type. This equality constraint may be formalized as an equivalence or 1:1 correspondence between the objectification type (e.g., Playing) and the fact type being objectified (e.g. Country plays Sport).

With respect to our example in Figure 10.45, this equivalence might be introduced to the model in one of three ways: (1) Start with the fact type Country plays Sport, and then objectify it as Playing; (2) start with the fact types Playing is by Country and Playing is of Sport and then define Country plays Sport as a fully derived fact type in terms of them; or (3) start with the three fact types Country plays Sport, Playing is by Country, and Playing is of Sport and then assert the equality constraint (corresponding to the equivalence) between them. Regardless of which way is used, the model fragment is internally stored in terms of the structure shown in Figure 10.45(b).

Although ORM treats objectification in this way internally, the link fact types and 1:1 correspondence are not displayed on the diagram. So diagrammatically objectification appears simply as in Figure 10.44(a).

Like any binary fact type, link fact types may be given predicate readings in both directions, and role names. For discussion purposes, the example in Figure 10.46 includes inverse predicate readings, role names, and a sample population. Note that the role names (acquirer, target) on the acquisition fact type provide role names for the Company roles in the link fact types.

ORM schemas may be navigated in relational style (using predicate names), attribute style (using role names), or a mixture of both. To navigate from Company to Acquisition, it matters which link fact type we use. For example, from the company Visio we may navigate via the left link to its acquisition of InfoModelers or via the right link to its acquisition by Microsoft. Navigating via the left link, the schema path may be verbalized in relational style as "Company **that** was acquirer in Acquisition". Navigating via the right link we have "Company **that** was acquired in Acquisition".

Figure 10.46 Predicate readings and role names support full navigation.

To navigate from Acquisition to company, it also matters which link we use. Navigating via the links, the schema paths may be verbalized in relational style as: Acquisition **that** was by Company (navigation via left link); Acquisition **that** is of Company (navigation via right link). Role paths may also be specified in attribute style, using role names for "attributes". For example, to navigate from Company to Acquisition, we have the following two different options: Company.acquisitionBySelf (navigation via left link); Company.acquisitionOfSelf (navigation via right link). To navigate from Acquisition to Company we have the following two different options: Acquisition.acquirer (navigation via left link); Acquisition.target (navigation via right link).

Although such expressions could be used to specify projections, here we use them simply to indicate a path obtained by jumping from an object type to one of its far roles. If the dot notation is replaced by the "of-notation" then the component order is reversed (e.g., instead of Company.acquisitionBySelf we have acquisitionBySelf **of** Company). In addition to purely relational and purely attribute styles, role paths may be specified using a mix of both styles (e.g., Company.acquisitionBySelf **that** occurred on Date). From an ORM perspective, it is all relational underneath.

UML provides no direct support for unary relationships, instead requiring their remodeling in terms of attributes or subclasses. ORM supports unary relationships and allows them to be objectified. Consider the unary fact: The President named 'Abraham Lincoln' died. We may objectify this death event using the nominalization "that death" and declare the following additional fact: That death occurred in the Country with country code 'US'. This natural way of communicating may be supported in a similar way to objectification of nonunary facts. An ORM model for this situation is shown in Figure 10.47. Small, sample populations are included for the object types and fact types. Here the unary fact type President died is objectified by the object type Death. If desired, the death entries in the fact table for Death occurred in Country may be expanded by prepending "the death of".

Internally, we interpret this case of unary objectification using the expanded schema shown in Figure 10.48. Here, Death is a normal entity type, with a simple reference scheme provided by its injective relationship to President (e.g., Abraham Lincoln's death may be referenced by the definite description "**The** Death **that** is of **the** President **who** has **the** PresidentName 'Abraham Lincoln'"). Hence, the result of objectifying a relationship type may be viewed as an entity type that has a reference scheme whose reference projection bears an equality constraint to the fact type being objectified.

Figure 10.47 Objectification of unary facts in ORM.

Figure 10.48 Objectification of unaries may be explicated as shown.

This interpretation does not assume that the Death is of President relationship provides the only or even preferred way of referring to deaths (e.g., we may introduce a death number to reference deaths, without impacting this analysis).

The aforementioned account of objectification of unaries as a case of situational nominalization was introduced in ORM 2 (objectification of unaries was not supported at all in ORM 1). The fact-oriented version known as Fully Communication Oriented Information Modeling (FCO-IM) also supports objectification of unaries, but in a very different manner (Bakema et al. 1994).

FCO-IM introduced objectification of unaries to provide an alternative way to support existential facts and to treat all base objects as lexical in nature. With this approach, an entity (nonlexical object) is an objectification of a role played by a value (lexical object). In Figure 10.49, for example, the entity type Country is derived by objectifying the unary fact type CountryCode refers to a country.

While the FCO-IM approach to objectified unaries encourages the use of natural reference schemes in modeling, and has tool support, we personally find it unintuitive, and awkward in dealing with issues such as multiple inheritance, context-dependent reference schemes, and changes to reference schemes. Nevertheless there are many modelers who favor this approach, so perhaps this is a subjective issue.

ORM 1 allowed an association to be objectified only if either it has just one uniqueness constraint, and this spans all its roles, or it is a binary 1:1 association. ORM 2 and UML allow in addition the following two kinds of associations to be objectified: an *n*:1 (or 1:*n*) binary association, and a ternary or longer association whose longest uniqueness constraint spans exactly *n* - 1 roles. We exclude from consideration any *n*-ary association whose longest uniqueness constraint spans fewer than *n*-1 roles because such an association is compound rather than elementary.

As discussed in Section 4.5 and Section 9.4, objectifying an association that has no spanning uniqueness constraint and is not 1:1 is often a case of poor modeling. The following *Objectification Guidelines* provide some heuristics to help decide whether to objectify in such cases.

Figure 10.49 In FCO-IM, nonlexical types objectify roles of lexical object types.

Objectification Guidelines:

A fact type may be objectified only if:

(a) It has only a spanning uniqueness constraint; or

(b) Its uniqueness constraint pattern is likely to evolve over time (e.g., from *n*:1 to *m:n*, or *m*:*n*:1 to *m*:*n*:*p*, etc.); or

(c) It has at least two uniqueness constraints spanning *n*-1 roles ($n > 1$), and there is no obvious choice as to which of these uniqueness constraints is the best basis for a smaller objectification; or

(d) The objectification significantly improves the display of semantic affinity between fact types attached to the objectification type.

Guideline (a) covers the common case of objectification over a spanning uniqueness constraint and has been illustrated before in both ORM and UML. For example, we might objectify the *m:n* association Student enrolled in Course as Enrollment.

As an example of guideline (b), we might objectify the *n*:1 association Employee works on Project as ProjectWork to facilitate recording the start and end dates for any employee's work on a project. If we later decide to allow an employee to work on many projects, we simply change the uniqueness constraint pattern to *m:n* and leave the attached date fact types (or date attributes in UML) unchanged.

As a simple example of guideline (c), nesting of the 1:1 CurrentMarriage association in Figure 10.50 avoids having to make an arbitrary decision at the conceptual level as to whether the marriage date should be associated with the husband or the wife. For simplicity, reference schemes for Person and Date are omitted. Currently the NORMA tool still requires one of the two uniqueness constraints to be flagged as preferred, even though the display of its preferred nature (with a double uniqueness bar) may be suppressed.

As a more complex example of guideline (c), the ternary association in Figure 10.51(a) has two overlapping uniqueness constraints, each spanning *n* - 1 (in this case two) roles. For simplicity, reference schemes are omitted. The constraint on the first two roles indicates that within each sport, each athlete has at most one current rank. The constraint on the last two roles indicates that for each sport, any given rank is assigned to at most one athlete (no ties are allowed). For each current ranking of an athlete in a given sport, we record the date on which this ranking was awarded, and optionally the achievements on which this award was based. Figure 10.51(b) depicts the same business domain in UML, where the ternary association is objectified as an association class, the award date is modeled as an attribute of this class, and achievements are modeled via an association.

Figure 10.50 Nesting of a 1:1 association in (a) ORM and (b) UML.

Figure 10.51 A case of two UCs spanning *n*-1 roles in (a) ORM and (b) UML.

Instead of objectifying the ternary, we could objectify a binary part of it with a spanning uniqueness constraint based on either Athlete is ranked in Sport or Rank is assigned in Sport. But structurally there is no formal feature that determines which of these two binary choices is preferable.

If Sport plays other roles but Rank does not, a binary objectification based on Athlete and Sport could be preferable. But one can easily imagine other cases where the decision of which binary nesting choice to make is truly arbitrary. In such cases, it seems better to avoid such arbitrary choices by objectifying the ternary as shown in Figure 10.51.

In such cases, the NORMA tool currently requires one of the uniqueness constraints to be declared as preferred, but allows the display of a double uniqueness bar to be suppressed to avoid contaminating the conceptual schema diagram with subconceptual decisions).

As an example of guideline (d), suppose we need to record for each moon the planet it orbits and the period of that orbit. Optionally, we also record one or both of the axes (minor and major) of that elliptical orbit. Since the period and axes relate directly to the orbit, it is natural to model these as facts about the orbit, as shown for ORM in Figure 10.52(a) or in UML as in Figure 10.52(b).

Instead, we could model all the facts as about the moon, as shown in Figure 10.52(c) and Figure 10.52(d). However this approach does a poorer job of displaying the semantic affinity of the period and axes details. To see this affinity you have to unpack the informal semantics of the predicate readings or attribute names rather than seeing immediately from the structure that these all relate to the orbit.

Moreover, suppose we modify the example to make it optional to record a moon's planet, but still record axes only if the moon's planet is recorded. This change is trivial in the nested versions, where we simply remove a mandatory role constraint. However, for the flattened versions we need to add two subset constraints to ensure that the axis facts are recorded for a moon only if we record which planet it orbits.

Hence, the objectified approach is preferable in such a case. Note that for relational mapping, the nested version is preprocessed to the flattened version anyway, so the same relational schema results with either approach.

Figure 10.52 Objectifying to show semantic affinity in (a) ORM and (b) UML.

The guidelines are only heuristics, and sometimes there are borderline cases. For example, suppose we record a person's birthdate only if we record his/her birth country. Figure 10.53 models this using nesting or flattening. Which do you prefer?

Figure 10.53 Do you prefer nesting (a and b) or flattening (c and d)?

The uniqueness constraint pattern is stable, so if there are no other details to record about births our personal preference is to flatten the schema, as in Figure 10.53(c) or Figure 10.53(d). This also makes it easier to modify the schema if we later remove the subset constraint. But if the subset constraint is stable, and you want to record other facts about births, then you might prefer the nested approach after all.

So far we have discussed objectification in the sense of situational nominalization, where the object being discussed is a state of affairs (event, activity, etc.). Although comparatively rare in information modeling, one may also encounter cases where the nominalized object is either a proposition (resulting from *propositional nominalization*) or a *communication act* (e.g., an utterance or inscription act by some speaker/writer).

The OMG's SBVR specification includes support for propositional nominalization. One of the SBVR rule examples involving propositional nominalization may be verbalized as follows: If a waiter earns an amount of money as a tip from serving a meal, the waiter must report that fact.

While one may interpret this as a case of propositional nominalization (reporting the fact rather than the act), there is no need to do so for information modeling purposes, as the rule may easily be declared using situational nominalization (reporting the act rather than the fact), as shown in compact form in Figure 10.54. If the rule is modified to require reporting after the service is performed, a time limit for reporting must be declared to make the rule operational; in this case, the relevant temporal object type may now be added to the model to cater for the extended rule in an obvious way. For simplicity then, we recommend modeling all propositional nominalizations instead by their corresponding situational nominalizations.

As regards modeling of communication acts, when it is of interest to model these acts, they are best modeled directly like any other business domain objects. For example, in a genealogy model we might be interested in not just descriptions of states of affairs, but assertion acts made by researchers about states of affairs. Such a model might include fact types such as: AssertionAct reported Proposition; AssertionAct was made by Researcher with ConfidenceLevel; etc. These comments relate to the information model only. For modeling communication processes, the information model should be supplemented by other kinds of model (e.g., workflow models) that provide a more intuitive and direct way of understanding essential business processes/services.

Figure 10.54 Replacing propositional nominalization by situational nominalization.

Exercise 10.5

1. With respect to the fact informally stated as "Tolkien authored *The Lord of the Rings*", classify each of the following nominalizations as situational or propositional.
   (a) I know that Tolkien authored *The Lord of the Rings*.
   (b) Tolkien's authoring of *The Lord of the Rings* took years.
   (c) It's true that Tolkien authored *The Lord of the Rings*.

2. Expand these ORM schemas to show how objectification is implemented underneath.

(a)

(b)

3. Remodel the following ORM schema to remove objectification. Include all constraints.

10.6 Open/Closed World Semantics

When modeling a business domain, one may take different positions with respect to the completeness of the knowledge captured in the model. The *Closed World Assumption* (CWA) is the assumption that all relevant facts are known (i.e., appear in the model, either as asserted facts or derived facts) and that all the relevant business rules are known. This is the usual assumption for databases and implies *Negation as Failure* (the failure to find or prove a proposition implies that it is false).

Consider, for example, the populated ORM model in Figure 10.55(a). There are four students (1001 through 1004), but only two of these (1001 and 1003) are modeled as intelligent. Using CWA, we may deduce that students 1002 and 1004 are not intelligent. Similarly we may deduce that students 1003 and 1004 do not work hard. Using the asserted facts and the derivation rule shown, we infer that three students (1001, 1002, and 1003) are expected to pass. If there are no other derivation rules for the fact type Student is expected to pass, the **if** operator is interpreted under CWA to mean **iff**. Hence student 1004 is not expected to pass.

(a)

* Student is expected to pass **if**
Student is intelligent **or** works hard.

(b)

Student (studentNr, isIntelligent, isHardWorker)

1001	true	true
1002	false	true
1003	true	false
1004	false	false

(c) *Student_View* (studentNr, isIntelligent, isHardWorker, isExpectedToPass*)

1001	true	true	true
1002	false	true	true
1003	true	false	true
1004	false	false	false

Figure 10.55 The Closed World Assumption requires all facts to be modeled.

If mapped to a relational database, the asserted positive facts and the implied negative facts are grouped into the Student table shown in Figure 10.55(b), using mandatory Boolean attributes for isIntelligent and isHardWorker. The derived facts about pass expectations can be mapped to an additional isExpectedToPass column in the Student_View shown in Figure 10.55(c). Instead of creating this view, a computed column for these derived facts could be added to the base Student table.

In contrast, the *Open World Assumption* (OWA) allows that some relevant facts may be unknown (as asserted or derivable facts). Figure 10.56 remodels our student example under OWA semantics.

(a)

* Student is expected to pass **if**
Student is intelligent **or** works hard.

(b)

Student (studentNr, isIntelligent, isHardWorker)

1001	true	true
1002	unknown	true
1003	true	unknown
1004	unknown	unknown

(c) *Student_View* (studentNr, isIntelligent, isHardWorker, isExpectedToPass*)

1001	true	true	true
1002	unknown	true	true
1003	true	unknown	true
1004	unknown	unknown	unknown

Figure 10.56 The Open World Assumption allows some facts to be unknown.

In Figure 10.56(a) the role boxes for the unary predicates have dashed lines suggesting openness and hence OWA semantics. At the time of writing, this OWA notation for ORM 2 is proposed but still tentative. The Open World Assumption implies *Classical Negation* (failure to find or prove a proposition or its negation implies that its truth value is *unknown*).

For example, as indicated in the base Student table in Figure 10.56(b), the failure to record the facts that students 1002 and 1004 are intelligent simply means that we do not know whether they are intelligent. Here we record the entry "unknown" to indicate this. In most databases, this entry would appear as a null. Check for yourself that the other "unknown" entries follow from the OWA interpretation.

As a halfway position between CWA and OWA, the *Semiclosed World Assumption* (SWA) allows that some relevant facts may be unknown as derived facts, but not as asserted facts. This implies *Semipositive Negation* (given any asserted fact type, failure to find a fact instance in it implies that instance is false; and failure to prove an instance of a derived fact type or its negation implies that its truth value is unknown). This approach can be used only when the fact types are asserted or fully derived (not semiderived).

Figure 10.57 remodels our student example under SWA semantics. The base Student table has no "unknown" entries, but the derived isExpectedToPass column in Student_View does include an "unknown" entry.

The *Open World with Negation* (OWN) assumption allows some relevant facts to be unknown (as asserted or derived facts), but also allows negations to be asserted. Like OWA, the OWN approach also leads to Classical Negation, which is the usual approach in classical logic. Although it is rare to assert negative facts in databases, it is common in logic systems. The OWN approach is just a special case of OWA.

(b)

Student (studentNr, isIntelligent, isHardWorker)

1001	true	true
1002	false	true
1003	true	false
1004	false	false

(c) *Student_View* (studentNr, isIntelligent, isHardWorker, isExpectedToPass*)

1001	true	true	true
1002	false	true	true
1003	true	false	true
1004	false	false	unknown

Figure 10.57 The Semiclosed World Assumption allows derived facts to be unknown.

Figure 10.58 Open World with Negation allows unknown and negated facts.

Figure 10.58 remodels our student example under OWN semantics. The main operator for the derivation rule has been changed to "iff" assuming there are no other rules for this fact type. As a tentative notation, negated facts in the ORM fact tables are displayed prepended by a tilde "~" (often used for negation in logic). For example, the "~1004" in the fact table for Student is intelligent asserts that student 1004 is not intelligent.

As a tentative notation for ORM 2, the unary predicates with OWN semantics are displayed with dashed role boxes including "~". Notice that some entries in the relational tables are true, some are false, and some are unknown. Check out the entries in the tables to ensure that you understand the example.

In practice, the CWA and OWN approaches are the most common for information modeling. It is possible to support these approaches without introducing the new notations discussed earlier. For example, Figure 10.59(a) shows a schema for the closed world unary fact type Country is monolingual (has exactly one official language) together with sample data. Canada (CA) and Cyprus (CY) are not recorded as being monolingual, so the CWA approach implies that they are multilingual (not monolingual).

Figure 10.59(b) shows an alternative binarized model, and Figure 10.59(c) shows yet another alternative using a second unary that corresponds to the negation of the monolingual unary, with an exclusive-or constraint. UML versions of these three ORM schemas are shown, respectively, in Figure 10.59(d), Figure 10.59(e), and Figure 10.59(f). Here the Tchar type includes just one character 'T' (for true).

Suppose now that we know that Australia (AU) and the United States (US) are monolingual and that Canada is multilingual, but don't know the language status of Cyprus. This corresponds to the OWN approach, as modeled directly in Figure 10.60(a). The rest of Figure 10.60 shows alternative ways to model this in ORM and UML.

Figure 10.59 Some CWA modeling options.

A detailed procedure for mapping unaries for the various open/closed world approaches is discussed in Chapter 11.

Figure 10.60 Some OWN modeling options.

In principle, support for the various closure options could be extended beyond unary fact types, but this would add considerable complexity, especially for non-functional fact types. Moreover the same intent can often be achieved by remodeling (e.g., using subtypes), so we ignore this possibility.

Exercise 10.6

1. With reference to the ORM model shown, complete the following table with the appropriate True, False, or UNK (unknown) entries for the two propositions about Pat, for the three cases indicated. Assume that the predicates are given the appropriate interpretation (e.g., for classical negation, the predicates are open world even if they are shown with solid lines instead of dashed lines).

Negation Approach	*Pat is male*	*Pat is a parent*
Negation as Failure		
Classical Negation		
Semipositive Negation		

2. For the following schema, a modeler insists that the unary predicates should be declared open world rather than closed world. Does this make sense? Explain.

10.7 Higher-Order Types

This section evaluates the advisability of using higher order types in information models. We first review some basic concepts from logic. *First-order logic* quantifies over individuals only, not predicates. For example, the constraint "Each Person was born on at most one Date" may be formalized in first-order typed logic thus:

$\forall x$:Person $\exists^{0..1} y$:Date x was born on y

Here the types Person and Date appear as unary predicates, "$\forall$" is the universal quantifier ("for each"), and "$\exists^{0..1}$" is the "there exists at most one" quantifier. The quantification is over individual people and individual dates. First-order predicates may be instantiated only by individuals. *Second-order* logic allows quantification over first-order predicates (as well as individuals). For example, the rule "Each asymmetric predicate is also irreflexive" may be formalized in second-order thus:

$\forall R$:Predicate (Asymmetric $R \rightarrow$ Irreflexive R)

Third-order logic allows quantification over second-order predicates, and so on. All logics whose order is above first order are called *higher-order* logics. Since types may be formalized by unary predicates, if an instance of type A is itself a type, then A is a *higher-order type*.

A *categorization type* is an object type that is used to classify another type into subtypes. For example, Gender might be used in the fact type Person is of Gender to classify Person into MalePerson and FemalePerson.

Most formalizations of ORM, ER, and the relational model are based on first-order logic. In contrast, UML introduced the notion of powertypes, whose instances may themselves be types, thus requiring higher-order semantics. There appear to be three main arguments for using higher-order types in information modeling:

1. To allow one to think of instances of some categorization types (e.g., AccountType, CarModel) as being types themselves (as for UML powertypes).
2. To directly formalize the semantics of flexible data structures where attribute entries may themselves denote sets or general concepts (e.g., object-relational tables in non-first normal form).
3. To allow one to specify business rules that seem to cross levels/metalevels (or ignore level distinctions) in the same model (e.g., Only the Finance department may specify the allowed instances of AccountType).

As the move to higher-order logic may add considerable complexity to the task of formalizing and implementing a modeling approach, it is worth investigating whether the same practical modeling objectives can be achieved while staying within a first-order framework. We believe they can.

Let's first address the question: May instances of categorization types be types themselves? For example, in the banking schema shown in Figure 10.61, is the savings account instance of AccountType identical to the subtype SavingsAccount?

Figure 10.61 Is an instance of AccountType identical to a subtype of Account?

For discussion purposes, the two AccountType instances named "CheckingAccount" and "SavingsAccount" are depicted here as a white dot and shaded dot, respectively, and the subtype has been shaded. In practice, we would normally remove or expand the value constraint on AccountType to allow other types of account (e.g., LoanAccount). Whether or not we include such a value constraint, we need to address the fundamental question, which may be phrased in ORM terms as: *Is the AccountType instance denoted by the shaded dot identical to the subtype SavingsAccount?* If we answer yes, then we have a case of an instance being a type in the same model, making AccountType second-order.

In the UML schema, the use of an enumerated type demands a no answer, because UML treats enumeration types as data types whose instances are literals. However, as discussed shortly, UML allows us to remodel the situation using a powertype for AccountType, which requires a yes answer.

In order for the AccountType instance denoted by the shaded dot to be identical to the subtype SavingsAccount, the semantics of the fact type `Account is of AccountType` should satisfy at least the following necessary conditions: (1) the "*is of*" predicate *means* "is a member of" or "is an instance of", i.e., $\in$ (*set membership*) and (2) o*nly Account instances may be instances of AccountType instances*.

These conditions (which in combination we call *homogeneous set membership*) arguably follow from the *indiscernibility of identicals* (if $a = b$, then a and b have the same properties). The subtype SavingsAccount includes precisely all the possible savings accounts in the business domain—no other things can be instances of it. If we agree that the AccountType instance denoted by the shaded dot *is* actually the subtype SavingsAccount, then only accounts can bear the instance-of relationship to it.

If we allow the type = instance interpretation, we must use higher-order logic for formalization, and should apply the semantics of homogeneous set membership to categorization relationships of this kind. If we reject the type = instance interpretation, we may stay with first-order logic (at least for formalizing categorization relationships of this kind), and may optionally distinguish such categorization relationships as special and provide them with relevant formal semantics.

To note this distinction, one could adopt a special graphical or textual adornment for such categorization associations. For example, in ORM one might append a colon ":" to the forward reading of any predicate used for this purpose (based on the common use of colons to sometimes but not always introduce types). Applying this suggestion to the model in Figure 10.61(a) would replace "is of" by "is of:".

To provide a minimal, common approach to such categorization relationships, whether or not we adopt the type = instance viewpoint, we could use the colon marker to distinguish any such relationship, and give it the semantics of an *asymmetric, intransitive*, and *locally homogeneous* relationship. A fact type of the form *A R: B* is *locally homogeneous* if and only if *B* is used as a categorization scheme for *A*, but for no other type (so no other type bears a colon relationship to *B*). For the example in Figure 10.61, this means that only Account instances may be instances of AccountType instances. It is convenient to use the same predicate reading (e.g., "is of:", or even just ":") for all such categorization predicates, unless this makes the reading awkward. The choice of reading is language dependent.

The properties of asymmetry and intransitivity seem to be the only properties of the set-membership operator ($\in$) that are relevant here. If we always use the same reading (e.g., "is of:") for the categorization relationship, we may think of it as a predefined predicate constant that applies globally (all occurrences of this predicate have the same semantics). If we reserve the colon only for such homogeneous cases, we must not use it in cases where the classification scheme (e.g., Gender) may be applied to more than one type (e.g., Person, Dog).

Note that *any* role played by a type could be used as a basis for categorizing it. So the main reason for marking such is-of associations as categorization relationships is to enforce the formal properties of such relationships. We may now formalize categorization relationships of this kind (Figure 10.62(a)), assuming the same predicate reading "is of:" for all (if this is not so, then without loss of generality, begin by replacing each reading by "is of:"). We use "~" for negation ("it is not the case that"), "&" for conjunction ("and"), and "→" for material implication ("implies"). The asymmetric and intransitive properties may be declared independent of the object types, unlike local homogeneity.

For the *first-order logic interpretation*, all types are first-order, so instances of B are individuals, not types. We use lower-case letters (possibly subscripted) to range over individuals. For our *higher-order logic interpretation*, we use capital letters (in italics) to denote type variables of any order. The order (1, 2, 3, …) of any type is implicit, since it can be derived by inspecting the full schema. If it is desired to explicitly show the order of a type, a presuperscript may be used (e.g., ^{2}B indicates that B is a second-order type).

Figure 10.62 The categorization relationship "is of:".

Post-superscripts are typically used to denote arity, and postsubscripts are often used to distinguish variables of the same type. Ignoring the case of crossing metalevels, assign the order of a type to be 1, plus the number of relationships in a contiguous chain of zero or more categorization relationships that end at the type.

Using $\exists^1$ for the "there exists exactly one" quantifier, we may now formalize the constraints on the categorization relationship in Figure 10.62(b), which has a mandatory and uniqueness constraint. The first-order formalization treats AccountType as a type of individuals.

First-order formalization:

$\forall xy$ [x is of: y $\rightarrow$ ~ (y is of: x)]	-- asymmetric
$\forall xyz$ [x is of: y & y is of: z $\rightarrow$ ~ (x is of: z)]	-- intransitive
$\forall xy$ [AccountType y & x is of: y $\rightarrow$ Account x]	-- local homogeneity
$\forall x$ [Account $x \rightarrow \exists^1 y$ (AccountType y & x is of: y)]	-- mandatory and unique

The higher-order formalization treats AccountType as a type of first-order types, so it replaces "*y*" and "z" in the aforementioned formulation by "*Y*" and "*Z*". Alternatively, the higher-order formalization may replace any expression of the form α is of: β, where β is a type variable, by βα, since it regards α is of: β in such cases to entail that α instantiates the β predicate.

One motivation for distinguishing such categorization relationships is to facilitate transformation to or from UML models that include so-called *powertypes*, which UML includes specifically to model such categorization schemes. In UML 1.4, powertypes were declared as stereotyped classes using the «powertype» annotation, but this notation was retired in UML 2. The only way in UML 2 to know that a class is a powertype is to see it referenced in a generalization constraint.

Figure 10.63 shows a simplified version of an example often used to illustrate the need for powertypes, using UML 2 notation. This way of classifying trees is botanically wrong, but that's irrelevant to the issue. Let's assume that trees can be classified into species such as Oak, Elm, and Willow in this simple way. Here TreeSpecies is a class analogous to the object type AccountType in Figure 10.61.

If the name "powertype" derives from the notion of power set (the power set of a set *A* is the set of all subsets of *A*), the term is misleading, as the powertype TreeSpecies excludes many instances in the power set of the set of trees (e.g., the null set, the set of all trees, and many other tree sets). For this reason, the term "*higher-order type*" seems more appropriate than "powertype".

Figure 10.63 Powertype example for TreeSpecies in UML 2 notation.

At any rate, this diagram does not explicitly include a membership association such as Tree is a member of TreeSpecies (analagous to Account is of AccountType). We may treat this association as implicit here, but in practice an explicit version of this association would typically be needed, since we would normally want to know the species of any given tree, and with hundreds of tree species it would be diagrammatically extravagant to introduce subtypes for all of them.

In UML 2, a colon ":" prepends the powertype name (e.g., ": TreeSpecies") to annotate the collection of displayed subtype–supertype connections that belong to the set of all possible generalization relationships (called a GeneralizationSet) based on the categorization scheme provided by the association that relates the supertype to the powertype.

If the subtypes are connected to the supertype using different arrowheads, the powertype annotation is placed next to a dashed line that crosses the relevant subtype connections (see Figure 10.63(a)). If a common arrowhead is used, the annotation is placed next to that (see Figure 10.63(b)).

We can know that a class is a powertype if its name is used in a generalization set constraint. We may not assume that any binary association from a class to a powertype is of this nature. For example, in addition to Tree is a member of TreeSpecies we might have fact types such as Person named TreeSpecies.

Figure 10.64 shows an example from the UML 2 specification that provides a different way to model a variation of our earlier account example. Here AccountType is modeled as a powertype rather than an enumeration, and the fact type Account is of AccountType is modeled as an association rather than as an attribute. It is possible to include this association in the model *without explicitly introducing any subtypes* (SavingsAccount etc.). In that case, there is *no way to formally capture the categorization semantics of the association, or to declare that AccountType is a powertype*.

While the role name "account classifier" *informally* suggests these additional semantics, this has no formal force (other examples in the UML specification use different role names such as "vehicle category", etc.). So *we need to distinguish the categorization association itself*, for example, by appending a colon to the relationship reading (e.g., "is of:") and/or the relevant association role names.

Even if generalization relationships are annotated with the relevant powertype name, this does not always formally guarantee the required semantics.

Figure 10.64 Powertype example from the UML 2.0 Superstructure specification.

Figure 10.65 Powertypes do not guarantee an unambiguous classification scheme.

For example, suppose we classify employees as part time or full time, and also record their preferred employment status (part time or full time) if any. We might model this in UML, as shown in Figure 10.65, with two associations between Employee and EmployeeType. There is no formal way of knowing which of these two associations is used as the basis for membership in the subtypes.

To solve this problem, we could require role names in the annotation or adopt the ORM practice of supplying formal subtype definitions. The types Employee, ParttimeEmployee, and FulltimeEmployee are time deictic because their sense is determined in part by the time that their terms are uttered/inscribed

UML does allow generalization annotations to include names for the generalization set (presumably prepended to :powertype names when powertypes are involved, although the UML specification does not clarify this possibility). However, such names are treated as informal comments. There is no formal requirement to link these back to properties (attributes or far roles) of the supertype, as is done in those database modeling techniques that use discriminators to indicate a basis for subtyping.

If we are to allow higher-order types in our models, what kind of higher-order logic is appropriate? The current ORM formalization uses only first-order logic, basic arithmetic, and bag comprehension. Once we adopt higher-order logic, we need to allow any order (not just second-order), because in principle one might always introduce new types to categorize existing types, whether or not those types are first-order. If we adopt standard semantics for higher-order logic, where quantifiers may range over any imaginable predicates, we lose some useful properties of first-order logic (e.g., completeness and compactness).

Moreover, care is needed to avoid some well-known paradoxes. *Russell's paradox* considers the set of all sets that are not members of themselves: is this set a member of itself? If yes, then no, and if no then yes, leading to a contradiction.

Grelling's paradox deals with self-predicable or autological properties (properties that apply to themselves). For example, we might argue that the property of being nonhuman is itself nonhuman (and hence is a self-predicable property), whereas the property of being human is not itself human (and hence is a non-self-predicable property). If predicates may instantiate themselves, then using *N* and *H* for the properties of being nonhuman and human we might formalize this as *NN* and ~*HH*, respectively.

But then what about the property of being non-self-predicable? Is this self-predicable or not? If it is, then it is not, and if it is not, then it is. Either way, we have a contradiction.

To avoid such paradoxes, Bertrand Russell developed a type theory in which types are ordered in a hierarchy, and it is meaningful to say that a type is an instance of another type only if the second type is on the next level of the hierarchy. Similarly, predicates of higher-order apply only to predicates or objects of lower orders. In particular, no predicate may apply to any predicate of the same order. Hence no predicate may apply to itself. Essentially the paradoxes are avoided by forbidding predicates to apply to themselves, by adopting a hierarchy of levels in which types can have instances at lower levels only.

While this seems a reasonable approach to adopting higher-order logic, there are other versions of higher-order logic that do not take this approach. For example, both the Knowledge Interchange Format (KIF) and the ISO Common Logic standard allow predicates to instantiate themselves, so expressions such as (*R R*) are allowed.

To make the implementation of higher-order logic more tractable, it seems best to adopt a non-standard semantics, similar to Henkin semantics, to limit the range of predicates/functions over which we may quantify, in order to retain useful properties of first-order logic such as completeness. With standard semantics, a monadic first-order predicate may range over the power set of the domain of individuals (objects: lexical or nonlexical).

To deal with categorization-types (e.g., AccountType) where we wish to assert that instances are types, it seems that the only extension we need beyond first-order logic is to allow quantification over object types that are instances of a declared categorization type (whether or not these instances have been explicitly declared as a subtype).

As a separate decision, if we wish to allow crossing metalevels in the same model (see later), we should allow quantification over object types (primitive or derived), of any order, that are explicitly declared in the schema. If we do allow this, there seems to be no compelling case to allow quantification over polyadic predicates, so this relaxation may be regarded as a restricted case of Henkin semantics.

We now outline an approach to categorization that remains first order. Consider the ORM schema in Figure 10.66, which is a classic case where higher-order logic proponents would demand that CarModel is a second-order type, whose instances are subtypes of Car (e.g., FordFutura2004).

Figure 10.66 The population of a subtype (or any type) typically varies over time.

When we think of an instance of Person and Car (concrete concepts), we think of an actual person or car. When we think of an instance of CarModel (an abstract concept), we typically think of an abstraction that is essentially a car design—a car *structure or specification* that might be denoted by a schematic diagram, for example.

For any given business domain, we define a *type* as a *set of possible instances, where for any given state of the business domain, exactly one subset of the type is the population of the type in that state*.

At any given time, the *population* of a type is the set of instances of that type that exist in the business domain at that time. The temporal aspect is needed to distinguish between types that have the same *extension over time* but may differ in *extension at some time* (e.g., HumanPerson, and HumanBaby).

By the Principle of Extensionality, sets are defined by their extension. So unless we resort to non-well-founded set theories, we must regard sets to be fixed—they cannot change over time. For any given business domain, the current population of a type (e.g., Person or Country) may change over time, but the type itself does not.

Given the aforementioned definition of type, it seems reasonable not to think of an instance of a car model as a set of possible cars of that model (a subtype). Each CarModel instance is in 1:1 correspondence with such a subtype (implicit or explicit), but it's not identical to a subtype. It seems to us that this distinction can always be made. If so, we can treat instances of such "categorization types' as ordinary individuals, that are not types, so first-order logic is enough.

Suppose we explicitly introduce a car subtype called "FordFutura2004", as shown in Figure 10.66. The subtype definition provides the formal connection between the subtype and the car model instance. Using the standard reference mode semantics for ORM, the expression "CarModel 'FordFutura2004'" in the definition abbreviates "CarModel **that** has CarModelName 'FordFutura2004'".

Suppose we add the existential fact **There exists a** CarModel **that** has CarModelName 'FordFutura2004' to the information base before any cars of that model are produced. At this time, the population of the car subtype FordFutura2004 is the null set. Suppose at a later time, 100 cars of that model are produced. At that later time, the population of the car subtype FordFutura2004 includes 100 cars.

At any time, the subtype FordFutura2004 includes those 100 cars, as well as all other cars of that model that will ever be produced. So it's perfectly okay to regard an instance of CarModel (conceived of as the fundamental car structure or design to which the car instances conform) to be in 1:1 correspondence with a set of (possible) cars.

Hence it is reasonable to think of an instance of a "categorization type" such as CarModel or AccountType as an individual (e.g., a structural pattern) that is ontologically distinct from a type (in the sense of a set of possible instances). Thus it seems sufficient to stay with first-order logic for such cases.

Although the word "Type" in "AccountType" may suggest that its instances are themselves types, this stems more from an unfortunate naming choice for the type rather than from any fundamental intuition.

The second approach that allows one to avoid higher-order types for categorization is to *avoid uninformative categorization schemes*. The term "AccountType" is uninformative, because it does not provide any basis for categorizing accounts. In principle, any object type such as Account might be categorized in many different ways, leading to different types of bank account. For example, we could define an AccountKind {Local, National, International}, an AccountCategory {Taxable, Nontaxable}, and so on. These are all categorization schemes, which we may wish to use in the same model, and names such as "AccountType" and "AccountKind" don't inform us at all about the criterion used by a given categorization scheme to place accounts into account categories.

One might argue that the value constraint placed on the categorization scheme provides this criterion, but this requires the modeler to induce the criterion based on his/her informal understanding of what the names for those values mean, an understanding that is not formally accessible to an automated system. More importantly, we may wish to introduce a categorization scheme without committing to a fixed set of instances.

As a pragmatic issue then, it seems reasonable to encourage the modeler to *choose informative names that reveal the basis for classification schemes*. If we adopt this approach, the type = instance issue typically disappears for the categorization case. In Figure 10.67, for example, the subtype Savings account is defined based on its primary function. The AccountFunction instance named 'Savings' and denoted by the shaded dot is clearly not identical to the subtype SavingsAccount (a function is not the same thing as a bank account). One may introduce other informative categorization schemes such as Account may be used in Region, etc. Similarly, our Car is of CarModel relationship might be renamed "Car conforms to CarModel".

Any binary fact type used for an enumerated categorization scheme may be replaced by one or more unary fact types. For example, instead of Account is of AccountType {Checking, Savings}, we may use the fact types Account is used primarily for savings and Account is used primarily for checking (the mandatory and uniqueness constraints are then captured by an xor constraint). Instead of Account is of AccountCategory {Taxable, Nontaxable}, we may use the fact type Account is taxable, applying the closed world assumption to determine nontaxable accounts. This is yet another way to avoid higher-order types for enumerated categorization schemes.

Figure 10.67 Informative categorization explains the basis for categorization.

Figure 10.68 A case where categorization instances are not identified with subtypes.

In many cases, *an object type may be used to categorize more than one kind of object*. For example, Figure 10.68 includes two categorization fact types: Person is of Gender and Animal is of Gender, where Gender has two possible instances identified by the gender codes 'M' (for male gender) and 'F' for female gender). Here the semantics of the categorization fact types does not involve homogeneous set membership. Clearly, instances of Gender (MaleGender, FemaleGender) are not identical to any of the four subtypes shown. This kind of categorization scheme is quite common and clearly excludes any type = instance identities.

The use of Gender in this model to define the subtypes seems better than using two powertypes PersonType {MalePerson, FemalePerson} and NonhumanAnimalType {NonhumanMaleAnimal, NonhumanFemaleAnimal}. While UML allows this case to be modeled using Gender as an attribute or non-powertype, any use of a generalization set name (e.g., 'gender') is merely an informal comment, with no formal connection to the model element used as the basis of the categorization.

We conclude this section by considering the impact of higher-order types on *set structures and metalevel crossing*. Consider a categorization scheme whose instances intuitively correspond to set-like containers. Figure 10.69 includes a model that, apart from some constraints, is structurally similar to the account example in Figure 10.61. In this business domain, only A-team members may earn privileges, so a subtype is created to record facts of this nature. The shaded dot denotes team A, and the white dot denotes team B. The shaded subtype A-TeamMember is the type whose population at any time is the set of A-team members at that time.

Figure 10.69 Is Team *A* identical to the subtype A-TeamMember?

Table 10.3 Table with entries that may be sets of individuals or attributes.

Car	*CarModel*	*ColorChoices*	*AirConditioning*	*CustomerChosen*
1	Ford Escape 2003	{red, green, black}	Yes	{ColorChoices}
2	Ford Escape 2003	{red, green, black}	No	{ }
3	Mazda MPV 2004	{green, sand-mica}	Yes	{ColorChoices, AirConditioning}

It seems natural to think of Team A at any point in time as more than just a set of people. The concept of a team brings in other semantics (a social unit whose members work together for a common purpose). While the thing denoted by the shaded dot appears to have this additional informal semantics, the subtype A-TeamMember does not—at any point in time its denotation seems to be no more than a set of people who just happen to be members of team A.

If these intuitions are correct, and team membership is considered to be a categorization scheme, here is another example where it is reasonable not to identify the subtype with the instance of the categorization scheme.

Now consider Table 10.3, which is a simplified version of an example by Fitting (2000), used to motivate a formalization of databases that uses higher-order modal logic. The table has two aspects that are unusual. First, it is in nonfirst normal form, allowing unnamed sets as entries (e.g., ColorChoices). This is permitted in some object-relational databases. Second, its final attribute (column) allows as entries unnamed sets whose instances appear to be attributes themselves, thus crossing levels/metalevels.

Fitting's formalization of this situation is higher-order, as he treats the structure directly as it stands. The price paid for this directness is deep complexity and an implementation nightmare. These disadvantages can be avoided by transformation into a first-order model that is cleaner and easier to implement.

In practice, it would be realistic to record the color chosen for a car. With this additional fact type, and omitting for now the air-conditioning aspects of the model, the situation may be modeled by the ORM schema shown in Figure 10.70. Here the color choice sets are handled in the usual normalized way, with a many:many association.

Figure 10.70 A first-order solution for part of the unnormalized table.

The subset constraint ensures that colors chosen for a car belong to those available for its car model. Facts about whether a given customer chose a given color for a car are catered for by instantiating the unary predicate applied to the objectified CarColor association. The airconditioning aspects can be catered for in a similar way.

In cases where there are many attributes about which information is to be recorded, and the attributes are not all known in advance, this may be modeled by introducing Attribute as a first-order type, along with fact types that record its name and value. By thus demoting meta-data to ground data, we remain at first order.

A final argument for using higher-order types is to allow the expression of business rules that appear to cross levels/metalevels (or ignore level distinctions) in the same model (e.g., the Finance department is responsible for defining the possible values of AccountType).

If we really want to formalize such cases directly, then higher-order types are clearly needed. However, as a pragmatic alternative, it is usually possible to handle such rules in a first-order way, either by separating the meta-aspects into a separate first-order model where the former meta-types are now ground types, or by demoting meta-data to ground data, or simply ignoring the cross-level identities.

As a simple example, we might build into the core package of the metaschema such metafact types as FactRole is played by ObjectType, BusinessRule has IllocutionaryForce etc., while in the management package of the metaschema we include metafact types dealing with aspects of security and authorization etc. (e.g., UserGroup has AccessRight to FactType). Populating the latter metafact types in the metamodel may then allow us to add rules of the desired kind without crossing metalevels.

For example, consider the simple business model in Figure 10.71. Here there are two elementary fact types (F2 and F4) and three existential fact types (F1, F3, and F5) depicted in abbreviated form using parenthesized reference modes. In this case, the current values of AccountType are stored in a reference table instead of being rigidly declared in a value-type constraint. One of our business rules is that only the marketing department may choose what the account type instances may be.

With this approach, the one column fact table for AccountType may be treated just like any other fact table in the model, including the way in which we determine who has what kind of access to what fact type. A simplified meta-fragment for dealing with security is shown in Figure 10.72. The business rule mentioned earlier is now handled by populating this metafact type as shown.

Figure 10.71 A business model with five fact types (three existential and two elementary).

Figure 10.72 A model fragment from the management package of the metamodel.

The metamodel in Figure 10.72 exists at a level above the business model in Figure 10.71. Each of these models can be formalized separately, using either first-order logic or the higher-order logic extension for categorization discussed earlier.

What is missing from this picture is the ability to identify the marketing department mentioned in the business model with the marketing department that appears as a user group in the metamodel. There does not appear to be any compelling business case to require formal support of this identity, as businesses seem to run perfectly well without such support (e.g., in SQL systems, the application tables and meta-tables are typically accessed separately).

Exercise 10.7

1. (a) Represent the ORM schema shown as a UML 2 class diagram, using PersonType as a "powertype". Introduce other modeling elements if needed.
   (b) Explain the disadvantages of your solution to (a).
   (c) Revise the ORM schema to improve its informativeness, adopting first-order semantics rather than second-order semantics.

2. The UML class diagram shown is intended to model the situation where each student is currently enrolled as an undergraduate or graduate (but not both) and plans to finally be an undergraduate or graduate (but not both). For example, some current undergraduate students might plan to go on to do a graduate degree while some other current undergraduate students might plan to never progress to a graduate degree.
   (a) Explain what is wrong with the use of StudentType as a "powertype" in this schema.
   (b) The schema also lacks a textual constraint. Remodel the situation in ORM (include a textual constraint if needed) to address both these deficiencies.

3. Explain clearly in no more than 100 words why Russell's Paradox really is a paradox.

4. Consider the following ORM schema.

(a) If the subtype is an instance of LaptopModel, is LaptopModel a higher-order type? Answer yes or no.
(b) Provide in no more than 50 words an interpretation that allows one to think of LaptopModel as a first-order type.

10.8 Summary

Join constraints apply to roles projected from an ORM path that involves at least one conceptual join. The simplest kind of join constraint applies to a single join path. External uniqueness constraints are of this kind. *Join subset constraints* are subset constraints between two role sequences, where at least one of the role sequences is projected from a join path. *Join equality constraints* and *join exclusion constraints* also include at least one role sequence argument projected from a join path.

Conceptual joins may be *inner* or *outer*. If there are multiple paths connecting roles in a role projection, role sequence numbers are needed to disambiguate the intended role path.

Alethic rules impose necessities that cannot possibly be violated, and hence any attempt to record a violation of an alethic rule in the information model will be rejected (e.g., each person was born on at most one date). *Deontic rules* specify obligations that ought to be obeyed in the business domain but may be violated (e.g., in most Western cultures, it is obligatory that each person is husband of at most one person). Violations of deontic rules may be recorded in the information system (e.g., to enable notification to relevant authorities who might be able to take some appropriate action concerning the violations).

Positive verbalization of deontic rules begin with "It is obligatory that" (instead of the implicit alethic "It is necessary that"), and negative verbalizations of deontic rules begin with "It is forbidden that" (instead of the alethic "It is impossible that"). The expression "It is forbidden that *p*" is equivalent to "It is obligatory that not *p*" and "It is not permitted that *p*".

There are three basic *temporal data types*: *instant* (point in time), *interval* (duration of time), and *period* (anchored duration of time). *Valid time* indicates when an event occurred in the business domain, whereas *transaction time* indicates when it was recorded in the information system. Time may be measured with different *temporal granularities* (e.g., second, minute, day) and with respect to different *calendars* (e.g., Gregorian, Julian) and different *time zones* (e.g., UTC, MDT).

Fact types may be *definitional* (truth of instances is a matter of definition, independent of events), *once only* (instances correspond to a single event), or *repeatable* (instances may be made true by any one of a set of events). For each once-only or repeatable fact type we need to decide what (if any) temporal information or *history* is needed. Temporal data about once-only fact types (e.g., Person was born in Country, Person wrote Book) may be maintained by adding temporal fact types of appropriate temporal granularity to the key object type (e.g., Person was born in Year, Writing (objectified) began on Date, Writing ended on Date).

Fact types may be *unchangeable*, *individually changeable*, or *collectively changeable.* Many sentences are *time deictic* (their meaning and hence truth value depends on when they were uttered or inscribed). To keep *history of changeable fact types* (e.g., Person has Weight), insert a temporal object type into the key (e.g., Person on Date had Weight) or introduce a relevant event type (e.g.,WeightMeasurement).

To keep *history of multiple events relating to a fact type with a spanning uniqueness constraint* (e.g., Person visited Country), do one of three things: include a distinguishing temporal role (e.g., Visit began on Date), introduce a simple identifier (e.g., VisitId), or introduce an ordinal number as part of the identifier (e.g., VisitNr).

Dynamic constraints apply either to transitions between successive states (e.g., changes in marital status) or over a period of time (e.g., invoices should be paid within 30 days of issue). Transition constraints can be specified textually or by state transition tables or diagrams.

Instances of a *rigid type* (e.g., Person, Car) must remain in that type for their lifetime. Instances of a *role type* (e.g., Child, Employee) may move into or out of that type at different times. No rigid type may be a subtype of a role type. Migration between subtypes (e.g., from Child to Adult) is possible. To record history of an object as it migrates between role subtypes, use either the decreasing disjunctions pattern (e.g., Figure 10.34) or the role-playing pattern (e.g., Figure 10.35).

Various implementation targets include data structures for various *collection types* (e.g., sets, bags, ordered sets, sequences, and arrays). Although such collections could be modeled directly at the conceptual level by using constructors, it seems preferable to model them indirectly, and invoke a collection type simply as a mapping annotation. It also seems more convenient to introduce names for unnamed collections.

An *extensional uniqueness constraint* {_} may be used to declare that a set type is defined by its membership. *Ordered sets* (order significant, no duplicates) such as an author list may be modeled in ORM using a position type (e.g., Book was authored by Person in Position) and in UML using the {ordered} property. *Bags* (order not significant, duplicates allowed) such as a grade list may be modeled in ORM using a frequency type (e.g., Student scored Grade with Frequency) and in UML using the {nonunique} property. Bags are also called multisets.

Sequences (ordered bags) such as an award list may be modeled in ORM using a position type in the key (e.g., Lottery awarded prize of Rank to Person) and in UML using the {ordered, nonunique} setting. *Arrays* (indexed sequences) are modeled similarly.

Nominalization recasts a verb as a noun phrase. In information modeling, objectification of a relationship is typically *situational nominalization*, where the resulting object is a state of affairs (e.g., an event or activity) in one-to-one correspondence with, but not identical to, the relationship being objectified. For example, Person sings Song; **that** Singing is delightful.

In cases of objectification, ORM creates *linking fact types* underneath (e.g., Singing is by Person, Singing is of Song). In rare cases, objectification is *propositional nominalization*, where the object being discussed is a proposition; this is how association classes are defined in UML. In information modeling, propositional nominalization can typically be replaced by situational nominalization.

ORM 2 allows unary fact types (e.g., President died) to be objectified (e.g., Death) and includes *objectification guidelines* to help decide when to objectify (e.g., don't objectify an *n*:1 association unless it might evolve to an *m:n* association, or the objectification type has multiple fact types that relate directly to it).

The *Closed World Assumption* (CWA) assumes that all relevant facts are known (appear in the model either as asserted facts or as derived facts) and implies *Negation as Failure* (failure to find or prove a proposition implies that it is false). The *Open World Assumption* (OWA) allows that some relevant facts are unknown, and implies *Classical Negation* (failure to find or prove a proposition implies that its truth value is unknown).

The *Semiclosed World Assumption* (SWA) allows that some relevant facts are unknown as derived facts but not as asserted facts, and implies *Semipositive Negation* (missing instances of asserted fact types are false, but failure to derive a fact instance or its negation implies that its truth value is unknown). The *Open World with Negation* assumption (OWN) allows that some relevant facts are unknown, but also allows negations to be asserted; as a special case of OWA, it implies Classical Negation.

Graphical notations to distinguish different closed/open world flavors for unary fact types are currently being considered for ORM 2 (e.g., dashed line for open). Even without such notations, the different approaches can be modeled, either by introducing unaries for the negated predicates or by converting to a binary. In practice, CWA and OWN are the most useful approaches, and may be captured in UML and relational schemas using mandatory Boolean attributes and optional Boolean attributes, respectively.

First-order logic quantifies over individuals only, not predicates. Higher-order logics quantify over both individuals and predicates. A *higher-order type* includes instances that are themselves types. A *categorization type* (e.g., Gender) is used to classify another type (e.g., Person) into subtypes. UML allows higher-order types called "powertypes" to be used as categorization types. For example, in UML the subtype Oak may be treated as identical to an instance of the powertype TreeSpecies which is used to classify the type Tree. Higher-order types are sometimes also used to directly model metastructures and to allow rules that cross levels/metalevels.

Higher-order logics are vastly more complex to implement efficiently than first-order logics. In practice, it is typically possible (and, we suggest, advisable) to avoid higher-order constructs in information modeling. For example, instances of categorization types can be thought of as individuals, informative fact types may be used for categorization, and different levels may be handled in different models. However, if higher-order logic is used in information modeling, we suggest that restricted Henkin semantics be used.

Chapter Notes

This chapter covered a lot of territory, and is technically the most challenging chapter in this book. Further discussion can be found in the following research papers, upon which much of the material in this chapter is based. These papers also contain detailed references to related work.

For conceptual joins, see Halpin (2002a, 2005c). For deontic rules, see Halpin (2006a, 2007a). For temporal issues in data modeling see Halpin's series of online articles on this topic in *Business Rules Journal* (*www.brcommunity.com/index.php*). For collection types, see Halpin (2000b). For nominalization and objectification, see Halpin (2005a, 2008). For higher-order types, see Halpin (2004a, 2005d).

11 Relational Mapping

11.1 Implementing a Conceptual Schema
- Overview of main steps

11.2 Relational Schemas
- Horizontal and vertical notations
- Optionality
- Key constraints
- Other rules

11.3 Relational Mapping Procedure
- Avoiding redundancy
- Mapping 1:1 associations
- Mapping external uniqueness constraints
- Mapping objectified associations
- Mapping subtypes

11.4 Advanced Mapping Aspects
- Refinements to the Rmap procedure

11.5 Summary

11.1 Implementing a Conceptual Schema

Most database modeling tools allow you to enter a data model in one or more high level notations (e.g., ER, IDEF1X, ORM, or UML), as well as a logical level notation (e.g., relational). Typically a high level (conceptual or semiconceptual) schema must be mapped down to a logical and then physical schema in order for the database to be populated and queried. Assuming you do the right thing and model first at the conceptual level, the main steps in implementing your data model are as follows.

- Design the conceptual schema
- Annotate the conceptual schema with mapping choices as needed
- Map the design to a logical schema (e.g., relational or object-relational)
- Finesse the logical schema as needed (e.g., rename or reorder some columns)
- Generate the physical schema (e.g., in Microsoft SQL Server or IBM DB2)
- Create external schema(s) (e.g., forms, reports)
- Enforce security levels as needed
- Populate the database
- Issue queries and updates
- Update the schemas as needed

Earlier chapters discussed how to design the conceptual schema. The focus of this chapter is on mapping from conceptual to logical. Because of the dominance of relational database systems, the relational model is used for the logical schema. Basic ideas about relational databases were introduced in Chapters 1 and 2. Section 11.2 summarizes these points and expands on them briefly. Section 11.3 discusses the basic procedure for mapping an ORM conceptual schema onto a relational schema. This can be adapted easily to cover mapping from other notations, such as ER or UML. Mapping from IDEF1X to relational is trivial, as discussed in Chapter 8.

Section 11.4 discusses advanced aspects of relational mapping, including the use of conceptual annotations to override default mapping choices (e.g., to control how subtypes or 1:1 associations are mapped). After automated mapping, some finessing of table and column names may be needed to meet naming standards, and columns may be reordered to improve performance. Once the logical schema is determined, if not before, the target DBMS is selected. The physical schema can then be generated for this specific platform. Some aspects of tuning the physical model to improve performance are covered later in Chapter 14 (e.g., index selection and denormalization).

Updates and queries tend to be carried out either in a logical query language (e.g., SQL or QBE) or via an external forms and reporting interface defined on top of the logical schema. Some aspects of external schema design are covered in a later chapter. With multiuser applications, different user groups' access rights to the tables, forms, and reports should be enforced by the database administrator. Chapters 12 and 13 indicate how table updates, queries, and access rights can be declared in SQL.

All the major relational DBMSs either support SQL directly or provide translation facilities to/from SQL. Some higher level query languages allow users to formulate queries directly in terms of dimensions (for data warehousing), or even conceptual schema constructs. Some of these are discussed in Chapter 16.

The next section summarizes the fundamental constructs in relational database schemas using a generic notation. The following two sections then discuss basic and advanced mapping of conceptual to relational schemas. The final section provides the chapter summary.

11.2 Relational Schemas

A *relational schema* (or relational database schema) is a set of relational table definitions, constraints, and perhaps derivation rules. You may wish to review Sections 1.3 and 2.3, where the basic ideas were discussed. The structure of a single relational table is called a *table scheme* or relation variable. This is basically a named set of *attributes* (columns) that draw their values from data *domains*. Each table scheme may be populated by a set of unnamed *tuples* (rows) of data, but the population is not part of the table scheme itself.

Many *notations* exist for table schemes. To save space, and to facilitate discussion of populations, this book normally uses a *horizontal layout*, where the table name precedes a parenthesized list of column names, separated by commas. If desired, domain names may be displayed after the column names, using a colon separator.

For example, ignoring constraints, the ORM conceptual schema in Figure 11.1(a) maps to the table scheme *Employee*(empNr: EmployeeNr, salary: Money, tax: Money). Here the table name is "Employee", the attribute empNr is based on the domain EmployeeNr, and the attributes salary and tax are both based on the domain Money. Since the unit-based reference mode USD is based on the unit dimension Money, the ORM schema could also have been displayed as in Figure 11.1(b).

We adopt the convention of starting table and domain names with a capital letter, and attribute names with a lowercase letter. This case convention is similar to that of UML and is fairly popular. However, many other case preferences are also used in practice. Whichever convention you choose, using it consistently will help make your schemas more readable.

Figure 11.1 Mapping an ORM schema to a relational schema (constraints omitted).

When using horizontal layout, we usually write the table name in italics. We also prefer to concatenate name components, with later components capitalized (e.g., empNr) rather than using a separator such as an underscore or space (e.g., emp_nr or "emp nr"). As discussed in the next chapter, the SQL standard requires double quotes around a name that includes a space character.

Modeling tools usually give you control over how such names are generated. For example, the NORMA tool allows you to declare that "Employee" is to be abbreviated to "Emp" for column names but not for table names, and can make use of role names in determining column names. You can pick your own style and stick with it. For compactness, it is usual to omit the domain names in setting out table schemes, for example, *Employee*(empNr, salary, tax).

In theory, the relational model supports semantic domains, which basically correspond to ORM conceptual object types. Prior to 1992, the SQL standard required that each attribute be defined directly over a numeric or character string data type. For example:

Person (surname: **char**(20), city: **char**(20), height: **smallint**, weight: **smallint**)

This can lead to semantic nonsense, such as comparing people and cities, or height and weight. The SQL standard now allows attributes to be defined over user-defined syntactic domains that specify an underlying data type and optionally a value list and/or default value. The underlying data types in standard SQL support character strings, numbers, date, time, and bit strings, but not Money, which is instead defined in terms of a numeric data type, for example, **decimal**(9,2). SQL:1999 introduced user-defined types, which can be used as semantic domains. However, many commercial SQLs still provide no support for user-defined domains or types, so we'll typically ignore domain details from this point on.

Let's now consider the main notations used for specifying *constraints* in a relational schema. Later sections discuss how to group fact types into table schemes and how to map constraints and rules in more detail.

Using horizontal layout, **uniqueness constraints** on relational columns are shown by <u>underlining</u>. Each unique column, or unique column combination, provides a *candidate key* for identifying rows in the table. A key is a *minimal* set of uniquely constrained attributes. In other words, if an attribute is removed from a compound key, the remaining attributes are not spanned by a uniqueness constraint. If there is only one candidate key, this is automatically the *primary key*. For example, we may indicate that empNr is the primary key of the Employee table thus:

Employee (<u>empNr</u>, salary, tax)

If more than one candidate key exists, one of these must be selected as the primary key. The others are then called "*alternate keys*" or "*secondary keys*". Primary keys are <u>doubly underlined</u> if alternate keys exist. For example, suppose we also record the name and department of each employee, and the combination of employee name and department is unique. The primary and secondary keys may be shown thus:

Employee (<u>empNr</u>, <u>empName, deptCode</u>, salary, tax)

The order in which the columns are listed is semantically irrelevant, since each column has a name unique to its table. If columns in a composite key are not listed consecutively, dotted lines connect the underlines to show that a single, composite uniqueness constraint applies rather than multiple simple constraints. For example, the previous Employee table scheme may also be displayed thus:

Employee (empNr, empName, salary, deptCode, tax)

A column that does not allow null values is said to be **mandatory** *(for its table)*. A column that does allow null values is said to be *optional*. For horizontal layout, this book adopts the following conventions: optional columns are enclosed in *square brackets*; a column is mandatory in its table unless it is marked optional. This practice is consistent with the well-known BNF (Backus-Naur Form) notation. For example, since paying tax is optional, the previous ORM schema maps to the following table scheme, with the tax column optional (null values allowed):

Employee (empNr, salary, [tax])

This mapping is illustrated with a sample population in Figure 11.2. The conceptual population includes three elementary facts: employee 101 has a salary of 80,000 US dollars, employee 102 has a salary of 30,000 USD, and employee 101 paid a tax of 30,000 USD. No tax payment is yet recorded for employee 102.

Recall that each row of a relational table expresses one or more elementary or existential facts. Since both fact types are grouped into the same table, each row expresses one or two facts. The first row records the salary and tax facts about employee 101. The second row records the salary fact about employee 102. The second row also contains a null (shown here as "?"), indicating the absence of a tax fact for employee 102. The order in which the rows are displayed is not semantically relevant.

If all roles played by an object type map to the same table, its mandatory roles can be specified simply as mandatory columns. However, the relational model often requires different facts about the same object to be stored in different tables. In general, *mandatory role constraints are captured by making their columns mandatory in their table, and running a subset constraint from other tables (if any) that contain facts about that object type*. Consider Figure 11.3(a), which adds two fact types to Figure 11.1. Each employee's gender is now recorded, as well as any cars driven by the employee.

Figure 11.2 The populated ORM model maps to the populated table scheme shown.

Figure 11.3 The subset constraints are implied by the mandatory role constraints.

As discussed in the next section, the *m:n* nature of the drives fact type requires it to be mapped to a different relational table from the other three fact types. So information about Employee is spread over two relational tables shown in Figure 11.3(c). Here, the mandatory role constraints are only partly captured—there is nothing to stop us entering employee numbers in the Drives table that do not occur in the Employee table.

How then do we map the conceptual mandatory role constraints? A role *r* is mandatory for an object type *O* only if the population of each other role played by *O* must be a subset of the population of *r*. Hence each optional role of an object type has an implied subset constraint to each mandatory role of that object type. Figure 11.3(b) shows an implied subset constraint from each optional role of Employee to a mandatory role of Employee. Equality constraints are implied between the mandatory roles, but are irrelevant to our discussion.

The three *n*:1 fact types for salary, gender, and tax are grouped into the Employee table. The mandatory salary and gender predicates map to mandatory columns, and the subset constraint on the tax role is captured by having tax as the only optional column in the Employee table. The drives fact type maps to a separate table. So to capture the subset constraint from the drives role, we add to the relational schema a subset constraint from the empNr column of the Drives table to the empNr column of the Employee table.

In horizontal layout, this subset constraint is depicted as a dotted arrow, as shown in Figure 11.3(c). This ensures that any employee listed as driving a car is also listed in the Employee table, where gender and salary are mandatorily recorded.

In relational jargon, this subset constraint is said to be a *referential integrity constraint*, and the empNr attribute of the Drives table is a *foreign key* that *references* the empNr attribute of the Employee table. In the relational model, foreign keys can reference only primary keys, but SQL also allows foreign keys to target alternate keys.

The relational model has two basic integrity rules. The *entity integrity rule* demands that primary keys contain no nulls (i.e., each column in a primary key is a mandatory column for its table). The *referential integrity rule* basically says that each nonnull value of a foreign key must match the value of some primary key. The relational model also allows for user-defined constraints and derivation rules.

To get a feel for how a relational schema might be implemented in practice, let's extend our example a little and map it to SQL. Figure 11.4 adds fact types to record the department and name of each employee. It also adds three constraints other than internal uniqueness and simple mandatory constraints. Employee names are unique within a given department, the possible gender codes are 'M' and 'F', and each employee drives at most three cars.

The external uniqueness constraint on employee name and department maps to a composite uniqueness constraint spanning two attributes. In the horizontal layout, this is denoted by underlining the relevant columns as discussed before. The value constraint on gender codes, and the frequency constraint that each employee drives at most three cars are specified in the horizontal layout by annotating the table schemes with an ORM-like notation, as shown in Figure 11.5(a).

Most ORM constraint notations may be used in a similar way in setting out relational schemas. Sometimes we adapt these notations or introduce new ones (see later sections). While our semigraphical notation is compact and fairly intuitive, commercial tools usually depict relational schemas diagrammatically using a *vertical layout*, supplemented by textual rules stored in property sheets or code. Although it takes up more space, vertical layout is convenient when a table contains many columns.

Figure 11.5(b) shows one of the ways in which Visio diagrams the same example. Here the table name is listed in the top compartment with the column names listed vertically below. Mandatory columns are shown in bold, so tax is the only optional column. Primary key columns are underlined and annotated with "PK".

Figure 11.4 A more detailed ORM schema for employees.

Figure 11.5 A relational schema in (a) horizontal layout and (b and c) some vertical layouts.

If desired, primary keys can also be displayed in a separate compartment. Foreign keys are annotated with "FK*n*" ($n > 0$), and an arrow depicts the subset constraint from the foreign key to the target table. Non-primary uniqueness constraints are denoted by "U*n*". The value constraint on gender and the frequency constraint on carRegNr are not displayed on the diagram but can be inspected in the associated code.

Figure 11.5(c) shows one way in which Microsoft SQL Server 2005 diagrams the same example. Here the primary key columns are marked with a key symbol, and the foreign key constraint is depicted as a relationship line with a key symbol at the target end and "∞" (infinity, for a cardinality of "many") at the source end. The mandatory or optional nature of a column is indicated, respectively, by the absence or presence of an Allow Nulls check mark. Other constraints may be inspected in the associated code.

As these sample vertical layouts demonstrate, there is no industry standard that is uniformly adopted for diagramming relational schemas. This is another reason for this book's use of a generic horizontal notation. When using a particular CASE tool or DBMS, you need to familiarize yourself with its specific relational notation.

The same relational schema may be specified in SQL-92 as set out later. Reserved words are shown in bold, but this is not required. To save some writing, domains on which more than one column are based are declared initially (EmpNr and Money), and constraint names are omitted. In SQL:1999, types may be declared instead of domains.

```
create domain EmpNr smallint;
create domain Money decimal(9,2);

create table Employee (
  empNr      EmpNr       not null   primary key,
  empName    varchar(20) not null,
  deptCode   varchar(5)  not null,
  gender     char        not null   check( gender in ('M','F') ),
  salary     Money       not null,
  tax        Money,
  unique ( deptCode, empName ) );

create table Drives (
  empNr EmpNr            not null   references Employee,
  carRegNr   char(6)     not null,
  primary key ( empNr, carRegNr ),
  check( not exists
        (select empNr from Drives
         group by empNr
         having count(*) > 3) );
```

The create-table statements declare the schemes for the Employee and Drives tables. A not-null clause indicates that a column is mandatory for its table. Primary keys are declared with primary-key clauses. A unique clause specifies the alternate compound key. The intertable subset constraint is declared with a references clause. Check-clauses are used to declare the value and frequency constraints. Further explanation of SQL syntax is given in the next chapter.

Although the aforementioned SQL syntax is legal as far back as SQL-92, some commercial versions of SQL do not yet support all of this syntax. For data definition, some versions have barely progressed beyond the old SQL-89 standard (which had no domain clauses, and restricted check clauses to conditions on a single row). In practice, some features of a relational schema may need to be specified as procedural code rather than declaratively.

A conceptual schema comprises three sections: fact types, constraints, and *derivation rules*. Conceptual derivation rules may be specified in an appropriate language, using either relational or attribute style notation.

For example, in the schema shown in Figure 11.6(a), an employee's net pay is derived using the ORM rule **For each** Employee, **if** tax **exists then** netpay = salary – tax **else** netpay = salary.

As shown in Figure 11.6(b), the derivation rule may be simplified to a simple subtraction if we make tax mandatory with a default of 0 (append "**not null default** = 0" to the SQL code above for the tax column) or assume that nulls are treated as zero for subtraction. The next chapter discusses in detail how SQL handles nulls.

Various options exist for mapping derivation rules to the relational level. One option is to declare a derivation rule within a *view* (virtual table), for example, **create view** NetSalary(empNr, netPay) **as select** empNr, salary – tax **from** Employee.

Figure 11.6 Making tax paid optional (a) or mandatory with a zero default (b).

A second option is to include a *generated column* in the relevant base table. In the SQL standard, this could be done by including the following definition in the table definition given earlier: netPay **generated always as** (salary – tax).

Not all DBMSs support generated columns, and some that do support this feature use a different syntax. For example, in SQL Server we may simply declare netPay **as** (salary – tax).

A third option is to include a *triggered column* in the relevant base table. As discussed in Chapter 13, triggers on a table specify actions taken automatically when relevant columns are changed in a given way. The syntax for triggers varies with the DBMS. Assuming that a netPay column is included in the Employee table, the following trigger may be used in SQL Server to derive the net pay:

```
create trigger DeriveNetPay on Employee
after insert, update as
if update(salary) or update(tax)
begin
   update Employee
   set netpay = salary – tax
end
```

A fourth option is to include the derivation rule in a *stored procedure* that is invoked when needed (see Chapter 13). The examples of SQL given earlier are intended simply for overview purposes. The next two chapters explain SQL in detail.

Once the relational schema has been declared, the initial data definition stage is over. We then perform data maintenance by adding or deleting elementary facts and perhaps some existential facts (by inserting, deleting, and modifying rows of data). Data retrieval to list base or derived facts of interest can then be carried out (e.g., by formulating appropriate SQL select statements).

Data presentation and data security tasks are also typically addressed, by creating input screens and output reports, and assigning data access rights to various user groups. These operations are discussed in detail in the next two chapters.

Exercise 11.2

1. The following table contains details about members of a martial arts club. A null value "?" indicates that a member has no phone or is unranked in any martial art. Black belt ranks are known as dan grades and lower ranks are kyu grades.

Member	*Gender*	*Phone*	*Arts and ranks*
Adams B	M	2052777	judo 3-dan; karatedo 2-kyu
Adams S	F	2052777	judo 2-kyu
Brown C	F	3579001	?
Collins T	M	?	aikido 2-dan; judo 2-dan
Dancer A	F	?	?

(a) Specify a conceptual schema for this UoD, assuming that rank (e.g., 3-dan) may be stored as a single value. Many other martial arts are possible.
(b) Explain why the table shown is not a relational table.
(c) Given that any fact type with a composite uniqueness constraint must be stored as a separate table, map your conceptual schema to a relational schema. Underline keys, mark optional fields in square brackets, and show any intertable subset constraint as a dotted arrow from the subset end to the superset end. Then draw the tables including the above data as a sample population.
(d) In a "nested relational database", the data may be stored in a single table, as shown, where Ranking is a relation nested within the Member relation. Discuss any advantages or disadvantages that might result from this approach, compared with the purely relational approach in part (c)..

Member:

memberName	*gender*	*phone*	*Ranking*	
			art	*rank*
Adams B	M	2052777	judo karatedo	3-dan 2-kyu
Adams S	F	2052777	judo	2-kyu
Brown C	F	3579001	?	?
Collins T	M	?	aikido judo	2-dan 2-dan
Dancer A	F	?	?	?

11.3 Relational Mapping Procedure

The previous section introduced a generic notation for setting out a relational schema, and discussed simple examples of mapping from a conceptual to a relational schema. We now discuss the main steps of a general procedure for performing such a mapping. Advanced aspects of this procedure are considered in Section 11.4.

For a given conceptual schema, several different relational designs might be chosen. Ideally the relational schema chosen should be *correct, efficient, and clear*. Correctness requires the relational schema to be equivalent to the conceptual schema (within

the completeness allowed by relational structures). Efficiency means good response times to updates and queries, with reasonable demands on storage space. Clarity entails that the schema should be relatively easy to understand and work with.

Since correctness of data is usually more important than fast response times, and correctness requires adequate constraint enforcement, a high priority is normally placed on simplifying the enforcement of constraints at update time. The main way to do this is to *avoid redundancy*. This strategy can lead to more tables in the design, which can slow down queries and updates if extra table joins are now required. For efficiency, we should try to keep the number of tables down to an acceptable limit.

With these criteria in mind, the **Rmap** (*R*elational *map*ping) procedure guarantees a redundancy-free relational design and includes strategies to restrict the number of tables. Rmap extends and refines an older mapping procedure known as the "Optimal Normal Form" (ONF) algorithm. The full version of Rmap includes details for completely mapping all graphical conceptual constraints and is beyond the scope of this book. However, the central steps of this procedure are covered in this section and the next.

As discussed in Chapter 14, more efficient relational designs with fewer tables may possibly result if the conceptual schema is transformed by an optimization algorithm before Rmap is applied, and sometimes lower level optimization using controlled redundancy may be needed to meet critical performance requirements.

Even without such further optimization, the Rmap procedure is extremely valuable since it guarantees a safe, reasonably efficient, normalized design. Happily, the basic steps of the procedure are simple. *Redundancy is repetition of an atomic fact*.

An atomic fact is typically elementary (e.g., The Scientist named 'Einstein' was born in the Country named 'Germany') but in rare cases could be existential (e.g. There exists a Country named 'Germany').

Having gone to the trouble of ensuring that our conceptual fact types are atomic, we can easily avoid redundancy in our relational tables.

Typically each row of a relational table stores one or more elementary facts. Otherwise the table row stores a single reference or existential fact (e.g., a lookup table of country names). Hence we can automatically *avoid redundancy* in tables by ensuring that *each fact type maps to only one table in such a way that its instances appear only once*.

To achieve this, there are two basic rules, as follows. As an exercise, convince yourself that grouping facts like this makes it impossible for any fact to be duplicated.

1. Each fact type with a compound, internal uniqueness constraint maps to a separate table by itself, keyed on the uniqueness constraint.
2. Fact types with functional roles attached to the same object type are grouped into the same table, keyed on the object type's identifier.

These two rules show how to group fact types into table schemes. The first rule is illustrated in Figure 11.7. Any nonobjectified predicate with a uniqueness constraint spanning two or more of its roles must map to a table all by itself. Hence *m:n* binaries, and all *n*-aries ($n \geq 3$) on the conceptual schema, map to a separate table (unless they are objectified associations, as discussed later).

Figure 11.7 Each fact type with a compound, internal UC maps to a table by itself.

Each object type maps to one or more attributes depending on the number of components in its reference scheme. If the predicate has only one uniqueness constraint, the table's primary key is the attribute (or attribute set) spanned by this constraint. Otherwise one of the uniqueness constraints spans the primary key, and the others span alternate keys. ORM tools allow you to specify which constraint you prefer for the primary key before mapping.

When mapping conceptual schemas, care should be taken to *choose meaningful table and column names*. In the case of Figure 11.7, the table is used to store instances of the conceptual relationship type, so is often given a name similar to the conceptual predicate reading. If the object types involved are different, their names or the names of their value types, or a concatenation of the object and value type names, are often used as column names. For instance: *Drives*(empNr, CarRegNr).

When information about the same object is spread over more than one table, it is usually better to consistently use the same column name for this object, unless this loses an important distinction between the semantics of the different roles involved (e.g., empNr was also used in the Employee table to refer to employees). Apart from helping the designer see the connection, this practice can simplify the formulation of natural joins in the SQL standard. However, if different roles of the same predicate are played by the same object type, different column names must be chosen to reflect the different roles involved. For example: *Contains*(superpart, subpart, quantity).

A typical example of the second grouping rule is illustrated in Figure 11.8. Recall that a functional role has a simple uniqueness constraint. Here two functional roles are attached to the object type *A*. The rule also applies when there is just one functional role, or more than two. The handling of mandatory and optional roles was considered earlier (optional column in square brackets). The identification schemes of *A*, *B*, and *C* are not shown here, but may be simple or composite. The fact types are grouped together into a single table, with the identifier of *A* as the primary key (shown here as *a*).

Figure 11.8 Functional fact types of the same object type are grouped together.

Figure 11.9 A simple relational mapping.

The name of *A* is often chosen as the table name. To avoid confusion, it is best never to use the same name for both a table and a column, so the name of *A*'s value type is often chosen for *a*. The names of *B* and *C* (or their value types or role names) are often chosen as the other column names (here *b* and *c*). For example: *Employee* (empNr, gender, [phone], salary, [tax]).

Once table groupings have been determined, keys underlined, optional columns marked in square brackets, and other constraints (e.g., subset, value list) mapped down, any derivation rules are also mapped, as discussed in the previous section.

To help understand the Rmap procedure it will help to consider several examples. In our initial examples, all entity types have simple identifiers, and no subtypes or objectified associations occur.

Consider the conceptual schema in Figure 11.9. The many:many fact type maps to a separate table (using rule 1), while the functional fact types map to a table keyed on the identifier of Employee (using rule 2). The primary keys are underlined, and the optionality of mobileNr is shown by the square brackets. The dotted arrow depicts the *subset constraint* that each employee referenced in the Drives table is also referenced in the primary key of the Employee table (referential integrity).

Now consider the ORM schema in Figure 11.10. Notice the *equality constraint*, shown as an arrow-tipped dotted line between the empNr fields of both tables. This is needed since both roles played by Academic are mandatory, and each maps to a different table. An equality constraint is equivalent to two subset constraints, going in opposite directions. This causes a *referential cycle*, since each table refers to the other.

The subset constraint from Qualification.empNr to Academic.empNr is a simple foreign key constraint, since it references a primary key. But the subset constraint from Academic.empNr to Qualification.empNr is not a foreign key constraint, since it targets only part of a primary key.

Figure 11.10 The equality constraint entails a referential cycle.

Figure 11.11 An ORM schema for horse races and its relational map.

This latter subset constraint may be enforced in various ways (e.g., by assertions, triggers, or stored procedures). Referential cycles typically result from a role that has a mandatory constraint but no simple uniqueness constraint. Since referential cycles require special care to implement, this conceptual constraint pattern should be avoided if possible. In our example, however, the business rule that each academic holds at least one degree does seem reasonable, so in this case we leave it as is.

Before leaving this example, a few more comments about naming are worth making. The table name "Qualification" is one of many possible choices to convey the notion of academics being qualified by their degrees. Two other choices are to use the association name ("Holds") or to concatenate the object type names ("AcademicDegree"). The department column is named "deptCode" because we expect other departmental facts (not shown here) to be recorded in a department table, and we want to avoid using "department" to name both a column and a table. Similarly, the degree column is named "degreeCode" rather than "degree", since we might want to record other facts about degrees (e.g., their titles). In general, if the object type underlying a column will also underlie the primary key of another table, it is advisable to include the reference scheme in the column name.

Now consider Figure 11.11. Here each horse has its gender and weight recorded. Each race has at most one winner (we do not allow ties). There are no composite keys for rule 1 to work on. The gender and weight fact types have functional roles attached to Horse. By rule 2, these two associations must be grouped into the same table keyed on the identifier for Horse. Similarly, the time and win fact types are functions of race and are grouped into the Race table. We chose the column name "raceTime" instead of "time" mainly because "time" is a reserved word in SQL. It's best to avoid using reserved words for names, since they require double quotes. We later provide links to online lists of SQL reserved words. The column name "winner" is used instead of "horseName" to convey the semantics better.

Mapping 1:1 Associations

Let's now consider some examples involving *1:1 fact types*. In the conceptual schema of Figure 11.12, both Bankcard and Client share a 1:1 association. If we are certain that no other functional roles will be played by Bankcard, it is usually best to group the 1:1 fact type together with the other functional fact types into the same table, as shown.

Figure 11.12 The mapping choice if Bankcard plays no other functional roles.

The bankcard column is optional, but the disadvantage of nulls here is usually outweighed by the advantage of having all the data in one table. The underlining of bankcard indicates a uniqueness constraint for its nonnull values. Because it may contain more than one null, the bankcard column cannot be used to identify rows, and hence cannot be the primary key. With two uniqueness constraints, we doubly underline clientNr to highlight it as the primary key.

If Bankcard does play other functional roles, however, the bankcard ownership association should be grouped into a Bankcard table, as shown in Figure 11.13. Notice that the owned role played by Bankcard is now explicitly mandatory; in the previous example it was only implicitly mandatory (because it was the only role played by Bankcard). Notice that *only one role of this 1:1 association is mandatory*. In asymmetric cases like this, it is usually better to *group on the mandatory role side* as shown.

To illustrate this idea further, consider the conceptual schema in Figure 11.14. Here each employee is identified primarily by his/her employee number, but also has a unique name (unusual, but some departments work that way by refining formerly duplicate names). Each department has one head, who heads only one department. Since not all employees are department heads, the role of heading is optional. So the 1:1 association between Employee and Department is optional for Employee but mandatory for Department. Since it is usually better to group on the mandatory role side, here this means including the headEmpNr column in the Department table, as shown.

Figure 11.13 The mapping choice if Bankcard plays another functional role.

Figure 11.14 The heads fact type is grouped on the mandatory role side.

Suppose we grouped on the optional side instead, by adding a deptHeaded column to the Employee table. This leads to the following relational schema:

Employee (empNr, [deptHeaded], empName, salary)

Department (deptCode, budget)

This alternative has two disadvantages. First, the optional deptHeaded column permits nulls (unlike the mandatory headEmpNr column). All other things being equal, nulls should be avoided if possible. In addition to consuming storage, they are often awkward for people to work with. The second disadvantage is that we now have an equality constraint between the tables, rather than just a subset constraint.

In principle, we might group into one table all the functional predicates of both object types involved in a 1:1 association. With our example this gives the scheme: *EmployeeDept*(empNr, empName, salary, [deptHeaded, deptBudget]). Here enclosing the last two columns in the same square brackets declares that one is null if and only if the other is.

However, apart from requiring two optional columns and a special constraint that requires them to be null together, this grouping is unnatural. For example, the primary key suggests that the whole table deals with employees rather than their departments. For such reasons, this single-table approach should normally be avoided.

Now consider Figure 11.15(a). Here each employee has the use of exactly one company car, and each company car is allocated to exactly one employee. Employees and cars are identified by their employee number and by their registration number, respectively. The names of employees and the car models (e.g., Mazda MPV) are also recorded, but these need not be unique. Here both roles of the 1:1 association are mandatory, and each is attached to an entity type with another functional role. Should the 1:1 fact type be grouped into an Employee table or into a Car table?

Unlike the previous example, we now have a symmetrical situation with respect to mandatory roles. An arbitrary decision could be made here. We could group to the left, as in Figure 11.15(b) or to the right as in Figure 11.15(c). Either of these approaches is reasonable.

Figure 11.15 The ORM schema (a) may map to relational schema (b) or to schema (c).

We might also try a single table: *EmployeeCar*(empNr, empName, carRegNr, carModel). Although possible, it is unnatural, requiring an arbitrary choice of primary key. This becomes more awkward if other facts about employees (e.g., gender) and cars (e.g., purchase date) are recorded in the table. Also, consider the additional update overhead to change the car used by an employee, compared with the two table approach.

Now what about the case of a 1:1 fact type with both roles optional? For example, consider a UoD identical to that just discussed except that only some employees are given company cars and only some company cars are used by employees (e.g., some may be reserved for important visitors). The two table approach is recommended. Because of the symmetry with respect to mandatory roles, we could map the 1:1 association into the Employee table to give the schema: *Employee*(empNr, empName, [carRegNr]); *Car*(carRegNr, carModel); Employee.carRegNr references Car. Alternatively we could map it to the Car table yielding the schema: *Employee*(empNr, empName); *Car*(carRegNr, carModel, [empNr]); Car.empNr references Employee. The *percentage of nulls* is likely to differ in these two designs. In this case, the design with fewer nulls is usually preferable.

Yet another option is a three table approach, in which the 1:1 association has a table to itself. This option becomes more attractive if the two table approach yields high percentages of nulls. For example, if only 1% of employees and 1% of cars are likely to be involved in Employee drives Car, we might map this fact type into a table by itself, giving three tables overall. A fourth option for 1:1 cases is to use two tables, but include the 1:1 association in both, with a special equality constraint to control the redundancy. More detailed discussions of mapping 1:1 fact types are referenced in the chapter notes. Our default procedure is summarized in Figure 11.16; here "functional role" means a functional role in an elementary fact type (not a reference type).

In Figure 11.16, the arrow "⇦" indicates that the 1:1 fact type should be grouped into the functional table of the left-hand object type. In case (a), the right-hand object type may play nonfunctional roles not shown here, and any role in the 1:1 fact type may be optional or mandatory. The third case (no other functional roles) is rare and requires a choice of primary key in the separate table. The final line refers to symmetric cases where the roles of the 1:1 predicate are both mandatory or both optional, and both object types play another functional role—here we have a grouping choice.

Each *1:1 fact type* maps to only one table:

if (a) only one object type in the 1:1 predicate has another functional role
then group on its side
else if (b) both object types have other functional roles
and only one role in the 1:1 is explicitly mandatory
then group on its side
else if no object type has another functional role
then map the 1:1 to a separate table
else the grouping choice is up to you

Figure 11.16 Default procedure for mapping 1:1 fact types.

To understand some further cases, it will help to recall the *fundamental bridge between conceptual and logical levels*. This is summarized in Figure 11.17. In the populated ORM schema in Figure 11.17(a), the mandatory role pattern anticipates other roles being added later to Employee but not Gender. Figure 11.17(a) abbreviates the preferred reference schemes shown explicitly in Figure 11.17(b), which uses icons to directly depict a female employee and the female gender. The shaded, derived association abbreviates the conceptual path from EmployeeNr through Employee and Gender to GenderCode. This unpacks the semantics underlying the relational schema in Figure 11.17(c).

Figure 11.17 The conceptual/logical bridge.

Recall that uniqueness constraints on reference predicates are the responsibility of humans to enforce. The information system can't stop us from giving the same employee number to two different employees, or giving the same employee two employee numbers. Hence *uniqueness constraints on primary reference schemes are not mapped*. Instead, we must enforce them in the real world.

Assuming that we have enforced the primary reference constraints however, the system can enforce constraints on the fact types. For example, it enforces the uniqueness constraint in *Employee*(empNr, gender) by ensuring that each employee number occurs only once in that column and hence is paired with at most one gender code. Assuming that the reference types really are 1:1, this uniqueness constraint on empNr corresponds to the uniqueness constraint on the conceptual fact type (i.e., **Each** Employee is of **at most one** Gender). It does not capture any uniqueness constraint from the reference types.

Mapping External Uniqueness Constraints

Let's now consider mapping of schemas that include *external uniqueness constraints*. Recall that such constraints appear as a circled bar, and if they are used for preferred reference they have a double bar. In the ORM schema of Figure 11.18, employees are identified by combining their family name and initials. Since the external uniqueness constraint in this example underlies the preferred reference scheme, it is not mapped. In the resulting table scheme *Employee*(familyName, initials, gender), the uniqueness constraint corresponds to the uniqueness constraint on the conceptual fact type Employee is of Gender. A good way to visualize the mapping is as follows:

- Mentally erase the identification scheme of each object type.
- Group facts into tables, using simple surrogates for the roles of real world objects.
- Replace each surrogate by the attribute(s) used in the table to identify the objects.

For example, the conceptual schemas of Figure 11.17 and Figure 11.18 each map initially to *Employee*(*e*, *g*) with the meaning Employee *e* is of Gender *g*. In both cases, *g* is then replaced by "gender". With Figure 11.17, *e* is replaced by empNr, but with Figure 11.18, *e* is unpacked into familyName, initials since the identification scheme is composite. Since the uniqueness constraint spans *e*, it must also span the attribute combination that replaces it.

Figure 11.18 Composite primary identifier, and functional fact type.

⇨ *Drives* (e, c) ⇨ *Drives* (familyName, initials, carRegNr)

Figure 11.19 Composite primary identifier and nonfunctional fact type.

Now consider the conceptual schema of Figure 11.19. The structural difference here is that the fact type has a composite uniqueness constraint. We may initially think of it mapping to the table *Drives*(e, c), where *e* and *c* are surrogates for the employee and car roles. Replacing the surrogates by the real identifiers results in the table scheme *Drives*(familyName, initials, carRegNr). Since *e* was just part of a longer key, so is the composite identifier for employee that replaces it.

Now consider the conceptual schema of Figure 11.20. This is like that of Figure 11.18 except that Employee now has empNr as its primary identifier. The external uniqueness constraint now applies to two fact types rather than reference types, and hence can be mapped and enforced by the information system. As shown, this maps to a uniqueness constraint spanning surname and initials in the relational table (ensuring each familyName, initials combination is paired with only one employee number).

The uniqueness constraint on the table's primary key, empNr, captures the three simple uniqueness constraints on the three conceptual fact types (i.e., since each empNr is unique, it is paired with only one family name, only one sequence of initials, and only one gender).

Now consider the conceptual schema of Figure 11.21. Here each laboratory session is identified primarily by a session number and is used for a particular course. Once these course bookings have been made, sessions are assigned for use by students. The external uniqueness constraint says that each student is assigned at most one session for each course (e.g., laboratory resources might be scarce). Since this constraint involves fact types rather than reference types, it can be mapped.

An unusual feature of this example is the application of the external uniqueness constraint to a set of roles that includes a role in an *m:n* fact type (LabSession is assigned to Student).

⇨ *Employee* (empNr, familyName, initials, gender)

Figure 11.20 Composite secondary identifier and functional fact type.

Figure 11.21 External uniqueness constraint involving an *m:n* fact type.

Because the *m:n* fact type must map to a table by itself, the external uniqueness constraint spans two relational tables. This is equivalent to an internal uniqueness constraint spanning courseCode and studentNr in the natural join of the two tables.

Mapping Objectified Associations

Let's now discuss some cases involving *objectification* (also known as *nesting*). In Figure 11.22 the association Employee worked on Project is objectified as Work. This objectified type plays one mandatory and one optional role, both of which are functional. As with other object types, we *initially treat the objectified association as a "black box", mentally erasing its identification scheme*.

From this viewpoint, the conceptual schema appears to have just two fact types, both having functional roles attached to the object type Work. *Fact types are now grouped in the usual way*. So these two fact types are grouped into the same table. Visualizing the nested object type Work as a black box "■" results in the table: *Work*(■, startdate, [enddate]). Finally we *unpack* ■ *into its component attributes* (empNr and projectName), giving *Work*(empNr, projectName, startdate, [enddate]).

The value constraint in Figure 11.22 verbalizes as: **For each** Work, **existing** enddate >= startdate. This may be alternatively specified as a textual constraint. As discussed in Section 7.4, value constraints are violated if and only if they evaluate to false (just like SQL check clauses). The footnote annotation on the relational schema here corresponds to the SQL check clause **check** (enddate >= startdate).

Figure 11.22 An active, objectified association with functional roles.

Figure 11.23 Mapping an independent, objectified association with functional roles.

Independent object types, whether nested or not, require special treatment in mapping, as follows.

- Map each independent object type and the fact types (if any) in which it plays functional roles to a separate table, with the object type's identifier as the primary key, and all other attributes optional. Add a subset constraint to this primary key from each column sequence mapped from nonfunctional roles (if any) of the independent object type.

Consider the ORM schema of Figure 11.23. Here employees might be assigned to projects before their actual starting date is known. So some instances of the objectified Assignment association might be recorded that (in some database state) do not play either of the attached roles. In this example, since even the disjunction of the attached roles is optional, the nested object type is *independent*—this is noted by the exclamation mark in "Assignment !". Contrast this with the previous example (Figure 11.22), where the nested object type Work is active.

If we wish to record just the fact that an employee is assigned to a particular project, we enter values just for empNr and projectName. Details about start and end dates for work on the project can be added later when they are known. Since both start date and end date are optional, the value constraint now verbalizes as: **For each** Assignment, **existing** enddate >= **existing** startdate.

Figure 11.24 shows another way to model this UoD. Here the independent object type has no functional roles attached. In this case, it maps to a table all by itself. The *m:n* fact type maps to another table. A subset constraint captures its optionality. If instead we tried to map everything to one table, this would violate entity integrity (why?).

The constraint that if an assignment's start and end dates are known, the end date must come after or on the start date is now harder to formulate, both at the conceptual level and at the relational level. Moreover, the approach of Figure 11.24 spreads the information over more tables and demands an intertable constraint. This is usually undesirable, since it often slows down the execution of queries and updates. For such reasons, the approach of Figure 11.23 is normally preferred to that of Figure 11.24.

Figure 11.24 Mapping an independent, objectified association with no functional roles.

Notice that the schemas in Figure 11.23 and Figure 11.24 allow us to record an end date without a start date. In practical applications, we sometimes allow things like this because our information may be incomplete. For example, we might want to record the date somebody ended a project, but not *know* when that person started. For the same reason, conceptual schemas sometimes have fewer mandatory roles than the ideal world suggests, and in consequence relational schemas may have more optional columns than complete knowledge would allow.

Suppose, however, that if we know an end date then we *do* know the starting date. To enforce this constraint in Figure 11.23, add a subset constraint on the ORM schema from the first role of the "ended on" predicate to the first role of the "started on" predicate. This constraint may be declared in the relational schema by using *nested option brackets*. Here we enclose the option brackets for enddate *inside* the option brackets for startdate, giving: *Assignment*(empNr, projectName, [startdate, [enddate]]). This indicates that a (nonnull) value is recorded for enddate only if a value is recorded for startdate.

To add this constraint to Figure 11.24 is not as easy. A textual constraint is required at both conceptual and relational levels to declare that for each employee, project pair, phase "end" is recorded only if phase "start" is.

Chapter 14 examines in detail the notion of "equivalent" conceptual schemas and provides guidelines for transforming a conceptual schema to improve the efficiency of the relational schema obtained from Rmap.

If the start and end dates for an employee's work on a project are recorded, a derivation rule can be specified to compute the work period (by subtracting the start date from the end date, if known). Since each computer system has an internal clock, conceptually there is a unary fact type of the form: Date is today. So for someone still working on a project we could also derive the time spent so far on the project by subtracting the start date from the current value for "today".

By default, derived facts are not stored. So *by default, derived columns are excluded from the base tables* (i.e., stored tables) of the relational schema. For example, the derivation rules could be implemented using views or stored procedures.

In some cases, efficiency considerations may lead us to derive on update rather than at query time, and store the derived information. In such cases the derived fact type is marked "**" on the conceptual schema diagram. During the relational mapping, the fact type is mapped to a base table, and the derivation rule is mapped to a rule that is triggered by updates to the base table(s) referenced in the derivation rule. For example, we could implement the derivation rule by declaring appropriate generated columns or table triggers (recall the earlier net pay example). As well as derivation rules, *all* conceptual constraints should be mapped (not just uniqueness and mandatory constraints).

Mapping Subtypes

Let's now consider the *mapping of subtype constraints*. Table 11.1 was met earlier in Chapter 6. Although we have no rule to determine when a phone number is recorded, we know that prostate status may be recorded only for male patients, and pregnancies are recorded for all female patients and only for female patients. The conceptual schema is reproduced in Figure 11.25.

Table 11.1

PatientNr	*Name*	*Gender*	*Phone*	*Prostate status*	*Pregnancies*
101	Adams A	M	2052061	OK	–
102	Blossom F	F	3652999	–	5
103	Jones E	F	?	–	0
104	King P	M	?	benign enlargement	–
105	Smith J	M	2057654	?	–

⇨ *Patient* (patientNr, patientName, gender {M, F}, [phone], [prostateStatus][1], [nrPregnancies][2] {0..})

[1] **exists only if** gender = 'M'
[2] **exists iff** gender = 'F'

Figure 11.25 Subtype constraints on functional roles map to qualified optionals.

Some limited support for subtyping is included in the SQL standard, but current relational systems typically do not support this concept directly. Nevertheless, there are three main ways in which subtyping can be implemented on current systems: absorption, separation, and partition. With *absorption*, we *absorb the subtypes back into the (top) supertype* (giving *qualified optional roles*), *group the fact types as usual*, and then *add the subtyping constraints as textual qualifications,* as shown in Figure 11.25.

To best understand this example, visualize the subtypes on the conceptual schema being absorbed back into the supertype Patient, with the subtype roles now attached as qualified optional roles to Patient. All the roles attached to Patient are functional, so the fact types all map to the same table, as shown. The phone, prostateStatus, and nrPregnancies attributes are all optional, but the latter two are qualified.

In the qualifications, "**exists**" means the value is not null. Qualification 1 is a pure subtyping constraint, indicating that a nonnull value for prostate status is recorded on a row **only if** the value of gender on that row is 'M'. Some men might have a null recorded here. Qualification 2 expresses both a subtype constraint (number of pregnancies is recorded **only if** gender is 'F') and a mandatory role constraint (nrPregnancies is recorded **if** gender is 'F'). Recall that "**iff**" is short for "if and only if".

Since only functional fact types are involved in this example, the absorption approach leads to a table that basically matches that of the original output report (except that most relational systems support only one kind of null). The main advantage of this approach is that it maps all the functional predicates of a subtype family into a single table. This usually makes related queries and updates more efficient. Its main disadvantage is that it generates nulls.

The second main approach, *separation*, creates separate tables for facts with subtype specific functional roles. With this approach, the conceptual schema of Figure 11.25 maps to three tables: one for common facts, one for male-specific facts, and one for female-specific facts.

The third main approach, *partition*, may be used when the subtypes form a partition of their supertype. Since MalePatient and FemalePatient are exclusive and exhaustive, this approach may be used here, resulting in two tables: one containing all the facts about the male patients and the other all the facts about female patients.

The next section discusses the relative merits of these three approaches and also considers mapping cases where subtypes use a preferred identification scheme different from the supertype's. Our default approach, however, is to absorb the subtypes before grouping. Note that even with this approach, *any fact types with a nonfunctional role played by a subtype map to separate tables, with their subtype definitions expressed by qualified subset constraints* targeting the main supertype table.

For example, the ORM schema in Figure 11.26 adds the optional *m:n* fact type to FemalePatient: FemalePatient attended prenatal clinic on Date. This fact type maps to a separate table with the qualified subset constraint "**only where** gender = 'F'", as shown. If the new fact type were instead mandatory for FemalePatient, the qualification would read "**exactly where"** instead of "**only where**".

We have now covered all the basic steps in the Rmap procedure. These are now summarized. Even if you are using a CASE tool to do the mapping for you, it's nice to understand how the mapping works.

⇨ *Patient* (patientNr, patientName, gender, [phone], [prostateStatus][1], [nrPregnancies][2])
{M, F} {0..}
3
PreNatalVisit (patientNr, attendanceDate)

[1] **exists only if** gender = 'M'
[2] **exists iff** gender = 'F'
[3] **only where** gender = 'F'

Figure 11.26 Nonfunctional fact type of subtypes map to separate tables.

Basic Rmap procedure:

0 Absorb subtypes into their top supertype.
Mentally erase all explicit preferred identification schemes,
treating compositely identified object types as "black boxes".

1 Map each fact type with a compound UC to a separate table.

2 Fact types with functional roles attached to the same object type
are grouped into the same table, keyed on the object type's identifier.
Map 1:1 cases to a single table, generally favoring fewer nulls.

3 Map each independent object type with no functional roles to a separate table.

4 Unpack each "black box column" into its component attributes.

5 Map all other constraints and derivation rules.
Subtype constraints on functional roles map to qualified optional columns,
and those on nonfunctional roles map to qualified subset constraints.
Nonfunctional roles of independent object types map to column sequences
that reference the independent table.

Step 0 may be thought of as a preparatory mental exercise. Erasing all explicit preferred reference schemes (i.e., those shown other than by parenthesized reference modes) ensures that all the remaining predicates on display (as box sequences) belong to fact types (rather than reference types). Recall that a *compositely identified object type* is either a nested object type (objectified association) or a coreferenced object type (identified via an external uniqueness constraint).

Figure 11.27 The conceptual schema from the compact disc case study.

There are plenty of questions in the section exercise to give you practice at performing the mapping manually. As preparation for this exercise, some larger examples are now considered. Figure 11.27 shows the conceptual schema for the compact disc case study discussed in Chapter 5. If you feel confident, you might like to try mapping this to a relational schema yourself before reading on.

As there are no subtypes in this example, step 0 amounts to mentally erasing any preferred reference schemes that are shown explicitly. There are only two, each involving a compositely identified object type: Track and Listing. Figure 11.28 depicts this erasure by removing the predicate boxes and showing their connections to object types as dashed lines. For steps 1–3 we treat these two, compositely identified entity types just like any other entity type.

We now proceed to group fact types into tables. To help visualize this we *place a lasso around each group of predicates that map to the same table* (see Figure 11.28). We lasso only the predicates, not the object types. Since each fact type should map to exactly one table, *all predicates must be lassoed*, and *no lassos may overlap.*

In step 1 we look around for a predicate with a compound uniqueness constraint. Since the objectified association is now hidden, we see only one such predicate: `is sung by`. So we lasso this predicate, indicating it goes to a table all by itself.

In step 2 we group functional fact types of the same object type together. For example, CompactDisc has five functional roles attached to it, so these five fact types are grouped into a table keyed on the identifier for CompactDisc. Similarly the two functional fact types for Track are lassoed together, as are the two functional fact types of the nested object type.

Figure 11.28 The reference types are erased, and fact types are lassoed into groups.

This example has no 1:1 cases, and no independent object types. We have now roped all the predicates, and there are four lassos, so the conceptual schema maps to four tables.

The final relational schema is shown in Figure 11.29. Since five functional fact types map to the CD (CompactDisc) table, and all the objects involved have simple identifiers, this table has six columns (one for the key and one for each fact attribute). The other three tables involve a compositely identified object type, which is unpacked into its component attributes (step 4).

Sales (cdNr, monthCode, qtySold, revenue)

CD (cdNr, cdName, [artist], company, stockQty, rrp)

Track (cdNr, trackNr, title, duration)

Vocals (cdNr, trackNr, singer)

```
* SalesTotals (monthCode, totalQtySold, totalRevenue) :: =
        monthCode, sum(qtySold), sum(revenue)
        from Sales
        group by monthCode
```

Figure 11.29 Relational schema mapped from the conceptual schema in Figure 11.27.

The keys of the tables are already determined, but the mandatory role constraints are enforced by mandatory columns and inter-table subset/equality constraints. Note the composite subset constraint from the pair cdNr, trackNr in Vocals to the primary key of Track. Finally the two derivation rules are mapped (step 5).

If you lasso fact types into groups before writing down the table schemes, you may mentally erase all preferred identification schemes (including reference modes) in step 0 (thus treating all object types as black boxes) and then in step 4 replace each column by its identifying attribute(s). This alternative formulation of the Rmap procedure is logically cleaner. If performing Rmap manually, you might find it convenient to photocopy the conceptual schema and use colored pencils to cross out the reference types and lasso the fact types.

As a complicated example, consider the ORM schema in Figure 11.30. This concerns television channels. Notice that TimeSlot is modeled as a coreferenced entity type (a time slot is identified as a given hour on a given day). For variety, the object types Office and Department are modeled as objectified associations. A television channel may have different offices in different suburbs. For this UoD, a channel may have only one office in any given suburb. A given office may have many departments.

Figure 11.30 An ORM schema for TV channels.

For example, one department might be the advertising department for the channel 9 office located in the suburb Toowong. Alternatively (and arguably more naturally), the object types Office and Department could have been modeled as coreferenced object types. However this would not change the mapping.

The external uniqueness constraint on Program indicates that a channel can screen only one program at a given time. As an exercise, try to map this yourself before reading on. Start by mentally erasing the reference types and lassoing the fact types that should be grouped together before you write down the relational schema.

Figure 11.31 hides the identifying predicates for Time, Office and Department, and lassos the fact types into groups. The 1:1 associations must be grouped with Channel and Program since these have other functional roles, but Frequency and Title do not.

The detailed relational schema is shown in Figure 11.32. The identifier for Office unpacks into two attributes, while the Department key unpacks into three (two for Office and one for DepartmentKind). Notice also the intertable uniqueness constraint. This indicates that when a natural join is performed between the Program and ProgTime tables, there will be a uniqueness constraint spanning the three columns channelNr, progDay, and progHour. In other words, for any given channel and time, there is at most one program being shown.

Figure 11.31 Reference types are erased, and fact types are grouped.

Figure 11.32 A relational schema mapped from the model in Figure 11.30.

Notice the many equality constraints in the relational schema. These lead to referential cycles that may be awkward to implement. The many mandatory roles also entail that a lot of data must be entered in a single transaction. As discussed earlier, to minimize referential cycles, you should not add a mandatory role constraint to a nonfunctional role unless it is absolutely required.

As an optional exercise, you may wish to consider which mandatory role constraints would best be removed from Figure 11.30.

The following exercise contains many questions to give you practice with relational mapping. The choice of names for tables and columns is up to you, but you may wish to consider the naming guidelines discussed earlier.

Mapping from a conceptual to a relational schema is a bit like doing CSDP step 1 in reverse. Try to choose names for tables and columns that would make it easier for you to perform CSDP step 1 if presented with the relational tables. Remember that tables are basically just collections of facts.

Exercise 11.3

1. Map the following conceptual schema onto a relational schema using the Rmap procedure. Use descriptive table and column names. Underline the keys, and enclose any optional columns in square brackets. Indicate any subset or equality constraints.

2. Rmap the following conceptual schema.

3. (a) The conceptual schema for a given UoD is shown. A novice designer maps the likes and costs fact types into a single table: *Likes* (woman, dress, cost). With the aid of a small, sample population, explain why this table is badly designed. Use surrogates w1, w2, … for women and d1, d2, … for dresses.
   (b) Rmap the conceptual schema.

4. In a given UoD, each lecturer has at least one degree and optionally has taught at one or more institutions. Each lecturer has exactly one identifying name, and at most one nickname. Each degree is standardly identified by its code, but also has a unique title. Some degrees might not be held by any lecturer. Each institution is identified by its name. The gender and birth year of the lecturers are recorded, as well as the years in which their degrees were awarded.
   (a) A novice designer develops a relational schema for this UoD that has the following two tables. Explain, with the aid of sample data, why these tables are badly designed.

   Lecturer (lecturerName, gender, degreeCode, degreeTitle)
   Qualification (degreeCode, yearAwarded)

   (b) Draw a conceptual schema for this UoD.
   (c) Rmap your answer to (b).

5. Add constraints to your conceptual schema for the following and then perform Rmap.
   (a) Exercise 3.5, Question 2.
   (b) Exercise 3.5, Question 3.

6. Rmap the Invoice conceptual schema for Exercise 6.3, Question 3.

7. Rmap the Oz Bank conceptual schema for Exercise 6.4, Question 6.

8. Consider a naval UoD in which sailors are identified by a sailorNr, and ships by a shipNr, although both have names as well (not necessarily unique). We must record the sex, rank and birthdate of each sailor and the weight (in tonnes) and construction date of each ship. Each ship may have only one captain and vice versa. Specify a conceptual schema and relational schema for this UoD for the following cases.
   (a) Each captain commands a ship but some ships might not have captains.
   (b) Each ship has a captain but some captains might not command a ship.
   (c) Each captain commands a ship, and each ship has a captain.
   (d) Some captains might not command a ship, and some ships might not have captains.

9. Consider a UoD in which students enroll in subjects, and later obtain scores on one or more tests for each subject. Schematize this conceptually using nesting and then Rmap it.
   (a) Assume that scores are available for all enrollments.
   (b) Assume that enrollments are recorded before any scores become known.

10. Rmap the conceptual schema in Figure 6.46 (absorb the subtypes before mapping).

11. Refer to the MediaSurvey conceptual schema in Figure 6.53.
    (a) Rmap this schema (absorb subtypes).
    (b) Set out an alternative relational schema with separate tables for each node in the subtype graph. Which schema is preferable?

12. (a) Rmap the Taxpayer conceptual schema for Exercise 6.5, Question 3(a).
    (b) Rmap the Taxpayer conceptual schema for Exercise 6.5, Question 3(b).

13. Rmap the SolarSystem conceptual schema for Exercise 6.5, Question 6.

14. Rmap the conceptual schema in Figure 7.5 (note that Panel is an independent object type).

15. Rmap the CountryBorders conceptual schema for Exercise 7.3, Question 5.

16. Rmap the IT Company schema shown.

17. Rmap the University schema shown. Note that Degree is compositely identified (e.g., a Ph.D. from UCLA and a Ph.D. from MIT are treated as different degrees). Student is also compositely identified (by the time you finish the mapping, you will appreciate how much better it would be to use a studentNr instead!). The codes ‘y’, ‘n’, ‘PT’, ‘FT’, ‘int’, and ‘ext’ abbreviate “yes”, “no”, “part time”, “full time”, “internal”, and “external”.

IT Company Schema

University Schema

18. Rmap the following conceptual schema. Ignore the implicit semantic connection between Year and Date.

19. A life insurance company maintains an information system about its clients. The following information samples are extracted from a fragment of this system. For simplicity, several data items (e.g., client's name and address) are omitted and may be ignored for this question. Each client is identified by his/her client number (clientNr).

The following table uses these abbreviations: Emp. = employment (EA = Employed by Another; SE = Self-Employed; NE = Not Employed); Acct = accounting; NS = Non-Smoker; S = Smoker. Some clients are referred by other clients (referrers) to take out a policy (this entitles referrers to special benefits outside our UoD). The value "?" is an ordinary null value meaning "not recorded" (e.g., non-existent or unknown). The value "–" means "inapplicable" (because of some other data).

ClientNr	*Referrer*	*Birth date*	*Smoking status*	*Emp. status*	*Job*	*Work phone*	*Acct. firm*
101	?	15/2/46	NS	EA	lecturer	3650001	–
102	?	15/2/46	S	SE	builder	9821234	Acme
103	101	1/1/70	NS	NE	–	–	–
104	103	3/4/65	S	EA	painter	?	–
105	?	1/1/70	NS	SE	painter	2692900	?

For each client a record of major illnesses (if any) is kept; one recovery figure and hospital is noted for each client illness. The extract for clients 101–105 is shown.

ClientNr	*Illness*	*Degree (%) of recovery*	*Hospital treated*
102	stroke	90	Wandin Valley
	diabetes	80	Burrigan
103	diabetes	95	Wandin Valley

Each client selects one insurance coverage (currently $25,000, $50,000, or $100,000) and pays a monthly premium for this coverage. Premiums are determined by coverage, age, and smoking status, as shown in the following schedule. This schedule is stored in the database (it is not computed by a formula). From time to time, this schedule may change in premiums charged, coverages offered, or even age groups. Premiums for both smokers and non-smokers are always included for each age group/coverage combination. For simplicity, only the latest version of the schedule is stored (a history of previous schedules is not kept). Moreover, payments must be for 1 year at a time and are calculated as 12 times the relevant premium current at the date of payment (using the age of the client at that date).

The computer system has an internal clock, which may be viewed conceptually as providing an always up-to-date instance of the fact type: Date is today. You may use "**today**" as an initialized date variable in derivation rules.

	Nonsmoker premiums ($)			*Smoker premiums ($)*		
Age *(y)*	25,000 coverage	50,000 coverage	100,000 coverage	25,000 coverage	50,000 coverage	100,000 coverage
21–39	5.00	7.50	12.50	6.50	10.50	18.50
40–49	6.50	10.50	18.50	10.00	17.50	32.50
50–59	13.00	23.50	34.00	24.00	45.50	88.50
60–69	36.50	70.50	138.50	57.50	112.50	222.50

Although not shown in the schedule, age groups are identified primarily by an age group number (currently in the range 1..4). Assume that all clients have paid their 12-month fee by completing a form like that shown here (details such as name and address are omitted for this exercise). The first four fields are completed by the insurance agency and the rest by the client. Records of any previous payments are outside the UoD.

ClientNr: ..... Date: .....

Insurance coverage: $ .....

Payment for next 12 months' insurance: $ .....

Method of payment (Please check one): ❑ cash ❑ check ❑ credit card

If paying by credit card, complete the following:

❑ Visa ❑ Mastercard ❑ Bankcard

Card Number: ☐☐☐☐ ☐☐☐☐ ☐☐☐☐ ☐☐☐☐

Card expiration date: ☐☐/☐☐

Choose suitable codes to abbreviate payment methods and card types. Each credit card is used by at most two clients (e.g., husband and wife). In practice the card type could be derived from the starting digits of the card number, but ignore this possibility for this question.

(a) Specify a conceptual schema for this UoD. Include **all** graphic constraints and any noteworthy textual constraints. Include subtype definitions and derivation rules. If a derived fact type should be stored, include it on the diagram with an "**" mark to indicate it is both derived and stored.

(b) Should the payment by a client be derived only, stored only, or both? Discuss the practical issues involved in making this decision.

(c) Map your conceptual schema to a relational schema, absorbing subtypes while maintaining subtype constraints. Underline keys and mark optional columns with square brackets. Include **all** constraints. As an optional exercise, map any derivation rules.

11.4 Advanced Mapping Aspects

The previous section discussed the main steps in the default relational mapping procedure (Rmap). Step 0 of this procedure may be refined as shown here. This section discusses these refinements in order and clarifies some fine points. If desired, this section may be safely skipped on a first reading.

Rmap Step 0:

0.1 Transform exclusive unaries; map unaries according to their open/closed world semantics.
0.2 Mentally erase all reference (preferred identification) predicates.
Treat compositely identified object types as "black boxes".
0.3 Indicate any absorption-overrides (separation or partition) for subtypes.
0.4 Identify any derived fact types that must be stored.
0.5 Indicate mapping choices for symmetric 1:1 cases.
0.6 Consider replacing any disjunctive reference schemes
by using an artificial or concatenated identifier or mandatory defaults.
0.7 Indicate mapping choice where required for any objectified associations
that have no spanning uniqueness constraint.

As background to step 0.1, please review the discussion of open/closed world semantics in Section 10.6. Two unaries with an exclusive-or constraint (e.g., Person is male, Person is female) should be replaced by a single closed world unary (e.g., Person is male) or a mandatory binary (e.g., Person is of Gender {'M', 'F'}).

After that, mutually exclusive unaries should be replaced by a binary. For example, in Figure 11.33(a) the disjunction of the unaries is optional, so it is possible that some applications are pending a decision. In Figure 11.33(b) the unaries are transformed to a single binary, where the status of the application is identified by a code (S = Succeeded, F = Failed, P = Pending). This transformation is understood to be performed automatically as a preprocessing stage to the rest of Rmap. For analysis purposes, you may still work with the original conceptual schema unless you prefer the transformed version. The final result of the mapping is shown at the bottom of Figure 11.33(b).

Figure 11.33 Replacing exclusive unaries by a binary.

Remaining unaries are mapped according to their open/closed world assumption. Figure 11.34 shows a simple example, with cases (a), (b), and (c) corresponding to closed world, simple open world, and open world with negation. Typically, Person would play other mandatory roles and then would not be independent. In practice, CWA leading to a mandatory Boolean and OWN leading to an optional Boolean are the most useful cases.

Figure 11.34 Basic mapping patterns for unobjectified unaries.

Figure 11.35 Basic mapping patterns for objectified unaries.

Figure 11.35 shows how to map an objectified unary, with cases (a), (b), and (c) corresponding to closed world, simple open world, and open world with negation. This mapping pre-processes the functional fact type attached to Smoking by attaching it directly to Person, renaming the role name "startdate" to "smokingStartdate, and then mapping as usual, resulting in a single table rather than two. Simple variations apply if the attached role is mandatory (e.g., in the CWA case replace "**only if**" by "**iff**").

Step 0.2 involves mentally erasing any reference predicates and treating objectified associations as simple object types. The previous section discussed typical cases. Figure 11.36 depicts a special case. Here each Olympic Games is identified by the year in which it is held. There are three fact types (Athlete competes in Games; Athlete was born in Year; City hosts Games) and one reference type (Games is held in Year).

The reference type must be shown explicitly, since Year is an entity type, not a value type. One instance of the hosts fact type is: The City with name 'Sydney' hosts the Games that is held in the Year 2000 CE.

Figure 11.36 Games is identified by a single, explicit reference type.

It is rare to have an object type identified by means of a single, explicit reference type. To map this case, first mentally erase the reference type, leaving the three fact types to be grouped into tables. As an exercise, try the mapping yourself and then check your answer with the solution shown in Figure 11.36.

By default, subtypes are absorbed into their root supertype for mapping purposes. Although the subtype constraints are still mapped (as qualifications on optional columns or intertable subset constraints), absorption generally causes any functional roles of subtypes to be grouped together with the functional roles of the root supertype. In step 0.3 we may wish to override this default. Let's consider some examples, starting with the usual case where the subtypes inherit the preferred reference scheme of their top supertype.

Figure 11.37 depicts a simple subtype graph with functional roles. Reference schemes and subtype definitions are omitted for simplicity. As discussed in the previous section, there are three basic ways in which the functional fact types associated with the nodes may be grouped into tables: *absorption, separation,* and *partition*.

The first option *absorbs* the subtypes back into the supertype before grouping. For example, assume that all the roles played by *A*, *B,* and *C* are functional (i.e., they have a simple uniqueness constraint). In this case we generate just one table (with subtype constraints expressed as qualifications on the optional subtype attributes). This absorption default has two main advantages: queries that require attributes from more than one node (no joins required) perform better, and subtype constraints are usually easy to specify and inexpensive to enforce (no joins).

The main disadvantages of subtype absorption are as follows: nulls are required for objects that belong to only one subtype; the supertype's functional table is larger (more columns); queries about only one subtype require a restriction; and viewing just a subtype is less convenient (projection and restriction needed). Usually the advantages outweigh the disadvantages. Note that any nonfunctional roles of the subtypes map to separate tables, so these are not affected by our subtype mapping choice.

The second option, *separation*, groups the functional roles attached directly to each object type node into separate tables, one for each node. Here the functional predicates of *A* map to one table (the common properties), the attributes specific to *B* map to another table, and the attributes specific to *C* map to a third table. The main advantages of separation are that it minimizes nulls, and queries about each subtype are fast. Its main weaknesses are the following: queries requiring attributes from more than one node are slower (joins needed) and insertions to subtype tables are slower (subtype constraints are now specified as qualified subset constraints, so access to a supertable is required to enforce them).

Figure 11.37 How do we map functional fact types played by subtypes?

The third option is to horizontally *partition* the instances of A. This should normally be considered only if B and C form a partition of A (i.e., they are exclusive and exhaustive: $B \cap C = \{\ \}$; $B \cup C = A$). In this case one table holds all the functional predicates of B (including those attached to A), and another holds all the functional predicates of C. However, if B and C do not exhaust A, a separate table is needed for $A - (B \cup C)$.

If B and C overlap, redundancy results for the facts about $B \cap C$, and this must be controlled. This partition option departs from the usual practice of grouping each fact type into only one table and tends to be useful only in distributed or federated database settings.

The main advantages of partitioning the supertype are as follows: it minimizes nulls; queries about all the properties of B (or C) are fast; and subtype constraints typically need not be coded (because implied) or are trivial to code (e.g., Man and Woman tables without/with gender field). Its main disadvantages are the following: it slows queries about all of A, or B and C (joins needed); it is very awkward unless B and C form a partition of A; and if B and C are exclusive, enforcement of this constraint normally requires inter-table access.

The criteria discussed can help us decide whether to override the default absorption option. Override decisions may be selected from an option list in a CASE tool. For larger subtype graphs, the mapping choices increase rapidly, as mixtures of the three options might be used.

To clarify some of the previous discussion, let's look at some examples. The ORM schema in Figure 11.38 recalls a patient example discussed in earlier sections. The three basic mapping choices are shown at the bottom of Figure 11.38. With absorption, all the fact types are grouped into a single table, and the subtyping constraints appear as qualifications on optional columns (the **only if** condition captures the subtyping, and the **iff** condition captures both subtyping and mandatory role constraints).

With separation, the common fact types are grouped into the Patient supertable, and the subtype-specific fact types are grouped into separate subtype tables, giving three tables overall. The subtyping constraints appear as qualifications on the subset constraints (in this case, foreign key references). In the subset constraint qualifications, the **only where** clause captures just a subtyping constraint, while the **exactly where** clause covers both subtyping and mandatory role constraints.

With partition, the common fact types are pushed down into subtype tables that also contain the specific fact types for those subtypes. An exclusion constraint between the keys of the tables ensures that each patient maps to only one of these tables. A view definition is included to list all the patients by unioning the keys of the two tables. The gender information is now informally captured in the table names. If desired, a view could be created to form a single table with all the Patient details, deriving gender from the table names.

While the three choices just discussed are typically the only ones considered, it is possible to mix and match the options. For example, we could choose to absorb the prostate fact type for male patients but keep a separate table for pregnancy count for female patients.

Figure 11.38 Absorption, Separation, and Partition mapping options.

If we were to add the *m:n* prenatal fact type considered earlier in Figure 11.26, this would map to its own separate table, regardless of which of the three subtype mapping options is chosen. Recall that the absorption choice absorbs only the fact types functional for the subtype.

Now consider the conceptual schema in Figure 11.39. Here academics have one of three ranks (L = Lecturer, SL = Senior Lecturer, P = Professor). Students may be counseled only by senior lecturers, and academic chairs are held only by professors. Since the subtypes do not form a partition of Academic, we would normally choose either absorption or separation to map them. As an exercise try both these options yourself before reading on.

Figure 11.39 Senior lecturers and professors play specific roles.

In the counseling predicate, the role attached to SeniorLecturer is not functional. So this predicate maps to a separate table regardless of whether we choose absorption or separation. Also, being *m:n*, the degrees fact type maps to a separate table regardless. So the only choice we have in the subtype mapping is whether or not to group the rank and chair fact types together. With absorption, we do so (see Figure 11.40(a)). With separation we do not (see Figure 11.40(b)). Note the different ways of specifying the subtype and mandatory role constraints for the Professor subtype.

Let's now restrict our attention to the mapping of functional predicates of the nodes in the subtype graph, since the mapping of other predicates to separate tables with qualified subset constraints is straightforward.

As an example with multiple inheritance, consider Figure 11.41. This figure also shows one way of displaying an ER view of an ORM diagram. Here all attributes are single valued; underlining an attribute indicates the 1:1 nature. The absence of a mandatory role dot on nrKids (number of children) indicates that recording of this attribute is optional for female academics.

Suppose we decide on separate tables for each subtype. This yields the relational schema in Figure 11.42. Note that qualified subset constraints from FemaleProfessor.empNr to Professor.empNr and FemaleAcademic.empNr are implied.

Figure 11.40 Mapping Figure 11.39, using subtype absorption (a) or separation (b).

Figure 11.41 An ER view with multiple inheritance.

For efficiency reasons we adopt the default policy of specifying qualified subset constraints with respect to the root supertable where possible. The order of table creation then does not matter as long as the root table is created first, and the schema is easier to change.

Figure 11.42 Relational map of Figure 11.41, choosing subtype separation.

Mixed approaches may be adopted. For example, if we absorb FemaleProfessor and Professor into Academic, but map roles specific to FemaleAcademic to a separate table, we obtain the relational schema in Figure 11.43. Notice that qualification 2 captures two subtype constraints. Other mixtures are possible, but this gives the idea.

Figure 11.43 Relational map of Figure 11.41, separating FemaleAcademic only.

Section 6.6 discussed the awkward situation of *context-dependent reference schemes*, where a subtype may have a different preferred identification scheme from its supertype(s). Let's briefly discuss how such cases may be mapped.

Recall that a *direct* supertype of a subtype is connected directly to the subtype (i.e., with no intermediate subtypes on the connecting path). For each subtype, exactly one of its direct supertypes must be declared as its *primary supertype*. A connection between a subtype and its primary supertype is shown as a solid arrow. Connections from a subtype to its secondary supertypes (if any) are shown as dashed arrows. *By default, a subtype inherits the preferred identification scheme of its primary supertype*. In this case, the preferred reference scheme for the subtype is not displayed. If a preferred reference scheme is shown explicitly for a subtype, this is a *context-dependent reference scheme* that differs from the reference scheme of the primary supertype.

In mapping, all specific roles attached to an entity type require the object type to be identified by its preferred reference scheme. Adopting this requirement avoids some extremely complex reference constraints that could otherwise arise in practice.

If two overlapping subtypes have the same preferred reference scheme, and one is not a subtype of the other, they must have a common supertype with this preferred reference scheme (if not, create one). This is needed to avoid redundancy later.

For example, suppose we allowed FemaleStudent(StudentNr) and PostGradStudent(StudentNr) as direct subtypes of Person(.nr). Mapping personNr down into tables for both subtypes would create redundancy of the facts that associate personNr and studentNr for the intersection of the subtypes. So this pattern is not allowed. Instead we insert the intermediate supertype Student(.nr) as a subtype of Person(.nr) and a supertype of FemaleStudent and PostGradStudent, which now inherit their identification scheme from Student (which is used for establishing the correspondence between personNr and studentNr).

We confine our discussion here to the subtype separation option (separate tables are created for functional roles specific to each subtype) and assume that all reference schemes are simple (not composite). For each subtype with a preferred reference mode different from the root's, we define its "*total table*" (if any) as set out in the following total table procedure.

Total Table Procedure:

```
if the object type has a mandatory disjunction of one or more functional roles
  all of which map to the same table
then the table to which its functional roles map is its total table
else if it has a mandatory nonfunctional role
      then select one of these roles arbitrarily;
            the table to which this role maps is its total table
      else { all roles are optional }
            if the object type is independent or is a subtype with separation mapping option
            then its table is the total table
            else if the object type plays only functional roles or only one role
                  all of which map to the same table
                then the table for this is its total table
                else the object type has no total table.
```

The basic idea is that the total table of an object type contains its total population (i.e., it includes all instances in the population of the object type). If an object type does have a total table, this procedure will find it. In what follows, where total tables exist we basically ignore other tables of the object type (the other tables are linked by foreign keys to the primary ones in the normal way).

The root supertype table(s) is/are computed in the normal way. Copy the preferred reference of the root supertype down as an extra attribute in the total table of each subtype, which "introduces" a different preferred reference scheme to the graph; if the introducing subtype has no total table, create an extra reference table for it (to store just the fact connecting the two identifiers for the object type's population).

In specifying, at the relational level, subtype links between object types with different primary reference schemes use the root reference scheme; if the object types have the same reference scheme, use this common reference scheme.

These guidelines are best understood by way of example. Consider the schema in Figure 11.44. Here the subtypes are all asserted rather than derived. Better students are often employed to do some tutoring. If known, the number of hours tutoring undertaken by such a student is recorded. In practice many other fact types might be stored about each node. Within the student records subschema, students are identified by their student number (studentNr). Within the staff record subsystem, all employees are identified by the employee number (employeeNr). A student may be an employee, and we want to know when this occurs. If it did not already exist, we introduce a person number (personNr) to enable people to be identified across the global schema. For student employees, we have three possible identifiers: personNr, studentNr, or employeeNr.

In this example, we chose employeeNr as the preferred identifier for student employees in the context of roles specific to StudentEmployee. This choice is shown by the solid subtyping arrow from StudentEmployee to Employee (indicating Employee as its primary supertype), together with the lack of an explicit context-dependent reference scheme for StudentEmployee. Although StudentEmployee inherits its identification scheme from Employee, it also has Student as a direct, secondary supertype (as shown by the dashed subtyping link).

Figure 11.44 A subtype graph with differing reference schemes.

Figure 11.45 The relational map of Figure 11.44, choosing subtype separation.

Choosing subtype separation for mapping each subtype, each of the three supertypes has a total table that is identical to its functional table. The functional fact types of the nodes in the subtype graph map to the four tables shown in Figure 11.45. Here an ellipsis "…" denotes any other (functional) attributes omitted in Figure 11.44.

If there are other mandatory columns in StudentEmployee, then tutePeriod becomes optional. Notice that personNr is not included in the StudentEmployee table: this avoids duplicating any instances of Person(.nr) has StudentNr(). Other mapping options exist. For example, we might absorb StudentEmployee into Employee.

Note that subtype links imply *is-associations* as depicted in the underlying explanatory schema of Figure 11.46. The four subtype-to-supertype connections in Figure 11.44 are actually metafact types (relationships between types) that imply the instance level fact types shown here. The subtyping link from StudentEmployee to Employee implies the reference type StudentEmployee is Employee.

The implied is-associations from Student and Employee to Person allow us to express facts such as "The employee with employeeNr 23 is the person with personNr 507".

Figure 11.46 Subtype links to new reference schemes are treated as base fact types.

The implied is-association from StudentEmployee to Student is a derived association, equivalent to the longer path from StudentEmployee through Person to Student, as shown in the derivation rule included in Figure 11.46.

By default, derived fact types are not stored. Instead, their derivation rules are used to compute the derived values on request. Apart from saving storage space, this "derive on query" (lazy evaluation) approach ensures that every time the derived information is requested it will be based on the latest values of the stored data. Sometimes, however, the same derivations are required many times with large volumes of stable data. In such cases, it can be much more efficient to store the derived information so that it can be accessed immediately at later times without having to be recomputed. Typically in such cases a "derive on update" (eager evaluation) approach is used so that as soon as the base data is updated, the derived information is computed and stored. Recall the bank balance example from Exercise 6.4.

In step 0.4 of the mapping procedure, any derived fact types to be stored should be included on the schema diagram and marked "**", with the derivation rule also declared. The fact type should then be grouped into a table in the normal way, and the derivation rule mapped as well. For example, the Window schema in Figure 11.47(a) uses the default mapping so that no column for area is included in its base table. However the area fact type in Figure 11.47(b) is marked "**" for storage, so it maps to a base table column.

The rule for computing area may be included in insert and update triggers, or a generated column declaration, a view, or a stored procedure. Care is required with storing derived data. For example, unless the derivation rule is fired every time the relevant base data is updated, the derived values can become outdated.

Figure 11.47 Derived fact types may be evaluated lazily (a) or eagerly (b).

Figure 11.48 PlantKind has a disjunctive reference scheme.

For symmetric 1:1 binaries (roles are both optional or both mandatory, and both object types have other functional roles) we have a choice as to how the 1:1 binary should be mapped. This mapping choice should be noted in step 0.5 (e.g., by annotation or option selection) and adhered to when the fact type is mapped in step 2.

In rare cases a conceptual schema may include a disjunctive preferred reference scheme (identification by a mandatory disjunction of two or more roles, at least one of which is optional). In Figure 11.48, PlantKind is identified in this way (this botanical naming convention was discussed in Chapter 5). When mapped to a relational schema, such schemes can prove awkward to handle.

For example, the conceptual schema in Figure 11.48 maps to the following relational schema. This violates the relational entity integrity rule, since the primary keys may contain nulls. We enclose both infraRank and infraName in the same pair of square brackets to indicate that the qualification applies to *both*, and that if one is null so is the other. Enclosing both of these inside the option brackets for species indicates that they can only be given a nonnull value when species is nonnull.

PlantKind (genus, [species, [infraRank, infraName]], author)

DisplayedIn (genus, [species, [infraRank, infraName]], museum)

Although forbidden by the relational model, this is allowed in most relational database systems (which permit tables without primary keys), and the relevant uniqueness constraint can be enforced by a procedure or assertion. However, a simpler implementation can often be obtained by altering the conceptual schema to replace the disjunctive reference with a nondisjunctive one. In step 0.6 we consider such replacements.

There are three basic ways of replacing a disjunctive reference scheme: artificial identifiers, concatenated identifiers, or use of special default values. Let's look at these three alternatives briefly, using this botanical example. The first option is to introduce a new identifier for PlantKind (e.g., plantKindNr), leading to the following relational schema. This approach has two main advantages: the primary keys have no nulls, and table joins are faster (now based just on plantKindNr).

PlantKind (plantKindNr, genus, [species, [infraRank, infraName]], author)

DisplayedIn (plantKindNr, museum)

The artificial identifier option is not all good news. The secondary key in PlantKind still has optional fields. Moreover, since this secondary key uses the natural reference scheme, this is what users will normally want to see (not the artificial plantKindNr). For example, to find what kinds of plant are on display in a museum, we now need to join the tables since we want to see the natural plant names.

A simpler solution is just to concatenate the formerly separate parts of the identifier into a single name. This leads to the very simple relational schema:

PlantKind (plantKindName, author)

DisplayedIn (plantKindName, museum)

This simple schema is the best solution if we don't need to list or group the formerly separate parts of the identifier. However if we wanted to issue a query such as "Which species has the most plant kinds?" there is now no simple way to do this.

The third option is to keep the original components of the reference scheme separate but make each of them mandatory by using special default values when no actual value exists. For example, the plant kind *acacia interior* could be stored as the tuple ('acacia', 'interior', 'nil', 'nil') using "nil" as a special default value (different from null) to indicate that plant kind has no actual value for infraRank and infraName. The conceptual schema in Figure 11.48 is modified by making the species and infraspecies predicates mandatory for PlantKind, deleting the subset constraint, and adding "nil" to any value list for Rank, e.g., {'ssp', 'var', 'forma', 'n-ssp', 'n-var', 'n-forma', 'nil'}. The following relational schema is obtained (value-list omitted):

PlantKind (genus, species, infraRank, infraName, author)

DisplayedIn (genus, species, infraRank, infraName, museum)

In this case the default value "nil" is unlikely to ever be confused with an actual value. However, in some cases, such confusion might arise (e.g., a default score of 0) and the user may then be burdened with the responsibility of distinguishing default from actual values. Also, bag functions that ignore nulls will not ignore nils.

The final refinement to step 0 of the mapping procedure (step 0.7) deals with cases where an objectified predicate is not spanned by a uniqueness constraint. Some versions of ORM forbid this, since it violates elementarity. However, if it is allowed, then some pre-processing is needed before executing the grouping part of the procedure.

In nested 1:1 cases, a decision should be made to favor one of the roles in grouping. For example, suppose the schema in Figure 11.49 is used to model current marriages in a monogamous society. This schema violates elementarity. It can be split into two fact types: one about the marriage and one about the marriage year. If no other functional roles are played by Person, we might map this to either of the tables schemes shown in Figure 11.49. For such 1:1 cases a choice must be made (e.g., by choosing one of the uniqueness constraints as primary).

If only one role in the objectified predicate is functional, it is automatically chosen for the primary key. For example, in a polyandrous society where a man may be married to at most one woman but not vice versa, the husband column would be chosen as the primary key (and wife would no longer be a key). Despite the provisions of step 0.7, objectified *n*:1 associations should normally be avoided.

Once the columns and constraints are determined in a relational mapping, you may wish to consider changing the *order of columns* within a table. Although the relational model treats column order as irrelevant, in practice, column order can have an impact on both readability and performance. A CASE tool will choose an ordering for you, but you can override this.

Usually it is best to include primary key columns first, and to group columns together if they are components of a compositely identified object type or have a strong semantic affinity. In such cases, a top-down order is usually preferable (e.g., buildingNr, roomNr instead of roomNr, buildingNr). For such composite cases, placing the attributes with fewer values first can help speed up joins in some cases. A similar comment applies to composite indexes. Some systems also perform better if optional columns are placed last.

Figure 11.49 Two possible mappings for an objectified 1:1 association.

Exercise 11.4

1. Map each of the following ORM schemas to a relational schema. Use the horizontal notation. Underline keys, place square brackets around optional columns, and declare any textual constraints. A solid line unary indicates closed world, a simple dashed line indicates open world, and a dashed line around "~" indicates open world with negation. GraduationLevel codes have the following meaning: STD = Standard, CL = cum laude, MCL = magna cum laude, SCL = summa cum laude.

(e) As for (d) but OWA unary: first graduated
(f) As for (e) but OWN unary: first graduated

2. Suppose heads of government are identified by the country that they head, and have their salary and country of birth recorded. Schematize this and Rmap it.

3. Refer to the hospital UoD of Exercise 6.6 Question 2. Rmap the conceptual schema for part (a), then discuss any changes in the mapping for parts (b), (c) and (d).

4. Consider a UoD in which people are identified by the combination of their surname, first given name and (if it exists) second given name. Each person's weight is recorded, as well as the sports they play (if any).
   (a) Schematize this UoD.
   (b) Rmap this, using the given identification scheme for persons.
   (c) Introduce personNr as the primary identifier. Rmap the new conceptual schema.
   (d) Instead, concatenate surname and given names to a single name. Rmap this.
   (e) Instead, introduce "nil" as a default for no second given name. Rmap this.
   (f) Which do you prefer?

5. Consider the functional fact type: Flight(.nr) departs Airport(.code). Suppose that facts about the flight departure are modeled by objectifying the previous association as "FlightDeparture", and attaching the fact type: FlightDeparture occurred at Time(hm).
   (a) Draw the conceptual schema.
   (b) Unnest the schema.
   (c) Rmap it

11.5 Summary

Conceptual schemas are normally implemented by mapping them to a logical schema (e.g., relational), refining this, and then generating internal and external schemas (including access rights). Updates and queries may then be performed on the databases and schema(s).

A *relational (database) schema* is a set of table definitions (stored base tables or derived views), constraints, and derivation rules. A *table scheme* is a named set of *attributes* (columns) that draw their values from *domains*. Each column, or column set, spanned by a minimal uniqueness constraint is a *candidate key*. Keys are underlined. Each table row is identified by its *primary key* (doubly underlined if another key exists). The *entity integrity* rule requires primary keys to have no nulls. Optional columns

allow nulls and are enclosed in square brackets. For example: *Employee*(empNr, empName, address, gender, [phone]).

Mandatory roles map to non-optional columns, with subset constraints running from any other tables that contain facts about that object type. A *referential integrity constraint* is a subset constraint from a foreign key to some primary key. Subset constraints between tables appear as dotted arrows. Other constraint notations are used.

The relational mapping procedure (Rmap) groups each fact type into a single table, using two basic ideas: each fact type with a compound uniqueness constraint (e.g., an *m:n* association or a ternary association) is mapped to a separate table; fact types with functional roles attached to the same object type are grouped into the same table, keyed on the object type's identifier.

The *Basic Rmap procedure* may be summarized as follows. Absorb subtypes into their top supertype. Mentally erase all explicit preferred identification schemes, treating compositely identified object types as "black boxes". Map each fact type with a compound UC to a separate table. Group fact types with functional fact roles attached to the same object type into the same table, keyed on the object type's identifier. Map 1:1 cases to a single table, generally favoring fewer nulls. Map each independent object type with no functional fact roles to a separate table. Unpack each "black box column" into its component attributes. Map all other constraints and derivation rules. Subtype constraints on functional roles map to qualified optional columns, and those on nonfunctional roles map to qualified subset constraints.

If the total population of an object type is included in one table, this is called a total table for the object type. See the *total table procedure* for more details. Advanced mapping aspects in the *Rmap step 0 procedure* may be summarized as follows. Transform exclusive unaries and then map unaries according to their open/closed world semantics. Mentally erase all reference (preferred identification) predicates. Treat compositely identified object types as "black boxes". Indicate any absorption overrides (e.g., separation or partition) for subtypes. Identify any derived fact types that must be stored. Indicate mapping choices for symmetric 1:1 cases. Consider replacing any disjunctive reference schemes by using an artificial or concatenated identifier or mandatory defaults. Indicate mapping choice where required for any objectified associations with no spanning uniqueness constraint.

Chapter Notes

Many database textbooks discuss mapping ER schemas to relational schemas (e.g., Batini et al. 1992) and mapping UML class diagrams to relational schemas (e.g., Blaha and Premerlani 1998). The mapping extensions to the old ONF algorithm to develop the original version of Rmap were developed jointly by Peter Ritson and Terry Halpin. For more details on mapping 1:1 predicates, see Ritson and Halpin (1993a). For further details on Rmap and its SQL version see Ritson and Halpin (1993b). The version of Rmap discussed in this chapter includes several revisions to cater for new features in ORM 2, such as further options for objectification and open/closed world semantics.

12 Relational Languages

12.1 Relational Algebra
- Union, Intersection and Difference
- Cartesian Product (Unrestricted Join)
- Relational Selection
- Relational Projection
- Relational Joins
- Relational Division
- Query Strategies

12.2 Relational Database Systems

12.3 SQL: Historical and Structural Overview

12.4 SQL: Identifiers and Data Types

12.5 SQL: Choosing Columns, Rows, and Order

12.6 SQL: Joins

12.7 SQL: in, between, like, and is null Operators

12.8 SQL: Union and Simple Subqueries

12.9 SQL: Scalar Operators and Bag Functions

12.10 SQL: Grouping

12.11 SQL: Correlated and Existential Subqueries

12.12 SQL: Recursive Queries

12.13 SQL: Updating Table Populations

12.14 Summary

12.1 SQL: Relational Algebra

So far we've seen how to design a conceptual schema and map it to a relational schema. The relational schema may now be implemented in a relational DBMS, and its tables populated with data. To retrieve information from the resulting database, we need to issue queries. In practice the most popular query languages are *SQL* (popularly, if incorrectly, called "Structured Query Language") and *QBE* (Query By Example). Both of these are based, at least partly, on a formal query language known as *relational algebra*, which is discussed in this section. Studying this algebra first clarifies the basic query operations without getting distracted by the specific syntax of commercial query languages.

With the algebra under our belt, we will be able define what is really meant by the term "relational database system". The following sections then cover SQL in some depth, starting with the basics and moving on through intermediate to advanced concepts. Although our focus is on the SQL standard(s), some additional details will be included about commercial dialects of SQL. Now let's begin with relational algebra.

In the relational model of data, all facts are stored in tables (or relations). New tables may be formed from existing tables by applying operations in the relational algebra. The tables resulting from these operations may be named and stored, using relational assignment. The original relational algebra defined by Codd contained eight *relational operators*: four based on traditional set operations (union, intersection, difference, and Cartesian product) and four special operations (selection, projection, join, and division). Each of these eight relational operators is a table-forming operator on tables. For example, the union of two tables is itself a table.

Relational algebra includes six *comparison operators* (=, <>, <, >, <=, >=). These are proposition-forming operators on terms. For example, `x <> 0` asserts that *x* is not equal to zero. It also includes three *logical operators* (**and**, **or**, **not**). These are proposition-forming operators on propositions (e.g., `x > 0` **and** `x < 8`). Since the algebra does not include arithmetic operators (e.g., +) or functions (e.g., **count**), it is less expressive than SQL. Proposed extensions to the algebra to include these and other operators are ignored here.

Many different notations exist for expressing the relational algebra. A comparison between our notation and a common academic notation is given later. To simplify discussion, we'll often use informal terms instead of the strict relational terminology (e.g., "table" and "row" instead of "relation" and "tuple").

Union, Intersection, and Difference

Two tables are *union compatible* if and only if they have the same number of columns and their corresponding columns are based on the same domain (the columns may have different names). Treating a table as a set of rows, the traditional set operations of union, intersection, and difference may be defined between tables that are union compatible.

Figure 12.1 The conceptual schema maps to two tables.

Eats:

person	*food*
Ann	pizza
Ann	spinach
Bill	pizza

Likes:

person	*food*
Ann	chocolate
Ann	pizza
Bill	pizza
Bill	steak

Eats ∪ Likes ⇨

person	*food*
Ann	pizza
Ann	spinach
Bill	pizza
Ann	chocolate
Bill	steak

Figure 12.2 The union of two tables is the set of rows that occur in either table (or both).

Consider the conceptual schema in Figure 12.1. In this small UoD, people are identified by their first name. We record facts about the foods that people eat, and about the foods that they like. The relational schema comprises two tables as shown. The populations of the tables may properly overlap or even be disjoint. Hence there are no intertable constraints in the relational schema—this situation is fairly unusual.

Figure 12.2 includes sample data. The **union** of tables A and B is the set of all rows belonging to A or B (or both). We write this as "$A \cup B$" or "A **union** B". Suppose that we want to pair each person up with foods that they either eat or like. This may be specified simply as the union of the tables, Eats ∪ Likes. Figure 12.2 includes this query expression and the resulting (unnamed) table, which includes all the rows in the Eats table as well as all the rows in the Likes table. As with any table, duplicate rows are excluded, so rows in both the Eats and the Likes tables appear only once in the result.

Note that "Eats ∪ Likes" is an expression describing how the result may be derived in terms of tables stored in the database (i.e., base tables): it is not a table name. We may refer to a query result as a result table, answer table, or derived table. The order of the rows in the tables has no significance, since we are dealing with sets. Actual query languages such as SQL provide ways to display rows in any preferred order.

The **intersection** of tables A and B is the set of rows common to A and B. We write this as "$A \cap B$" or "A **intersect** B". For instance, to list facts about foods that people both eat and like, we may specify this as the intersection of our two base tables, Eats ∩ Likes (see Figure 12.3).

Eats ∩ Likes ⇨

person	*food*
Ann	pizza
Bill	pizza

Figure 12.3 The intersection of two tables is the set of rows common to both.

(a) Eats - Likes ⇨

person	*food*
Ann	spinach

(b) Likes - Eats ⇨

person	*food*
Ann	chocolate
Bill	steak

Figure 12.4 (a) $A - B$ is the set of rows in A that are not in B; (b) $B - A$.

We define the **difference** operation between tables thus. $A - B$ is the set of rows belonging to A but not B. We may also write this as A **minus** B, or A **except** B. For example, the expression Eats – Likes returns details about people who eat foods they don't like (see Figure 12.4(a)).

As in ordinary set theory, the union operation is commutative (i.e., the order of the operands is not significant). So, given any tables A and B it is true that $A \cup B = B \cup A$. Likewise, the intersection operation is commutative, i.e., $A \cap B = B \cap A$. The difference operation, however, is not commutative so it is possible that $A - B$ does not equal $B - A$. For example, Likes – Eats returns facts about who likes a food but doesn't eat it (see Figure 12.4(b)). Compare this with the previous result.

Cartesian Product (Unrestricted Join)

In mathematics, the **Cartesian Product** of sets A and B is defined as the set of all ordered pairs (x, y) such that x belongs to A and y belongs to B. For example, if $A = \{1, 2\}$ and $B = \{3, 4, 5\}$, then the Cartesian Product of A and B is $\{(1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5)\}$. We write this product as $A \times B$ (read "A **cross** B"). This definition applies also to tables, although the ordered pairs of rows (x, y) are considered to be single rows of the product table, which inherits the corresponding column names. Thus if A and B are tables, $A \times B$ is formed by *pairing each row of A with each row of B*. Here "pairing with" means "prepending to". The expression "$A \times B$" may also be written as "A **times** B". The Cartesian product is also called the "cross join" or "unrestricted join".

Since $A \times B$ pairs each row of A with all rows of B, if A has n rows and B has m rows, then the table $A \times B$ has $n \times m$ rows. So *the number of rows in* $A \times B$ *is the product of the number of rows in A and B*. Since the paired rows are concatenated into single rows, *the number of columns in* $A \times B$ *is the sum of the number of columns in A and B*.

Let's look at a simple example. In Figure 12.5, A is a 3×2 table (i.e., 3 rows $\times$ 2 columns) and B is a 2×1 table. So $A \times B$ has six (3×2) rows of three $(2 + 1)$ columns. The header rows that contain the column names *a1, a2, b1* are not counted in this calculation. In this example, dotted lines have been ruled between rows to aid readability. Note that the Cartesian Product $A \times B$ exists for any tables A and B; it is not necessary that A and B be union compatible.

If we define Cartesian products in terms of tuples treated as ordered sequences of values, the $\times$ operator is not commutative (i.e., $A \times B$ need not equal $B \times A$). This is the usual treatment in mathematics. With relational databases, however, each column has a unique name, so a tuple may instead be treated as an (unordered) set of attribute-value pairs (each value in a row is associated with its column name). In this sense, columns are said to be "ordered by name". With this view of tuples, the $\times$ operator is commutative.

To clarify this, compute $B \times A$ for the example in Figure 12.5. Although the resulting table has its columns displayed in the order *b1, a1, a2,* and the values are ordered accordingly, the information is the same as for $A \times B$, since each value can be linked to its column name. For example, the first row of $B \times A$ may be represented as {(b1, 7), (a1, 1), (a2, 2)}, which equals {(a1, 1), (a2, 2), (b1, 7)}, which represents the first row of $A \times B$. Once columns are uniquely named, their physical order has no semantic significance.

As in standard set theory, the operations of union, intersection, and Cartesian product are *associative*. For instance, $A \cup (B \cup C) = (A \cup B) \cup C$. Since no ambiguity results, parentheses may be dropped in such cases. For example, the expression "$A \cup B \cup C$" is permitted. Of the four table operations considered so far, difference is the only one that is not associative; that is, it is possible that $A - (B - C)$ is not equal to $(A - B) - C$. As an exercise, use Venn diagrams to check these claims.

A:

a1	*a2*
1	2
3	4
5	6

B:

b1
7
8

A × B ⇨

a1	*a2*	*b1*
1	2	7
1	2	8
3	4	7
3	4	8
5	6	7
5	6	8

Figure 12.5 $A \times B$ pairs each row of A with each row of B.

Person:

firstname	*gender*
Fred	M
Sue	F

Person × Person ⇨

firstname	*gender*	*firstname*	*gender*
Fred	M	Fred	M
Fred	M	Sue	F
Sue	F	Fred	M
Sue	F	Sue	F

Figure 12.6 What is wrong with this Cartesian product?

define alias Person2 **for** Person

Person × Person2 ⇨

Person.firstname	*Person.gender*	*Person2.firstname*	*Person2.gender*
Fred	M	Fred	M
Fred	M	Sue	F
Sue	F	Fred	M
Sue	F	Sue	F

Figure 12.7 Table aliases are needed to multiply a table by itself.

Recall that column names of a table must be unique. If A and B have no column names in common, then $A \times B$ uses the simple column names of A and B. If A and B are different but have some common column names, the corresponding columns in $A \times B$ are uniquely referenced by using *qualified names* (with the prefix "*A*." or "*B*."). However, what if A and B are the same (i.e., we wish to form the Cartesian product of a table with itself)?

Consider the Person × Person table in Figure 12.6. This breaks the rule that column names in a table must be unique. In this case, simply using the qualified column names will not solve the problem either because we would have two columns named "Person.firstname" and two columns named "Person.gender".

This problem is overcome by introducing **aliases** for tables. For example, we could use "Person2" as an alternative name for the table and form the product as shown in Figure 12.7. If desired, we could have introduced two aliases, "Person1" and "Person2", to specify the product Person1 × Person2. The need to multiply a table by itself may arise if we have to compare values on different rows. Aliases can also be used simply to save writing (by introducing shorter names).

Relational Selection

Let us now consider the four table operations introduced by Codd. The first of these is known as selection (or *restriction*). Don't confuse this with SQL's **select** command.

In relational algebra, the selection operation chooses just those rows that satisfy a specified condition. Visually we can picture the operation as returning a *horizontal* or *row* subset, since it returns zero or more rows from the original table (see Figure 12.8).

The selection operation may be specified using an expression of the form *T* **where** *c*. Here *T* denotes a *table expression* (i.e., an expression whose value is a table) and *c* denotes a *condition*. The selection operation returns just those rows where the condition evaluates to True, filtering out rows where the condition evaluates to False or Unknown. The "**where** *c*" part is called a *where* clause. This notation encourages top-down thinking (find the relevant tables before worrying about the rows) and agrees with SQL syntax. The alternative notation $\sigma_c(T)$ is often used in academic journals. The "σ" is sigma, the Greek "*s*" (which is the first letter of "selection").

In Figure 12.9, the Player table stores details about tennis players. To list details about just the male tennis players, we could formulate this selection as indicated, giving the result shown. Here the selection condition is just the equality: gender = 'M'. The use of single quotes with 'M' indicates that this value is literally the character 'M'. Single quotes are also used when referencing character string values such as 'David'. Numeric values should not be quoted. Although the table resulting from the selection is unnamed, its columns have the same names as the original.

Figure 12.8 The selection operation in relational algebra.

Player:

name	*gender*	*height*
David	M	172
Norma	F	170
Selena	F	165
Terry	M	178

Player **where** gender = 'M' ⇨

name	*gender*
David	M
Norma	F
Selena	F
Terry	M

Figure 12.9 The selection operation picks those rows that satisfy the condition.

Player **where** (gender = 'F') **and** (height < 170)

⇨

name	*gender*	*height*
Selena	F	165

Figure 12.10 Are the parentheses needed in this query?

The condition *c* may contain any of six *comparison operators*: = (equals); <> (is not equal to); < (is less than); > (is greater than); <= (is less than or equal to); and >= (is greater than or equal to). Sometimes the result of a table operation is an empty table (cf. the null set). For instance, no rows satisfy the condition in Player **where** height > 180.

If conditions do not involve comparisons with null values, they are Boolean expressions—they evaluate to True or False. If conditions do include comparisons with nulls, a three-valued logic is used instead, so that the condition may evaluate to True, False, or Unknown. This is also what happens in SQL. In either case, the selection operation returns just those rows where the condition is true. Any row where the condition evaluates to false or unknown is filtered out.

In addition to comparison operators, conditions may include three *logical operators*: **and**, **or**, **not**. As usual, or is inclusive. As an exercise, formulate a query to list details of those females who are shorter than 170 cm. Refer to the Player table and see if you can state the result, and then check your answer with Figure 12.10. To aid readability, reserved words are in bold.

The query in Figure 12.10 used parentheses to clarify the order in which operations are to be carried out. With complicated expressions, however, large numbers of parentheses can make things look very messy (cf. LISP). Unlike several programming languages (e.g., Pascal), most query languages (e.g., SQL) give *comparison operators higher priority than logical operators*. So the operators =, <>, <, >, <=, >= are given precedence over **not**, **and**, and **or** for evaluation purposes. We make the same choice in relational algebra since this reduces the number of parentheses needed for complex expressions. With this understood, the query in Figure 12.10 may be rewritten more concisely as:

Player **where** gender = 'F' **and** height < 170

To further reduce the need for parentheses, the algebra adopts the logical operator priority convention commonly used in computing languages, such as Pascal and SQL. First evaluate **not**, then **and**, and then **or**. Table 12.1 shows the evaluation order for both comparison and logical operators. This operator precedence may be overridden by the use of parentheses.

For example, suppose we want details about males who are taller than 175 cm or shorter than 170 cm. The query in Figure 12.11(a) looks like it should work for this, but the wrong result is obtained. Why?

Table 12.1 Priority convention for the operators (1 = first).

Priority	*Operators*
1	=, <>, <, >, <=, >=
2	**not**
3	**and**
4	**or**

(a) Player **where** gender = 'M'
and height > 175 **or** height < 170 ⇨

name	*gender*	*height*
Selena	F	165
Terry	M	178

(b) Player **where** gender = 'M'
and (height > 175 **or** height < 170) ⇨

name	*gender*	*height*
Terry	M	178

Figure 12.11 Unless parentheses are used, **and** is evaluated before **or**.

(a) Player **where** gender = 'M' **and** height > 175
or gender= 'F' **and** height >= 170 ⇨

(b) Player **where** (gender= 'M' **and** height > 175) ⇨
or (gender= 'F' **and** height >= 170)

name	*gender*	*height*
Terry	M	178
Norma	F	170

Figure 12.12 Two equivalent queries; if in doubt, include parentheses.

Because the **and** operator is evaluated before the **or**, this query is interpreted as Player **where** (gender = 'M' **and** height > 175) **or** height < 170. Notice that line breaks have no significance to the meaning of the query. So the result includes details about anybody shorter than 170 cm (including females). The query in Figure 12.11(b) corrects this by including parentheses around the height disjunction to evaluate the **or** operator before the **and** operator.

To obtain details about players who are either males over 175 cm or females at least 170 cm tall, either of the queries shown in Figure 12.12 may be used. In the query in Figure 12.12(b), parentheses are redundant since the **and** operators are evaluated before the **or** operator anyway. If in doubt about whether parentheses are needed, put them in.

Figure 12.13 shows three equivalent queries for listing information on players who are neither taller than 172 cm nor shorter than 168 cm. Operator precedence ensures that query (a) in Figure 12.13 means Player **where** (**not** (height > 172)) **and** (**not** (height < 168)). Note that the six comparison operators may be grouped into three pairs of opposites: =, <>; <, >=; and >, <=. So the query in Figure 12.13(a) may be replaced by the shorter query in Figure 12.13(b).

(a) Player **where not** height > 172 **and not** height < 168 ⇨

(b) Player **where** height <= 172 **and** height >= 168 ⇨

(c) Player **where not** (height > 172 **or** height < 168) ⇨

name	*gender*	*height*
David	M	172
Norma	F	170

Figure 12.13 Three equivalent queries.

Figure 12.14 The projection operation in relational algebra.

De Morgan's laws, named after the famous logician Augustus De Morgan, are the following:

not (p **and** q) ≡ **not** p **or not** q

not (p **or** q) ≡ **not** p **and not** q

Here *p* and *q* denote any proposition or logical condition. These laws hold in both two-valued and three-valued logic, so are safe to use with null values. Using the second of these laws, it is easy to see that the query in Figure 12.13(c) is equivalent to the query in Figure 12.13(a).

Relational Projection

The next table operation, known as **projection**, was introduced in Chapter 4. This operation involves *choosing one or more columns from a table* and then *eliminating any duplicate rows that might result*. It is said to produce a *vertical subset* or *column subset*. Figure 12.14 illustrates a projection on two columns.

We may represent the projection operation as *T [a,b,…]* where *T* is a table expression and *a, b,…* are the names of the required columns (this column list is called the *projection list*). To delimit the projection list we use square brackets rather than parentheses, since the latter have other uses in queries (e.g., to change the evaluation order of operations). Italicizing the square brackets "*[ ]*" helps distinguish them from the use of "[ ]" to delimit optional items, or bags, and to indicate an extra task (duplicate elimination) once the columns are chosen.

The alternative notation $\pi_{a,b\ldots}(T)$ is common in academic journals. The "π" symbol is pi, the Greek "*p*" (the first letter of "projection"). Like SQL, this notation lists the

required columns before the tables. Our preferred notation instead encourages top-down thinking by identifying the relevant tables before listing the columns.

Figure 12.15 gives two examples, based on the table *Player*(name, gender, height). When projecting on a single base table, if the chosen columns include a candidate key then no duplicates rows can result. If not, duplicates may arise. For example, choosing just the gender column gives the bag [‘M’, ‘F’, ‘F’, ‘M’]. To complete the projection the duplicate values are eliminated, giving the set {‘M’, ‘F’}. If we project on all columns, we end up with the same table. For instance, Player *[*name, gender, height*]* is the same table as Player.

The same column must not be mentioned twice in a projection. For example, Player *[*name, name*]* is illegal (why?). If desired, projection may be used to display columns in a different order. For example, Player *[*height, gender, name*]* reverses the order. Since unique column names are listed in the result, this doesn’t change the meaning.

If the relational schema is fully normalized, it is fairly unusual to have base tables that are union compatible (the Eats and Likes tables considered earlier being an exception). So the ∪, ∩, and − operations tend to be used almost exclusively with result tables that have become union compatible because of projections.

Let’s look now at some examples that are a little more difficult. To facilitate a discussion of a step-by-step approach to such cases, we’ll use the **relational assignment** operation. Because their contents may vary, table schemes may be regarded as *variables*. In fact, tables are the only kind of variable allowed in relational algebra. The notion of relational assignment, which is strictly separate from the algebra itself, is similar to that in programming languages. Using the symbol “:=” for “becomes” or “is assigned the value of”, we may write assignment statements of the form

table variable := table expression

Player:

name	*gender*	*height*
David	M	172
Norma	F	170
Selena	F	165
Terry	M	178

Player *[*name, gender*]* ⇨

name	*gender*
David	M
Norma	F
Selena	F
Terry	M

Player *[*gender*]* ⇨

gender
M
F

Figure 12.15 Projection involves picking the columns and removing any duplicates

This is an instruction to first evaluate the expression on the right, and then place its value in the variable named on the left. If this variable had a previous value, the old value would simply be replaced by the new value. For example, if X, A, and B are table values, then the statement "$X := A - B$" means X is assigned the value of $A - B$.

Once named in a relational assignment, result tables may be used just like base tables in later expressions. For example, to store details about male tall players in a table called "MaleTallPlayer", we could make the following three assignments. To list these details, our query is simply the table name MaleTallPlayer.

MalePlayer := Player **where** gender = 'M'

TallPlayer := Player **where** height > 175

MaleTallPlayer := MalePlayer ∩ TallPlayer

As a more difficult example, consider the UoD of Figure 12.16. How may we formulate this query in relational algebra: Which non-European countries speak French? Try this yourself before reading on.

We could formulate the query in steps, using intermediate tables on the way to obtaining our final result table. For example:

NonEuroCountry := (Location **where** region <> 'Europe') *[*country*]*

FrenchSpkCountry := (SpokenIn **where** language = 'French') *[*country*]*

FrenchSpkNonEuroCountry := NonEuroCountry ∩ FrenchSpkCountry

SpokenIn:

language	*country*
Dutch	Belgium
Dutch	Netherlands
English	Australia
English	Canada
French	Belgium
French	Canada
French	France

Location:

country	*region*
Australia	Australasia
Belgium	Europe
Canada	North America
France	Europe
Netherlands	Europe

Figure 12.16 Conceptual and relational schemas, with sample population.

Here our first step was to find the non-European countries: {Australia, Canada}. Then we found the French-speaking countries: {Belgium, Canada, France}. Finally we took the intersection of these, giving the result: {Canada}.

Setting queries out this way is known as *stepwise formulation*. Doing things in stages can sometimes make it easier to formulate difficult queries. However, it saves writing to express a query by means of a single expression. For instance, the three assignment statements just considered can be replaced by

(Location **where** region <> 'Europe') *[*country*]*
∩
(SpokenIn **where** language = 'French') *[*country*]*

This lengthy expression is said to be a *nested formulation* because it nests one or more queries inside another. For complex queries we might perform the stepwise formulation in our heads, or scribbled down somewhere, and then convert this into the nested formulation. In general, any information capable of being extracted by a series of relational algebra queries can be specified in a single relational algebra query.

Note that projection does not distribute over intersection. In other words, given tables A and B, and a projection list p, it is possible that $(A \cap B)[p] \neq A[p] \cap B[p]$. For instance, the following query is not equivalent to the previous one (why not?).

((Location **where** region <> 'Europe') ∩ (SpokenIn **where** language = 'French')) *[*country*]*

This query is illegal, since the table operands of ∩ are not compatible. Even if A and B are compatible, projection need not distribute over ∩ or over −. If A and B are compatible, projection does distribute over ∪, i.e., $(A \cup B)[p] = A[p] \cup B[p]$. As an exercise, you may wish to prove these results.

Relational Joins

Object joins at the conceptual level were introduced in Section 4.4 to help clarify the meaning of external uniqueness constraints. We now consider the relational join operation between two tables (or two occurrences of the same table), which compares attribute values from the tables, using the comparison operators (=, <, >, <>, <=, >=). There are several kinds of join operations, and we discuss only some of these here. Columns being compared in any join operation must be defined on the same domain; they do not need to have the same name. Where it is necessary to distinguish between join columns with the same local name, we use their fully qualified names.

Let Θ (theta) denote any comparison operator (=, < etc.). Then the **Θ-join** of tables A and B on attributes a of A and b of B equals the Cartesian product $A \times B$, restricted to those rows where $A.a$ Θ $B.b$. We write this as shown here. An alternative, academic notation is shown in braces.

$(A \times B)$ **where** c { or $A \bowtie_c B$ }

Figure 12.17 The ORM schema (a) maps to the relational schema (b).

Employee:

empNr	*empName*	*gender*
001	Hagar, T	M
002	Wong, S	F
003	Jones, E	F
004	Mifune, K	M

Drives:

empNr	*carRegNr*
001	ABC123
003	ABC123
003	TAH007

Figure 12.18 A sample population for the schema in Figure 12.17.

The condition c used to express this comparison of attributes between tables is called the *join condition*. The join condition may be composite (e.g., `a1 < b1` **and** `a2 < b2`). Because of the Cartesian product, the resulting table has a number of columns equal to the sum of the number of columns in A and B, but because of the selection operation, it typically has far fewer rows than the product of the numbers of rows of the joined tables.

With most joins, the comparison operator used is $=$. The Θ-join is then called an **equijoin**. Thus the equijoin of A and B equating values in column a of A with column b of B is $A \times B$ restricted to the rows where $A.a = B.b$. If the join column names are different in each table, there is no need to qualify them with table names. We write this as shown here. In general, the join condition may contain many equalities.

$(A \times B)$ **where** $A.a = B.b$
or
$(A \times B)$ **where** $a = b$ { if a, b occur in only one of A, B }

Let's consider an example. Figure 12.17.includes a simple conceptual schema and its corresponding relational schema. A sample population is given in Figure 12.18.

Before formulating queries you should *familiarize yourself with the structure and contents of the database*. From the schema, we see that employees are identified by their employee number, and their name and gender must also be recorded. An employee may drive many cars, and the same car may be driven by many employees. The population is not significant, since it does not include instances where the same name applies to more than one employee. However, it does illustrate that driving cars is optional

(employees 002 and 004), that an employee may drive more than one car (employee 003), and that the same car may be driven by more than one employee (car ABC123). Now consider the following query: "List the employee number, name, and cars for each employee who drives a car".

We could specify this by the equijoin in Figure 12.19, which joins the tables by equating their empNr attributes. To help explain this, dotted lines have been added between the base tables to indicate which rows match on their empNr value. Only the first and third rows from the Employee table are able to find a match. The third row (003) finds two matches. Check the result for yourself.

Because the join columns here have the same local name, "empNr", qualified names are used to distinguish the columns in the query and the join result. The join columns don't have to have the same (unqualified) name—as long as they belong to the same domain, the join can be made. For example, if the employee number column in the Drives table was named "empId", the join condition could be specified as "empNr = empId".

As this example illustrates, an equijoin contains two matching columns resulting from each join attribute (note the two empNr columns). If these columns actually refer to the same thing in the UoD (and they typically do), then one of these columns is redundant. In this case, we lose no information if we *delete one of these matching columns* (by performing a projection on all but the column to be deleted). This is done in Figure 12.20.

Employee:

empNr	*empName*	*gender*
001	Hagar, T	M
002	Wong, S	F
003	Jones, E	F
004	Mifune, K	M

Drives:

empNr	*carRegNr*
001	ABC123
003	ABC123
003	TAH007

(Employee × Drives)
where Employee.empNr = Drives.empNr

⇨

Employee.empNr	*empName*	*gender*	*Drives.empNr*	*carRegNr*
001	Hagar, T	M	001	ABC123
003	Jones, E	F	003	ABC123
003	Jones, E	F	003	TAH007

Figure 12.19 An equijoin.

(Employee × Drives)
where Employee.empNr = Drives.empNr
*[*Employee.empNr, empName, gender, carRegNr*]*

⇨

Employee.empNr	*empName*	*gender*	*carRegNr*
001	Hagar, T	M	ABC123
003	Jones, E	F	ABC123
003	Jones, E	F	TAH007

Figure 12.20 The duplicate column is removed by projection on the equijoin.

If the *columns used for joining have the same name in both tables*, then the unqualified name is used in the join result. The resulting table is then said to be the **natural inner join** of the original tables. Since "inner" is assumed by default, the **natural inner join** may be expressed simply as **natural join** (outer joins are discussed later). This is by far the most common join operation in practice. The natural join of tables *A* and *B* may be written as A ⋈ *B*, or in words as "*A* **natural join** *B*". This may be summarized as follows.

To compute *A* ⋈ *B*:
Form *A* × *B*
For each column name *c* that occurs in both *A* and *B*
  Apply the restriction *A.c* = *B.c*
  Remove *B.c*
  Rename *A.c* to *c*

There is no need to name the join columns because these are the ones with the same names in both tables (and of course these must be based on the same domain). To help remember the bow-tie "⋈" notation, note that "⋈" looks like a cross "×" with two vertical lines added, suggesting that a natural join is a Cartesian product plus two other operations (selection of rows with equal values for the common columns, followed by projection to delete redundant columns). Figure 12.21 shows the natural join for our driving example. Note that the empNr column appears just once and is unqualified.

If the tables have no column names in common, then the natural join is simply the Cartesian product. Like the Cartesian product, natural join is associative: (*A* ⋈ *B*) ⋈ *C* = A ⋈ (*B* ⋈ *C*). So expressions of the form *A* ⋈ *B* ⋈ *C* are unambiguous.

Tables being joined may have zero, one, or more common columns. In any case, the natural join is restricted to those rows where all the common attributes have the same value in both tables. The number of columns in *A* ⋈ *B* equals the sum of the number of columns in *A* and *B*, minus the number of columns common to both.

To illustrate a natural join on many attributes, consider the UoD in Figure 12.22. First examine the conceptual schema. Within a bank branch, an account may be identified by its local account number. But different accounts in different branches may have the same local account number. So, globally, the bank identifies accounts by a combination of branch and account number. Clients are identified by a global client number and have a name (not necessarily unique). Not all clients need an account. The relational schema is included in Figure 12.22.

Employee ⋈ Drives ⇨

empNr	*empName*	*gender*	*carRegNr*
001	Hagar, T	M	ABC123
003	Jones, E	F	ABC123
003	Jones, E	F	TAH007

Figure 12.21 A natural join.

Figure 12.22 Account has a composite identification scheme.

Notice the subset and equality constraints between the tables shown in Figure 12.22. Joins are usually, although not always, made across such constraint links. Simple joins may be performed on clientNr, and composite joins on branchNr–accountNr pairs (which identify accounts). Figure 12.23 shows sample data.

The two queries in Figure 12.24 use natural joins to display users and balances of accounts. The first query matches accounts by joining on both branchNr and accountNr. To add client names, the second query also matches clients by joining on clientNr.

In rare cases, comparison operators other than equality are used in joins. As a simple example, consider the Drinker and Smoker tables in Figure 12.25. These might result from a decision to map subtypes of Patient to separate tables.

Suppose we wanted a list of drinker, smoker pairs, where the drinker and smoker are distinct persons. This can be formulated by a <>*-join* as shown. Here the comparison operator is "<>". Notice that because drinkers and smokers overlap, some patient doubles may appear twice (in different order). Similarly, <-joins, >-joins , <=-joins and >=-joins may be defined.

Account:

branchNr	*accountNr*	*balance*
10	54	3000.00
10	77	500.55
23	54	1000.00

AcUser:

branchNr	*accountNr*	*clientNr*
10	54	1001
10	54	1002
10	77	2013
23	54	7654

Client:

clientNr	*clientName*
1001	Jones, ME
1002	Jones, TA
2013	Jones, ME
7654	Seldon, H
8005	Shankara, TA

Figure 12.23 Sample population for the bank schema.

(a) Account ⋈ AcUser

⇨

branchNr	*accountNr*	*balance*	*clientNr*
10	54	3000.00	1001
10	54	3000.00	1002
10	77	500.55	2013
23	54	1000.00	7654

(b) Account ⋈ AcUser ⋈ Client

⇨

branchNr	*accountNr*	*balance*	*clientNr*	*clientName*
10	54	3000.00	1001	Jones, ME
10	54	3000.00	1002	Jones, TA
10	77	500.55	2013	Jones, ME
23	54	1000.00	7654	Seldon, H

Figure 12.24 Two queries using natural joins.

Drinker:

patient	*liver*
Bloggs, F	OK
Smith, S	poor
Stoned, IM	bad

Smoker:

patient	*lungs*
Bloggs, F	bad
Stoned, IM	bad

(Drinker × Smoker)
where Drinker.patient <> Smoker.patient
*[*Drinker.patient, Smoker.patient*]* ⇨

Drinker.patient	*Smoker.patient*
Bloggs, F	Stoned, IM
Smith, S	Bloggs, F
Smith, S	Stoned, IM
Stoned, IM	Bloggs, F

Figure 12.25 Example of a <>-join.

The kinds of joins discussed so far are collectively known as *inner joins*. We can also have *outer joins*, which are used to include various cases with null values. An outer join is basically an inner join, with extra rows padded with nulls when the join condition is not satisfied. For example, given the data population shown in Figure 12.23, the result of Client **left outer join** AcUser includes a row to indicate that client 8005 has the name "Shankara TA" but uses no account (branchNr and accountNr are null on this row). The left outer join includes all the clients from the left-hand table (i.e., the table on the left of the join operator), whether or not they are listed in the right-hand table. Outer joins are discussed in detail later for SQL.

Relational Division

The final relational operation we consider is relational **division**. Division of table *A* by table *B* is only meaningful if *A* has more columns than *B*. Let's assume that table *A* has two sets of columns, *X* and *Y*, and table *B* has a set of columns *Y*. In other words, *X* is the set of columns in *A* that are not present in *B*. We'll assume an ordered relation between the *Y* columns in *A* and *B* so that we know which column in *B* corresponds to which column in *A*.

The result of $A \div B$ is formed by restricting the result to those rows of *A* where the *Y* column values in *A* *all* the rows of *B* and then deleting the *Y* columns from *A*. The expression $A \div B$ may also be written as *A* **divide-by** *B*. Figure 12.26 shows a trivial example, where the attribute domains are denoted by *X1*, *X2*, *Y1*.

Although not used very often, the division operation can be useful in listing rows that are associated with at least *all* rows of another table expression (e.g., who can supply all the items on our stock list?).

As a practical example, suppose the table *Speaks*(country, language) stores facts about which countries speak (i.e., use) which languages. A sample population is shown in the large table within Figure 12.27. Now consider the query: Which countries speak *all* the languages spoken in Canada? To answer this, we first find all the languages spoken in Canada, using the expression Speaks **where** country = 'Canada' *[*language*]*. This returns the table {English, French}. We then divide the Speaks table by this result, as shown, to obtain the final answer {Canada, Dominica}.

Figure 12.26 A simple example used to explain relational division.

Figure 12.27 A practical example of relational division.

Query Strategies

That completes all the main operators in relational algebra. Table 12.2 indicates the operator precedence adopted in this book. The comparison operators have top (and equal) priority, so are evaluated first. Next the logical operators are evaluated (first **not**, then **and**, then **or**). Then relational selection and projection are evaluated (equal priority). Finally the other six relational operators (union, intersection, difference, Cartesian product, natural join, and division) are evaluated (equal priority). Operators with equal priority are evaluated left to right. Expressions in parentheses are evaluated first. Some systems adopt a different priority convention for the relational operators.

To help formulate queries in relational algebra, the following *query strategies* are useful:

- Phrase the query in natural language, and understand what it means.
- Ask which tables hold the information.
- If you have table data, answer the query yourself and then retrace your mental steps.
- Divide the query up into steps or subproblems.
- If the columns to be listed are in different tables, declare joins between the tables.
- If you need to relate two rows of the same table, use an alias to perform a self-join.

Our first move is to formulate the query in natural language, ensuring that we understand what the query means. Next determine what tables hold the information needed to answer our query. The information might be in a single table or be spread over two or more tables. If the columns to be listed come from different tables (or different copies of the same table), then we must normally specify joins between these tables. These joins might be of any kind: inner or outer joins, natural or theta joins, or cross joins (Cartesian products).

Apart from some cases involving ∪, ∩, or −, relational algebra requires joins whenever different tables must be accessed to determine the result, even if the result columns come from the same table. SQL introduced subqueries, which can be used instead of joins when the result columns come from the same table.

Table 12.2 Operator priority (1 = highest).

Priority	*Operator type*	*Operator(s)*
1	Comparison	=, <>, <, >, <=, >=
2	Logical	**not**
3		**and**
4		**or**
5	Relational	selection (… **where** …), projection …*[*a, b*]*
6		∩ ∪ − × ⋈ ÷

Let's look at an example using our bank account database. To reduce page turning, the database is reproduced in Figure 12.28. Now consider the query: Which clients have an account with a balance of more than $700? Before formulating this in relational algebra, we should ensure that we understand what the natural language query means. In some cases we may need to clarify the meaning by asking the person who originally posed the query. For example, do we want just the client number for the relevant clients or do we want their name as well? Let's assume that we want the client name as well. Looking at the tables, we can now see that all three tables are needed. The Account table holds the balances, the Client table holds the client names, and the AcUser table indicates who uses what account.

Since we have sample data, we can first try to answer the English query, examine what we did, and finally try to express this in the formal language of relational algebra. If we were answering the request ourselves, we might go to the Account table and select just the rows where the balance exceeded $700. This yields the account (10, 54) and the account (23, 54). We might then look at the AcUser table to see who uses these accounts. This yields the clients numbered 1001, 1002, and 7654. Now we look to the Client table to find out their names ('Jones, ME', 'Jones, TA', 'Seldon, H'). So our answer is {(1001, 'Jones, ME'), (1002, 'Jones, TA'), (7654, 'Seldon, H')}.

Retracing and generalizing our steps, we see that we joined the Account and AcUser tables by matching the account (branchNr and accountNr), and we linked with the Client table by joining on clientNr. Since these attributes have the same names over the tables, we can use natural joins. We may set this out in relational algebra as

(Account **where** balance > 700) *[* branchNr, accountNr *]*
⋈ AcUser
⋈ Client *[* clientNr, clientName *]*

Account:

branchNr	*accountNr*	*balance*
10	54	3000.00
10	77	500.55
23	54	1000.00

AcUser:

branchNr	*accountNr*	*clientNr*
10	54	1001
10	54	1002
10	77	2013
23	54	7654

Client:

clientNr	*clientName*
1001	Jones, ME
1002	Jones, TA
2013	Jones, ME
7654	Seldon, H
8005	Shankara, TA

Which clients have an account with a balance of more than $700?

Figure 12.28 How can this query be formulated in relational algebra?

This isn't the only way we could express the query. For instance, using a top-down approach, both joins could have been done before selecting or projecting, thus:

(Account ⋈ AcUser ⋈ Client)
where balance > 700
[clientNr, clientName *]*

Notice how queries may be spread over many lines to make them more readable. A useful syntax check is to ensure that you have the same number of opening and closing brackets. Although these two queries are logically equivalent, if executed in the evaluation order shown, the second query is less efficient because it involves a larger join. Relational algebra can be used to specify transformation rules between equivalent queries to obtain an optimally efficient, or at least a more efficient, formulation. SQL database systems include a query optimizer to translate queries into an optimized form before executing them. So in practice we often formulate queries in ways that are easy for us to think about them rather than worrying about efficiency considerations.

Since relational algebra systems are not used in practice, you can ignore efficiency considerations when doing exercises in the algebra. However, for complex queries in SQL, practical optimizers are not perfect, and you may need to tune your queries to achieve the desired performance. In some cases, hand tuning a complex query can reduce its execution time dramatically (e.g., from hours to minutes).

As our next example, let's return to the Speaks table mentioned earlier. Consider the query: list each pair of countries that share a common language. For example, one pair is Australia and Canada since they both have English as an official language. Before looking at the solution provided, you might like to try solving this yourself. As a hint, *if you need to relate different rows of the same table, then a self-join is needed*. Also, recall that table aliases are required to join a table to itself.

The solution is shown in Figure 12.29. First we define two aliases, "Speaks1" and "Speaks2" for the Speaks table. You can think of this as two different copies of the table, as shown. By using the original table name we could get by with only one alias. Check to make sure that you understand the solution. Notice that we should not use the natural join for this query (why?).

The query of Figure 12.29 could have used "$>$" or "$<>$" instead of "$<$". However, "$<$" nicely arranges for the first name of each pair to be alphabetically prior to the second name. Moreover, "$<>$" is inadvisable since it would result in each pair being listed twice, once for each order (recall our earlier example about drinkers and smokers).

For a couple of more difficult examples, we return to the compact disc retailer UoD discussed earlier in the book. The relational schema for the base tables is set out in Figure 12.30.

As an exercise, try to formulate the following English queries in relational algebra before checking the solutions provided. These queries are much harder than our earlier examples. Recall that the month codes for January and February are "Jan" and "Feb".

Speaks1:

country	*language*
Australia	English
Belgium	Dutch
Belgium	French
Canada	English
Canada	French
Cuba	Spanish
Dominica	English
Dominica	French

Speaks2:

country	*language*
Australia	English
Belgium	Dutch
Belgium	French
Canada	English
Canada	French
Cuba	Spanish
Dominica	English
Dominica	French

define alias Speaks1 **for** Speaks
define alias Speaks2 **for** Speaks

(Speaks1 × Speaks2)
where Speaks1.language = Speaks2.language
  and Speaks1.country < Speaks2.country
*[*Speaks1.country, Speaks2.country*]*

⇨

Speaks1.country	*Speaks2.country*
Australia	Canada
Australia	Dominica
Belgium	Canada
Belgium	Dominica
Canada	Dominica

Figure 12.29 A self-join is needed to list pairs of countries with a common language.

Figure 12.30 Base table schema for compact disc retailer UoD.

(a) List the CD number and name of each compact disc that *either* had sales of more than 20 copies in each of the months January and February *or* has no track with a duration longer than 300 seconds.

(b) Who sings a track lasting at least 250 seconds *and* sings on *each* compact disc that sold more copies in February than its current stock quantity?

The relational algebra query for (a) is shown in QA. Intersection is required since the quantity sold must be greater than 20 in *each* month (i.e., January *and* February). As an exercise, explain why this can't be done using "**and**" or "**or**". Union is used for the *or* since the disjuncts are not available on the same row. Subtraction is used to enforce the condition about *no* track. The join provides the cdName.

(QA) (Sales **where** monthCode = 'Jan' **and** qtySold > 20 *[* cdNr *]*
∩
Sales **where** monthCode = 'Feb' **and** qtySold > 20 *[* cdNr *]*
∪
(Track *[* cdNr *]* − Track **where** duration > 300 *[* cdNr *]*)
⋈
CD) *[* cdNr, cdName *]*

Query QB formulates the query for (b). Intersection handles the *and* operation, while relational division is used to enforce the *each* requirement.

(QB) (Vocals ⋈ Track) **where** duration >= 250 *[* singer *]*
∩
(Vocals *[* singer, cdNr *]*
÷
(Sales ⋈ CD) **where** monthCode = 'Feb' **and** qtySold > stockQty *[* cdNr *]*)

Such queries are best formulated by noting the overall structure (e.g., a union) and then working on each part. These queries are unusual in requiring frequent use of ∪, ∩, and − . Remember in using these operators to ensure that the operands are compatible—this usually means a projection has to be done first.

Sometimes a query requires several tables to be joined. In this case, if you are joining *n* tables, remember to specify *n − 1* joins. For example, the query for Figure 12.28 joined three tables and hence required two joins.

Of the eight table operations covered, only five are primitive (i.e., cannot be defined in terms of the other operations). While some choice exists as to which to consider primitive, the following list is usual: ∪; −; ×; selection; and projection. The other three (∩, ⋈, and ÷) may be defined in terms of these five operations (the proof is left as an exercise). Talking about exercises, you must be bursting to try out your relational algebra skill on some questions. So here they are.

Exercise 12.1

1. The conceptual schema for a particular UoD is shown here. "Computer" is used as an abbreviation for "kind of computer".

(a) Map this onto a relational schema. Then populate the tables with these data:

Ann uses a P2.
Fred uses a P2 and a Mac, and owns a P2.
Sue uses a P3, and owns a P1.
Tom owns a Mac.

Given this schema and database, formulate the following queries in relational algebra and state the result. Make use of $\cup$, $\cap$, $-$, and *[ ]*.

(b) List the students who own or use a computer.
(c) List the students who use a computer without owning one of that kind.
(d) List the students who use a computer but own no computer.
(e) List the students who own a computer but do not use a computer.
(f) List the students who own a computer without using one of that kind.
(g) List the students who use a computer and own a computer of that kind.
(h) List the students who use a computer and own a computer.
(i) List the computers used (by some student) but owned by no student.
(j) List the computers owned (by some student) but used by no student.

2. (a) If A is a 200×10 table and B is a 300×10 table, under what conditions (ifany) are the following defined?

(i) $A \cup B$ (ii) $A \cap B$ (iii) $A - B$ (iv) $A \times B$

(b) If A is 200×10 and B is 100×5, what is the size of $A \times B$?

3. The relational schema and a sample population for a student database are shown here. Students are identified by their student number. It is possible for two students to have the same name. All students have their name, degree, gender, and birth year recorded, and optionally subjects (e.g., a student might enroll in a degree before picking which subjects to study). Subjects are identified by their codes. Two subjects may have the same title. For each subject, we record the title and the credit. Some subjects might not be studied (e.g., PY205 might be a newly approved subject to be introduced in the next year). This schema applies to a one-semester period only, so we can ignore the possibility of student repeating a subject. Subject enrollments are entered early in the semester, and ratings are assigned at the end of semester (so rating is optional). Formulate each of the following queries as a single relational algebra query.

(a) List the code, title, and credit for subject CS113.
(b) List the student number, name, and degree of male students born after 1960.
(c) List the codes of the subjects studied by the student(s) named "Brown T".
(d) List the studentNr and name of those students who obtain a 7 rating in at least one subject.
(e) List the studentNr and name of all students who obtain a 5 in a subject called Logic.
(f) List the studentNr and degree of those students who study all the subjects listed in the database.
(g) List the studentNr, name, and gender of those students who either are enrolled in a BSc or have obtained a rating of 7 for PD102.
(h) List the studentNr, name, and birth year for male students born before 1970 who obtained at least a 5 in a subject titled "Databases".

Student (studentNr, studentName, degree, gender, birthYr)

Result (studentNr, subjCode, [rating])

Subject (subjCode, title, credit)

Student:

StudentNr	*StudentName*	*degree*	*gender*	*birthyear*
861	Smith J	BSc	M	1967
862	Jones E	BA	F	1965
863	Brown T	BSc	M	1950

Subject:

subjCode	*title*	*credit*
CS113	Databases	8
PD103	Logic	10
PY205	Xenoglossy	5

Result:

StudentNr	*subjCode*	*rating*
861	CS113	7
861	PD102	5
862	PD102	7
863	CS113	4
863	PD102	5

4. The following table contains details on students who are to debate various religious topics. For this UoD, students are identified by their first name. Each debating team is to comprise exactly two members of the same religion but opposite gender.
Phrase each of the following as a single relational algebra query.

Debater:

firstname	*gender*	*religion*
Anne	F	Buddhist
Betty	F	Christian
Cathy	F	Hindu
David	M	Christian
Ernie	M	Buddhist
Fred	M	Hindu
Gina	F	Christian
Harry	M	Buddhist
Ian	M	Christian
Jane	F	Christian
Kim	F	Hindu

(a) List the name and religion of all females who are not Buddhist.
(b) List the name and gender of those who are either male Hindus or female Christians.

(c) List all possible debating teams, mentioning females before males. For example, one team comprises Anne and Ernie.

5. (a) Map the following conceptual schema to a relational schema.

Use your schema to formulate each of the following in relational algebra.

(b) Find names and salaries of all female employees who earn more than $25 000 or work on project "5GIS".

(c) List the name and gender of those employees who work on all projects with a budget of at least $100 000.

6. The Employee table stores the employee number and name of each employee, the department they work for, and the year they joined the firm. The Department table indicates the manager and budget for each department. Each employee manages at most one department (which must be the department for which he or she works).

Employee (empNr, empName, deptName, startYr)

Department (manager, deptName, budget)

(a) Draw a conceptual schema diagram for this UoD.

Phrase each of the following requests as a single relational algebra query.

(b) Who works for the Accounting department and started with the firm before 1970?

(c) What is the budget of the department in which employee 133 works?

(d) List the departmental managers and the year they joined the firm.

(e) Which employees are not departmental managers?

(f) Give the empNr, name, and year started for those managers of departments with a budget in excess of $50 000.

(g) Which employees worked for the firm longer than their departmental managers?

7. (a) Define $A \cap B$ in terms of $-$.

(b) Let relation A have attributes x, y and relation B have attributes y, z where both y attributes are defined over the same domain. Define the natural inner join operation $A \bowtie B$ in terms of $\times$, selection, and projection.

(c) Let A have attributes x, y and B have the attribute y where both y attributes are defined over the same domain. Define $A \div B$ in terms of $\times$, -, and projection.

8. In the following schema, a person is identified by the combining surname and forename. Formulate each of the following as single queries in relational algebra.

Person (surname, forename, gender, weight)

Plays (surname, forename, sport)

(a) Which females play judo?
(b) Which males in the weight range 70..80 kg play either judo or karatedo?
(c) Which females over 50 kg play judo but not karatedo?
(d) Who plays both judo and aikido? (Do not use a join.)
(e) Who plays both judo and aikido? (Do use a join.)

9. The following relational schema relates to the software retailer UoD from Exercise 6.3. The category codes are: DB = Database; SS = Spreadsheet; WP = Word Processor.

Customer (customerNr, customerName, address, [phoneNr])

Invoice (invoiceNr, customerNr, issueDate, [paidDate])

LineItem (invoiceNr, itemCode, qty, unitPrice)

{DB,SS,WP}

Item (itemCode, title, category, stock, listPrice)

Formulate the following as single queries in relational algebra.
(a) List the customerNr of each customer who has been charged less than the list price for at least one software item, but has been sold no word processor.
(b) List the name of each customer who was sold at least one copy of all the spreadsheets on the stock list.
(c) List the customerNr of each customer who has purchased a copy of all the word processors that are in stock but who has never purchased a database.
(d) List the customerNr of those customers who purchased both a spreadsheet and word processor on the same invoice.

12.2 Relational Database Systems

We may now define a relational DBMS as a DBMS that has the relational table as its only essential data structure and supports the selection, projection, and join operations without needing specification of physical access paths. A relational system that supports all eight table operations of the relational algebra is said to be relationally complete. This doesn't entail eight distinct operators for these tasks; rather, the eight operations must be expressible in terms of the table operations provided by the system.

The two main relational languages are SQL and QBE, with SQL being the most important. Most SQL systems are relationally complete. A system that supports all aspects

of the relational model, including domains and the two basic integrity rules (entity integrity and referential integrity), is sometimes said to be "fully relational". The relational model of data itself is evolving.

For Version 1 of the relational model, Codd proposed 12 basic rules to be satisfied by a relational DBMS. These may be summarized as follows:

1. All information is represented in relational tables.
2. All data is accessible using the table name, primary key value, and column name.
3. Systematic support for missing information (null values, independent of data type) must be provided.
4. The relational schema itself is represented in tables and is accessible in the same way as the database.
5. A relational query language must be provided that supports data definition, view definition, data manipulation, integrity constraints, authorization, and transaction boundaries (begin, commit, and rollback).
6. Basic support for view updatability is provided.
7. Table-at-a-time retrieval and update operations are provided.
8. Application programs are logically unaffected by changes to internal storage or access methods.
9. Application programs are logically unaffected by information-preserving changes to the base tables.
10. Integrity constraints must be definable in the query language and storable in the system tables.
11. The DBMS has distribution independence;
12. If record-at-a-time processing is supported, this cannot be used to bypass the constraints declared in the set-oriented query language.

Version 2 of the relational model as proposed by Codd (1990) includes 333 rules, including support for two kinds of missing information. While some of the proposed revisions have merit, others are debatable, and it is doubtful whether any commercial DBMS will ever try to satisfy all these rules. In practice, the SQL language itself has become more influential than the relational model in standardization efforts for relational DBMSs. Some SQL systems are fully relational with respect to Codd's original 12 rules, but most provide only weak support for domains. For example, many require heights and weights to be defined as numeric data types, allowing comparisons such as "height = weight".

In SQL-89 the union operation is allowed only between **select** statements, not between table expressions, and separate operators for $\cap$, $-$ and $\bowtie$ (natural join) are not explicitly provided (though expressible in terms of other SQL primitives). In 1992 the SQL standard was improved substantially, and all the table operators are explicitly included in SQL-92 (for example, $\cap$, $-$, and $\bowtie$ are called "**intersect**", "**except**", and "**natural join**"). The ability to declare constraints was also improved substantially.

The next version of the SQL standard, SQL:1999, was approved in 1999, but some of its features are still not supported in commercial systems. Features added in

SQL:1999 include triggers, user-defined data types, object identifiers, array-valued fields, recursive union, and procedural control structures.

Relational DBMSs provided higher level query facilities than pre-relational DBMSs but were originally slower. So they were first used for decision support and low-volume online applications. Nowadays their performance has improved greatly and they are dominant even for large-scale OLTP (On-Line Transaction Processing) applications.

SQL products (e.g., DB2, Microsoft SQL Server, Oracle) dominate on larger relational systems. Initially, many smaller relational systems (e.g., Access, Paradox, FoxPro) used a version of QBE or Xbase as their main query language. Nowadays, most relational systems, small or large, provide good support for SQL, with basic QBE support offered as an alternative for simpler queries.

In addition to relational algebra operations, relational DBMSs provide further capabilities, such as sorting, arithmetic, grouping, and formatting. Most systems also provide powerful tools for creating external interfaces (e.g., screen forms), report writing, and security. Despite such productivity benefits, the success of any database application still depends critically on the database design.

12.3 SQL: Historical and Structural Overview

The rest of the chapter focuses on SQL. A full coverage would require a large book itself, so many advanced features of the language are omitted. The treatment assumes familiarity with relational schemas and the main operations of relational algebra, as discussed earlier. This section provides a brief history and structural overview of SQL.

After the publication of Dr. Codd's classic paper on the relational model of data (Codd 1970), some early prototypes were developed to provide a relational DBMS, including a language for querying and updating relational databases. In 1974, Don Chamberlin and Raymond Boyce published a paper on a language called "SEQUEL" (Structured English Query Language) being implemented by a team at the IBM San Jose Research Laboratory as an interface to its System R relational prototype, within a project also called System R. By 1977, a revised version of this language (SEQUEL/2) had been defined and largely implemented by IBM. In the late 1970s it was discovered that "Sequel" was an existing trademark, so the language was renamed "SQL". Although "SQL" is often pronounced "Sequel", officially it is pronounced simply "ess-cue-ell".

The System R project ran from 1971 through 1979 and later evolved into a distributed database project (System R*). Using the experience gained from its System R project, IBM built its first commercial relational DBMS, known as SQL/DS, which it released in 1981. Its second and highly influential SQL product, known as DB2, was released in 1983. As the SQL language had been widely publicized in the 1970s in research papers, other firms began developing their own systems. Relational Software Inc. (later renamed Oracle Corporation) actually beat IBM to the market by releasing its commercial SQL product in 1979. In 1982, the American National Standards Institute (ANSI) began standardization work on the language. The first SQL standard was completed in 1986 and is known as SQL-86.

In 1987, the ANSI SQL-86 standard was adopted by the International Organization for Standardization (ISO). In 1989 a more comprehensive standard, SQL-89, was adopted by both ANSI and ISO. This defined a basic Level 1 version, a richer Level 2 version, and included an addendum on integrity enhancements (including declarative entity integrity and referential integrity).

In 1992 the next standard, known as SQL-92, was approved by both ANSI and ISO. Because of its size (over 600 pages) it was defined at three levels: entry, intermediate, and full. SQL:1999, was approved on December 8, 1999. SQL:1999 represented a significant departure from earlier versions in its inclusion of object-oriented features (e.g., object identifiers and references, array-valued fields, and procedural control structures). SQL is now object-relational rather than purely relational. Because of its size, the SQL:1999 standard (ANSI 1999) was divided into five separate parts: Framework; Foundation; Call Level Interface; Persistent Stored Modules; and Host Language Bindings. The Foundation part alone comprises 1151 pages.

The next version of the standard, SQL:2003, appeared in 2003. A significant addition was the first appearance of support for XML, but enhancements to many other areas were also included. By this time the standard had grown to nine parts, with the Foundation part alone weighing in at over 1300 pages. During the development of SQL:2003, the world of XML had also been evolving. In order to reflect the latest XML concepts, the part of the SQL standard dealing with XML alone (sometimes referred to as "SQL/XML") was revised in 2006, with other parts remaining at their 2003 revisions. A new version of the SQL standard is scheduled to appear in 2008.

Currently, most commercial DBMSs have implemented significant portions of SQL-92 and many of the features added in SQL:1999 (e.g., triggers, row types, and large objects), and later. However, it is highly unlikely that any commercial system will ever implement all of SQL-92, much less the later standards. Commercial dialects use their own, non standard syntax for some features that they do implement from the standard, and also have their own extensions that are not included in the standard. This overlap is portrayed in Figure 12.31, where dialects A and B denote commercial versions of SQL (e.g., DB2 and Microsoft SQL Server).

Commercial SQL systems are slowly replacing much of their non standard syntax with standard syntax, while retaining most of their alternative syntax for backward compatibility. When a feature is supported in both standard and non standard syntax, you should use the standard syntax. This makes your SQL code more portable and more understandable to users of other SQLs.

Figure 12.31 Proper overlap between standard SQL and commercial dialects.

This chapter focuses on the SQL standard versions, but also includes some details about commercial implementations.

The SQL language is vast. To help come to terms with it, SQL is sometimes informally classified into the following sublanguages: *DDL* (data definition language); *DML* (data manipulation language); *DCL* (data control language); and *DPL* (data procedural language). The DDL includes statements for creating, altering, and dropping various kinds of database objects, such as tables and views (e.g., **create table**, **alter table**, **drop table**, **create view**, **drop view**). The DML includes statements for querying and updating the database (e.g., **select**, **insert**, **update**, and **delete** statements). The DCL is used for security, controlling who has what kind of access to database objects (e.g., **grant** and **revoke** statements). The three previous sublanguages are essentially declarative, in that statements are used to declare what is to be done rather than how it is to be done. However, the DPL is a procedural language that supports branching, looping, and other programming constructs (e.g., **if** and **while** statements). This classification is only approximate (e.g., a **create view** statement includes a **select** query, and a **create trigger** statement may include procedural code). Our main focus is on the DML (in this chapter) and the DDL (in the following chapter).

12.4 SQL: Identifiers and Data Types

For a given schema, some database objects such as base tables, views, domains, and constraints are identified by name. A column is identified by appending its (local) name to the name of its table, using a "dot notation". This distinguishes columns with the same local name in different tables. For example, the second columns of the tables *Subject* (subjectCode, title, credit) and *Book* (isbn, title) are identified as "Subject.title" and "Book.title", respectively. In SQL, all names (including local names) are called *identifiers*, even though local names provide identification only within a limited context.

In SQL-89, identifiers were restricted to at most 18 characters, and all letters had to be in uppercase. From SQL-92 onward, an identifier is either regular or delimited. A *regular identifier* is a string of at most 128 characters, the first of which must be a letter ("a"..."z","A",..."Z"). Each later character must be a letter, digit ("0"..."9"), or underscore ("_"). Moreover, *no reserved word may be used as a regular identifier*. You may, however, use a reserved word as part of an identifier. For example, "note" is fine as an identifier even though it includes the reserved word "**not**".

Commercial SQLs may require shorter identifiers, or allow some other characters. In Oracle, the length of identifiers is restricted to 30 characters. Microsoft SQL Server allows identifiers of up to 128 characters, but also allows the first character to be "_", "#", or "@", and later characters to be "#", "@", or "$". It uses "#" to start names of temporary objects and "@" to start variable names.

Some examples are shown in Table 12.3. Names in the third row cannot be used as regular identifiers since they start with a digit or include an illegal character (e.g., a space or parenthesis). Names in the fourth to sixth rows were first introduced as reserved words in SQL-92, SQL:1999, and SQL:2003, respectively.

Table 12.3 Which names may be used as regular identifiers?

Allowed as (regular) identifiers?	*SQL-89*	*SQL-92*	*SQL:1999*	*SQL:2003*
A, R2D2, CUSTOMER_NR	Yes	Yes	Yes	Yes
a, This_is_a_long_identifier, CustomerNr	No	Yes	Yes	Yes
2B, CUSTOMER NR, SPEED(MPH)	No	No	No	No
date, first, level	Yes	No	No	No
before, row, trigger	Yes	Yes	No	No
multiset, xml, bigint	Yes	Yes	Yes	No
bit	Yes	No	No	Yes

Since the list of reserved words grows with each new standard, identifiers in existing applications might become illegal at a later stage. For example, Table 12.3 shows that many words became reserved for the first time in SQL-92, SQL:1999, and SQL:2003, and some words (such as **bit**) may be reserved words in some versions of the standard, but withdrawn in later versions. Different SQL dialects may omit some of these words from their reserved word lists, while adding others, which makes portability between dialects even harder. Partly to avoid such problems, the SQL-92 standard introduced delimited identifiers.

A *delimited identifier* is a string of at most 128 characters, delimited by (i.e., enclosed in) double quotes. Any character at all may be used, as well as reserved words, within the double quotes. For example, the following are legal delimited identifiers: "customer nr", "speed (km/h)", "&^%!!", "date", "group". Unlike regular identifiers, delimited identifiers are case sensitive (i.e., uppercase letters are not equated to lowercase letters). For example, the delimited identifiers "invoiceNr", "InvoiceNr", and "INVOICENR" are unequal, but the regular identifiers InvoiceNr and INVOICENR are equal.

In SQL, a string constant is delimited by single quotes. For example, 'USA' might be a value in a countryCode column. Some commercial SQLs allow string constants to be delimited by double quotes instead of single quotes, e.g., "USA". This practice should be discouraged, since it conflicts with the now standard use of double quotes for delimited identifiers. In Microsoft SQL Server, the command "**set quoted_identifer on**" ensures that double quotes are used only for delimited identifiers, forbidding their use to delimit string constants. SQL Server also allows square brackets ([…]) as an alternative to double quotes ("…").

Words that were reserved may cease to be so in later versions. For example, the names **avg**, **between, exists**, and **sum** were downgraded from reserved words to non reserved keywords in SQL:1999. Keywords are words that have predefined meanings. Reserved words are keywords that cannot be used as regular identifiers.

As mentioned earlier, a reserved word may be embedded within an identifier. Suppose a column is to store the names of tutorial groups. Since **group** is a reserved word, we cannot use this (unquoted) for the column name. However, we may add quotes to make it a delimited identifier, or include "group" in a longer regular identifier (e.g., "group", TuteGroup, GroupName).

A first impression of how a given SQL dialect compares with the standards may be gained by inspecting its list of keywords, and especially its reserved words. Where we deal with SQL in this book, keywords are usually distinguished by displaying them in bold.

Values entered in a table column belong to the *data type* declared for that column. Table 12.4 lists the standard data types for various versions of the SQL standard. Here, square brackets indicate optional components. All SQL dialects support at least character string and numeric types.

Table 12.4 Standard data types.

Standard		*Data types*	
SQL-89	introduced:	fixed-length string	**char, character** (*n*), **char** (*n*)
		exact numeric	**smallint**
			int
			numeric(*p* [,*s*])
			decimal (*p* [,*s*])
		approximate numeric	**float** [*p*]
			real
			double precision
SQL-92	introduced:	variable-length string	**varchar** (*n*), **character varying** (*n*)
		national character strings	**national character** (*n*), **national character varying** (*n*) and short forms such as **nchar**(*n*)
		bit string	**bit** (*n*) **bit varying** (*n*)
		datetime	**date** {year, month, day} **time** {hour, minute, second} **timestamp** {date and time} (time and timestamp can be with or without time zone)
		interval	year-month periods day-time periods
SQL:1999	introduced:	large object	**clob** (*n*) {character large object} **nclob** (*n*) {national character large object} **blob** (*n*) {binary large object} and various lengthier alternatives
		boolean type	**boolean**
		row type	**row** (field-definition [,…]
		collection type	data-type **array**(*n*)
		user defined type (UDT)	
		reference type	**ref** (UDT)
SQL:2003	introduced	exact numeric	**bigint**
		collection type	data-type **multiset**
		xml data	**xml**
	withdrawn		**bit** (*n*) **bit varying** (*n*)

In the standard, "**char**(*n*)" means the value is stored as a fixed length string of *n* characters. If the value has fewer than *n* characters, blanks are appended to fill out the length. If no size (*n*) is specified, this is treated as a string with only one character. A value of type **varchar**(*n*) is stored as a string of at most *n* characters. If the value is shorter, it is stored as it is, without padding it with extra blanks. A **varchar**(*n*) string also has an "end-of-string" marker, the size of which varies between different implementations. This has to be taken into account when estimating the storage space required. For short strings with little variation in length **char**(*n*) may be the preferred choice. SQL-92 allows various national character sets to be declared. The use of "**nchar**" or equivalent syntactic variations indicates that characters are selected from the designated national character set—this implies that the character encoding uses Unicode, which typically requires two bytes of storage per character instead of the one byte per character that was commonplace for encoding the standard English characters and symbols (often referred to as the *ASCII* code, from the name of the standards body).

With the **numeric** data type, the precision *p* is the maximum number of digits included in the number, and the scale *s* is the number of digits after the decimal point. For example, columns declared **numeric**(6, 2) allow values in the range -9999.99 to.+9999.99. The **decimal** type is like the numeric type except that an implementation may sometimes provide a precision greater than *p*. Many systems implement numeric and decimal as the same type. The **integer** (including **smallint** and **bigint**) data types allow integers only (no fractions). The three approximate numeric data types allow very large or very small numbers to be stored to a specified precision as a mantissa times an exponent of 10. The abbreviations **char**, **dec**, **int**, and **nchar** may be expanded to **character**, **decimal**, **integer**, and **national character**.

As set out in Table 12.4, SQL-92 provides direct support for bit strings as well as time points and time intervals. Times are local unless the **with time zone** option is specified. This option includes the offset from UTC (Universal Time, Coordinated, formerly called Greenwich Mean Time (GMT)). Various temporal operators are provided (e.g., to allow computation of intervals by subtracting one time point from another).

In SQL:1999, all of the previous data types are called *predefined types*. As Table 12.4 indicates, SQL:1999 adds four more predefined types: character large object, national character large object, binary large object, and boolean (true or false). In addition, SQL:1999 allows row types, user-defined types, *reference types*, and *collection types.* A row type is basically a sequence of fields and can be used as the basis for defining a table. A user-defined type (UDT) is identified by its name and may include a list of method specifications. If based on a single, predefined type, it is called a *distinct type*. If specified as a list of attribute definitions, it is called a *structured type*. The only collection type constructor allowed in SQL:1999 was array. The SQL:2003 standard added support for **multiset** (bags) as a collection type, large integers (**bigint**), and, significantly, **xml** as a recognized data type within SQL (we'll return to the **xml** type in the next chapter).

Currently, most commercial SQLs support all the SQL-89 data types, almost all the additional SQL-92 data types, a few of the extra SQL:1999 and SQL:2003 data types, and some additional non-standard data types (e.g., **money**). Many differences exist. For example, Oracle treats **varchar** as **char**, and uses **varchar2** to denote the standard **varchar**.

Informix implemented sets, multisets and lists years ago. SQL Server includes many extra types such as **tinyint**, **money**, **smallmoney**, and **uniqueidentifier**. The range of most standard types is usually implementation defined rather than standardized. For example, the range for **smallint** is often, but not always, –32,768..32,767.

Exercise 12.4

1. Which of the following are legal identifiers in which SQL standard, if shifted to uppercase?

(a) Payroll#	(b) PayrollNr	(c) "Payroll#"
(d) 1994Tax	(e) Tax in 1994	(f) "Tax in 1994"
(g) Tax_in_1994	(h) Deposit_in_$	(i) Mass_(kg)
(j) Order	(k) WorldWideWebIdentifier	(l) count

12.5 SQL: Choosing Columns, Rows, and Order

Recall that relational algebra includes the following eight table operations: union, intersection, difference, Cartesian product, selection, projection, join, and division. All of these operations (as well as others) can be expressed using SQL's powerful **select** statement. This section discusses how SQL is used to perform relational projection and selection, as well as bag projection and row ordering.

First let's see how to *choose columns.* Consider a small UoD where people are identified by their firstname. Table 12.5 provides sample data for the table scheme: *Person* (firstname, gender, starsign, birthyr).

The whole table may be retrieved by projecting on all its columns. In relational algebra, this may be formulated as Person or as Person *[*firstname, gender, starsign, birthyr *]*. In SQL, this is expressed as follows:

```
select * from Person
```

Here the asterisk "*" means "all columns" and may be read as "everything" or "all". The table named after **from** indicates the table from which the data is to be retrieved. When this command is executed, the result is an *unnamed* table with the same column names and contents as the original Person table.

Table 12.5 A relational table storing personal details.

Person:

firstname	*gender*	*starsign*	*birthyr*
Bob	M	Gemini	1967
Eve	F	Aquarius	1967
Fred	M	Gemini	1970
Norma	F	Aries	1950
Selena	F	Taurus	1974
Terry	M	Aquarius	1946

```
select firstname, starsign
from Person
```

⇨

firstname	*starsign*
Bob	Gemini
Eve	Aquarius
Fred	Gemini
Norma	Aries
Selena	Taurus
Terry	Aquarius

Figure 12.32 Projecting on columns that include a key.

(a)

```
select gender, starsign
from Person
```

⇨

```
select all gender, starsign
from Person
```

⇨

gender	*starsign*
M	**Gemini**
F	Aquarius
M	**Gemini**
F	Aries
F	Taurus
M	Aquarius

(b)

```
select distinct gender, starsign
from Person
```

⇨

gender	*starsign*
M	**Gemini**
F	Aquarius
F	Aries
F	Taurus
M	Aquarius

Figure 12.33 The distinct qualifier may be used to remove duplicate rows.

If only some of the columns are required, then instead of "*" the relevant columns should be listed (separated by commas). If the columns include a key, no duplicate rows can occur in the result, so the result corresponds to a relational projection on those columns (e.g., see Figure 12.32). To improve readability, an SQL query is usually spread over many lines (e.g., a new line is used to start the **from** clause). Except for delimited (quoted) identifiers or string constants, the case (upper or lower) in which letters are written has no significance. We'll use lowercase most of the time, but start table names with a capital letter. Many organizations have their own detailed standards on use of uppercase or lowercase.

If the chosen columns do not include a key, then duplicate rows may occur in the result. For example, the query in Figure 12.33(a) returns a bag (multiset) rather than a set, since ('M', 'Gemini') appears twice. This operation may be called a *bag projection*. It is sometimes useful to include duplicates (e.g., listing all the scores awarded by judges for a gymnastics event). You can explicitly request inclusion of duplicate rows by using the qualifier **all** before the **select** list, but this is what happens by default in SQL anyway. So a query of the form **select all** a, b, ... **from** T is equivalent to **select** a, b, ... **from** T.

To eliminate duplicates, include the qualifier **distinct** before the **select** list. This ensures that rows displayed will be distinct, thus providing a relational projection. Hence the **distinct** qualifier converts a bag of rows to a set of rows. For example, the query in Figure 12.33(b) eliminates the duplicate ('M', 'Gemini') row.

Columns are displayed in the order in which they are specified in the **select** list. If "*" is used, all the columns are displayed in the same order as the original table. So the command "**select** * **from** Person" is merely shorthand for "**select** firstname, gender, starsign, birthyr **from** Person".

In general, the relational projection *T[a, b,...]* of relational algebra may be expressed by an SQL statement of the form:

select distinct a, b,... **from** T

We may summarize that portion of the **select** statement covered so far in the following Extended Backus Naur Form (EBNF) notation. The symbol "::="may be read "is defined as". Here items in square brackets are optional. The expression "[,...]" means that the previous construct may be repeated any number of times, with its occurrences separated by commas. A stroke "|" is used to separate alternatives. By default, "|" has minimum scope (i.e., it applies just to the terms immediately next to it). Keywords are shown in bold. The final line of the **select** query shown here is called the **from** clause. From a logical point of view, it would be better to specify the **from** clause before the **select** list, but the syntax of the language does not allow this.

```
select-list     ::=   columnname [,...]

select-query ::=   select [all | distinct] select-list | *
                        from  tablename
```

Now let's see how to *choose rows* in SQL. A relational *selection* (or restriction) is the operation of selecting those rows that satisfy a specified condition. In SQL, the syntax for this operation is essentially the same as the relational algebra syntax used earlier in this chapter. A **where** clause is added to the **select** statement just after the **from** clause, and the specified condition is known as the *search condition*. Do not confuse the relational operation of selection with the **select** statement itself, which is used to perform other operations as well. Formulations in relational algebra and SQL listed in Table 12.6 are equivalent. For example, the query in Figure 12.34 may be used to list details about the Aquarians mentioned in our earlier Person table.

Table 12.6 Equivalent formulations in relational algebra and SQL.

Relational Algebra	*SQL*
T **where** *c*	**select** * **from** *T* **where** *c*
T **where** *c* *[a, b, ...]*	**select distinct** *a, b, ...* **from** *T* **where** *c*

```
select *
from Person
where starsign = 'Aquarius'
```

⇨

firstname	*gender*	*starsign*	*birthyr*
Eve	F	Aquarius	1967
Terry	M	Aquarius	1946

Figure 12.34 A **where** clause is used to select just the Aquarians.

Actor:

surname	*firstName*	*gender*
Bardot	Brigitte	F
D'Abot	Maryam	F
D'Abot	Tony	M

```
select *, 'This isn''t hard' as comment
from Actor
where surname - 'D''Abot'
```

⇨

surname	*firstName*	*gender*	*comment*
D'Abot	Maryam	F	This isn't hard
D'Abot	Tony	M	This isn't hard

Figure 12.35 Two single quotes embed a single quote, and "as" introduces an alias.

Notice that the character string "Aquarius" contains a mixture of uppercase and lowercase characters. Suppose this is the way it is stored in the table, but the query has the starsign typed in uppercase, that is,

```
select * from Person
where starsign = 'AQUARIUS'
```

Unless the system has been configured to convert characters strings into the same case before comparing them, no match would be made since the string "AQUARIUS" is different from "Aquarius". To make string comparisons case insensitive in SQL-92 onwards, either declare a collation set to act this way or use string functions to convert the case. For example, given a string expression *s*, the fold functions **upper**(*s*) and **lower**(*s*) return the uppercase and lowercase versions of *s*. Hence the following query will retrieve the Aquarians regardless of the case used for the letters in 'Aquarius' in the column entries:

```
select * from Person
where upper(starsign) = 'AQUARIUS'
```

Although many commercial dialects of SQL, such as SQL Server, support the standard upper and lower string functions, some dialects provide other ways of controlling case sensitivity of string constants.

The next example, shown in Figure 12.35, illustrates three new features. The sample data for the table scheme *Actor*(surname, firstname, gender) includes the surname "D'Abot", which includes a single quote as one of its characters. As discussed earlier, string constants are delimited by single quotes. To embed a single quote within a string like this,

we need to inform SQL that the quote is part of the string rather than ending it. To do this, we *use two adjacent single quotes to denote an embedded single quote.*

The following **insert** statement may be used to add the second row of data to the table: **insert into** Actor **values** ('D"Abot', 'Maryam', 'F'). Notice that this uses two single quotes in "D''Abot". Although this may display simply as "D'Abot" when queried, we must use the two single quotes when using the value in an SQL statement. For example, note the search condition: surname = 'D"Abot'.

The second thing to note about the query is that *we may include a constant as* **select** *item*. Here the **select** list includes the string constant "This isn''t hard". Note again the use of two single quotes to embed a single quote. Any constant included as an item in the **select** list will be included in each row returned by the query. In our example, two rows were returned.

The third aspect illustrated by the example is that the keyword **as** *may be used to introduce an alias for a* **select** *item*. This alias will appear as the name of the column for that item in the result table. The example introduces the alias "comment" for the string constant that was added to the **select** list. In practice, aliases for **select** items are used mainly to provide simple names for more complex expressions, as discussed later.

Search conditions may include terms, *comparison operators* (=, <>, <, >, <=, >=) and *logical operators* (**not**, **and**, **or**). Some SQL dialects allow symbols other than "<>" for "is not equal to" (e.g., "^=", "!="). SQL Server also includes "!<" for "is not less than" and "!>" for "is not greater than", which are equivalent to ">=" and "<=", respectively. For portability, you should avoid such non standard symbols. The same priority convention as for relational algebra is adopted. So unless brackets determine otherwise, *comparison operators are evaluated before logical operators,* which are evaluated in the order: **not**; then **and**; then **or**.

Consider the query in Figure 12.36. This lists the name and starsign of people born after 1950 who are either Aquarians or Geminis. If the parentheses were omitted, the condition would be interpreted as starsign = 'Aquarius' **or** (starsign = 'Gemini' **and** birthyr > 1950). This is different, since it would result in the older Aquarians (in this case Terry) being listed as well.

The selection operation becomes a bit trickier if nulls are present. Consider the relation scheme *Employee* (empNr, empName, dept, [carType]). Here the carType column is optional. Some employees might not drive a car, and even if they do perhaps it is not recorded. Figure 12.37 provides a sample population for this table, as well as two queries and their results. Suppose we want to know who drives a Ford, and who doesn't. We might formulate these questions as the queries in Figure 12.37. Although each employee in real life either does or does not drive a Ford, employees 1002 and 1005 are absent from both the query results. Can you make sense of this?

```
select firstname, starsign
from Person
where (starsign = 'Aquarius' or starsign = 'Gemini')
        and birthyr > 1950
```

⇨

firstname	*starsign*
Bob	Gemini
Eve	Aquarius
Fred	Gemini

Figure 12.36 Parentheses are needed to evaluate **or** before **and**.

Employee:

empNr	*empName*	*dept*	*carType*
1001	Thompson, E.	Sales	Ford
1002	Jones, E.	Sales	?
1003	Smith, F.	R&D	Toyota
1004	Adams, A.	Sales	Ford
1005	Dennis, A.	Admin	?

(a)
```
select empNr, empName
from Employee
where carType = 'Ford'
```
⇨

empNr	*empName*
1001	Thompson, E.
1004	Adams, A.

(b)
```
select empNr, empName
from Employee
where carType <> 'Ford'
```
⇨

empNr	*empName*
1003	Smith, F.

Figure 12.37 Rows are selected when the condition is true (not false and not unknown).

Where nulls are concerned, SQL operates on a *three-valued logic*. A condition may evaluate to true, false, or unknown. More correctly, a condition is known-to-be-true, known-to-be-false, or unknown. *A comparison between terms, one of which is null, always evaluates to unknown*. This holds for any comparison operator (=, <>, <, etc.). Consider the row for employee 1002, where a null value (displayed here as "?") is recorded for the carType. For the query in Figure 12.37(a), the condition is "null = Ford", and for the query in Figure 12.37(b), the condition is "null <> Ford". In both cases, the condition evaluates to unknown.

The selection operation performed by the **where** clause *returns just those rows that evaluate to true* (i.e., known to be true). Rows that evaluate to false or unknown are filtered out. Hence the rows for employees 1002 and 1005 (Jones and Dennis) are filtered out in both queries. Filtering out the unknown helps avoid making unwarranted assumptions. As discussed later, SQL provides a special function for detecting nulls.

Now let's discuss how to control the *ordering* of columns and rows in a result table. Columns are displayed in the order in which they appear in the **select** list. So if you want a different column order, simply reorder the **select** list. However, the order in which *rows* are displayed may be controlled by means of an **order by** clause. If used, the **order by** clause must come at the end of the **select** statement. One or more columns in the **select** list (identified by name or by position in the **select** list) may be used as ordering criteria.

Ordering for a criterion is ascending by default. You can also explicitly specify ascending order by appending **asc** to the column identifier. To specify descending order, you must append **desc**. For numeric values, "ascending" means that smaller numbers are listed first. For example, $1 < 3 < 7$, so 1, 3, and 7 are in ascending order. For character string values, "ascending" means strings that come earlier in the collating sequence are listed first. As a rough guide, words that come earlier in alphabetical order usually come first.

```
select empNr, empName
from Employee
order by empName
```

⇨

empNr	empName
1004	Adams, A.
1005	Dennis, A.
1002	Jones, E.
1003	Smith, F.
1001	Thompson, E.

```
select empNr, empName
from Employee
order by empName asc
```

⇨

Figure 12.38 Listing employees in ascending order of employee names.

As a simple example, each of the queries in Figure 12.38 may be used to list the employees in alphabetical order. Although highly undesirable, "2" may be used here instead of "empName" in the **order by** clause, since empName is the second item in the **select** list (it doesn't have to be the second column of the base table). This use of position numbers creates problems if you later modify the **select** list, makes the query harder to understand, and is a deprecated feature of the SQL standard (i.e., it may be removed in a later version). Because **as** can be used if needed to introduce a result-column name, there is never any need to resort to position numbers for ordering. So don't use them.

Character strings are ordered according to the character collating sequence. For example, if ASCII is used, then space (" ") precedes digits (0..9), which precede uppercase letters (A..Z), which precede lowercase letters (a..z). For instance, in ASCII, "M2" < "MY" < "Ma" < "Ma Kettle" < "MacTavish" < "Zen" < "apple". However, other collating sequences may differ from this. For example, in EBCDIC, lowercase letters precede uppercase letters, which precede digits.

The **order by** clause may be thought of as a way of converting a bag of rows to a sequence of rows. For ordering purposes, null values are treated as equal and, depending on the implementation, are either less than all non-null values (as in SQL Server) or greater than all non-null values. Consider the following query:

```
select carType from Employee
order by carType
```

This returns the sequence 'Ford', 'Ford', 'Toyota' either preceded or followed by the two nulls, depending on the implementation.

If the column chosen for ordering does not have a uniqueness constraint, then further ordering can be obtained within its duplicate values by specifying more columns in the order-by clause. Criteria listed first in the **order by** clause are ordered first. For any particular column, the ascending or descending option may be applied. For instance, a query of the form

```
select * from T order by a, b desc
```

first sorts the rows in ascending order of *a*, and then each set of rows with the same *a* value is sorted in descending order of *b*. For example, see Figure 12.39. Here the females are listed first, since 'F' < 'M', and within each gender the names are shown in reverse alphabetic order.

```
select firstname, gender
from Person
order by gender, firstName desc
```

⇨

firstname	*gender*
Selena	F
Norma	F
Eve	F
Terry	M
Fred	M
Bob	M

Figure 12.39 Ordering on two criteria.

```
select starsign, firstname, gender, birthyr
from Person
where starsign <> 'Taurus'
  and starsign <> 'Aries'
order by startsign, gender desc, birthyr
```

⇨

starsign	*firstname*	*gender*	*birthyr*
Aquarius	Terry	M	1946
Aquarius	Eve	F	1967
Gemini	Bob	M	1967
Gemini	Fred	M	1970

Figure 12.40 If no order is specified after an order-by item, **asc** is assumed.

Notice in Figure 12.39 that **desc** has *minimum back-scope*, applying just to firstname, not to gender. If descending order of both were required, we would use gender **desc**, firstname **desc**. If an item in an **order by** list is not qualified by either an **asc** or **desc** option, then **asc** is always assumed by default. Hence, to list in ascending order of gender and then name, we may use just gender, firstname, since this is taken to abbreviate gender **asc**, firstname **asc**. To test your understanding, see if you can predict the result of the query in Figure 12.40 and then check your answer with the table shown.

Did you get it right? The rows are first sorted on starsign in ascending order, since **asc** is assumed after starsign (see column 1). Then males of the same starsign as females are listed before those females as a second criterion (see rows 1, 2) since this is descending order of gender. Finally, for those of the same starsign and gender, the younger appear first using the third criterion of ascending order by birth year (see rows 3, 4). The **order by** clause may be specified explicitly by including the implied ordering options:

```
order by starsign asc, gender desc, birthyr asc
```

Let us use the term *colname* to indicate a column name, and *col* to indicate a column specification (either by name or number). The syntax of that portion of the **select** statement so far covered may be summarized in EBNF as follows. Although a column alias can be introduced immediately after a **select** item without using **as**, it is better to include **as** in such cases since it aids readability.

```
select * | [all | distinct] colname | constant [[as] column-alias]  [,...]
from   tablename
where condition
order by col [asc | desc]  [,...]
```

Exercise 12.5

1. The following table concerns statements provided in Modula 2. In the column "extra", the values "Y" and "N" indicate, respectively, whether the statement is extra to or already included in Pascal. Formulate each of the following requests as a single SQL query.

Statement:

kind	*composition*	*extra*
assignment	simple	N
procedure call	simple	N
if	structured	N
case	structured	N
for	structured	N
while	structured	N
repeat	structured	N
loop	structured	Y
with	structured	N
exit	simple	Y
return	simple	Y
empty	simple	N

(a) List the kind of each statement.
(b) List all information in the table.
(c) List the kind and composition of all statements.
(d) List the kind of each statement, and whether it is extra.
(e) List the possible composition values of the statements. (Avoid duplicates)
(f) Which statements are structured?
(g) Which simple statements are extra to Pascal?
(h) List the kind and composition of those statements that are extra to Pascal.
(i) List, in alphabetical order, the kinds of those statements already found in Pascal.
(j) List the kind and composition of all the statements, starting with the simple statements.
(k) List the kind and composition of all the statements, starting with the structured statements and listing statements of the same composition in alphabetic order.
(l) List the kind, composition, and extra status of all the statements, giving precedence to statements extra to Pascal. Statements with the same extra status should be listed starting with those of simple composition, with those of similar composition being shown in reverse alphabetic order.
(m) List alphabetically the kinds of statements that are neither structured nor extra to Pascal.
(n) List in reverse alphabetic order the kinds of those statements that are either structured and extra to Pascal or simple.
(o) As for (n) but exclude the exit statement from consideration and include the composition of each. Show all the structured statements first, in reverse alphabetic order, followed by the simple statements, in reverse alphabetic order.

12.6 SQL: Joins

We have seen how to choose, and order, columns and rows from a single table. But suppose the information we need from a single query is spread over many tables. SQL

provides two main methods to access such information. The first of these uses relational **joins** and is discussed in this section. The second involves subqueries and is treated in a later section. The following discussion assumes a basic familiarity with the concept of joins from the relational algebra section.

A *cross join* (Cartesian product) of tables pairs all the rows of one with all the rows of the other. In SQL-89, a cross join of tables is specified by listing the tables in the **from** clause, using a *comma* "," to denote the Cartesian product operator (depicted by "×" in relational algebra). A *conditional join* (Θ-join) selects only those rows from the Cartesian product that satisfy a specified condition. In SQL-89, the condition is specified in a **where** clause, just as we did in relational algebra. So the query expressions in Table 12.7 are equivalent.

Although the SQL-89 syntax has the advantage of being supported by every commercial dialect of SQL, it has drawbacks. First, it fails to distinguish join conditions (intertable comparisons) from nonjoin conditions (intratable comparisons on the same base row), instead lumping them together in a single **where** clause. This makes the query harder to understand. Second, it uses a comma for the cross join operation. This is unintuitive and also overloads the comma with different meanings (e.g., it is also used in SQL simply to separate items in lists). Third, it does not provide any direct support for operations such as outer joins, which can occur in practical queries.

For such reasons, SQL-92 (and later) uses special syntax for various kinds of join. In addition to supporting the SQL-89 syntax, these newer standards include special notations for the following types of join (any text after two hyphens "--" is a comment):

```
cross join
qualified join:   conditional join     -- on clause
                  column-list join     -- using clause
natural join
union join
```

Qualified and natural joins may be further classified into the following types:

```
inner                  -- this is the default
outer:    left
          right
          full
```

Table 12.7 SQL-89 syntax for joins.

Join type	*Relational algebra*	*SQL-89*
Cross join	$A \times B$	**select** * **from** *A, B*
Conditional join	$A \times B$ **where** A.a Θ B.b	**select** * **from** *A, B* **where** A.a Θ B.b

The new and old syntax for these is summarized in Table 12.8. In this summary, if tables *A* and *B* have any common columns (with the same local name), these columns are collectively referred to as "*c*". For the column-list join, "c_1, ..." denotes one or more of these common columns. Some entries for SQL-89 syntax include unions and subqueries and are explained in later sections.

Table 12.8 Joins in SQL-92 onward, compared with SQL-89.

Join type	*New syntax from SQL-92 onward*	*SQL-89 syntax*
cross	**select** * **from** A **cross join** B	**select** * **from** A, B
conditional	**select** * **from** A **join** B **on** *condition*	**select** * **from** A, B **where** *condition*
column-list	**select** * **from** A **join** B **using** (c_1, …) -- c_1, ... are unqualified	**select** A.c_1, ... , … -- omit B.c_1, ... **from** A, B **where** A.c_1 = B.c_1 **and** ... -- join columns are qualified
natural inner	**select** * **from** A **natural** [**inner**] **join** B -- join column in result is c	**select** A.c,... -- omit B.c **from** A, B **where** A.c = B.c -- join column in result is A.c
left outer	**select** * **from** A **natural left** [**outer**] **join** B -- join cols are unqualified; -- nulls generated for non-matches -- to join on fewer cols use: A **left** [**outer**] **join** B **using** (c_1, …) -- to join cols with different names: A **left** [**outer**] **join** B **on** *condition*	**select** A.c,... omit B.c **from** A, B **where** A.c = B.c **union all** **select** c, ..., ' ? ', ... **from** A **where** c **not in** (**select** c **from** B) -- for composite c, use **exists** with a -- correlated subquery. -- fewer or different cols cases -- are not shown here
right outer	**select** * **from** A **natural right** [**outer**] **join** B -- other cases: cf. left join	**select** B.c, ... -- omit A.c ... -- rest as for left join, -- but swap A and B
full outer	**select** * **from** A **natural full** [**outer**] **join** B -- other cases: cf. left join; -- rarely used	union of left and right outer joins
union	**select** * **from** A **union join** B -- rarely used	**select** A.*cols*, ' ', … -- ' ' ∀ B col **from** A **union all** **select** ' ', ..., B.*cols* -- ' ' ∀ A col **from** B

Many dialects of SQL now support the new syntax for cross joins and qualified joins (including outer joins), but few yet support the new syntax for natural joins. Let's now consider each of the new join notations.

For *cross join*, the more descriptive **cross join** may be used instead of a comma. This syntax is best reserved for unrestricted joins (full Cartesian products). As an example, Figure 12.41 shows details of Male and Female tennis players. The query lists possible male–female pairs for mixed-doubles teams. Cross joins are fairly rare in practice.

Much more common are conditional joins. The new syntax includes the join operator in the **from** clause, and the join condition in an **on** clause. Figure 12.42 shows a relational schema, a sample population, and a conditional join query to retrieve the name of each female employee and their department. To aid readability, it's a good idea to indent the **on** clause, as shown. Each column mentioned in the query occurs in just one of the tables, so there is no need to qualify column names with table names. In this case, the join takes place along the foreign key reference shown in Figure 12.42. Although joins often involve foreign key references, they don't have to. All we need is that the columns being joined are based on the same domain.

Male:

firstname	*height*
Dick	72
Necito	70
Pat	68
Scot	72

Female:

firstname	*height*
Linda	60
Mary	62

```
select Male.firstname, Female.firstname
from  Male cross join Female
```

Male.firstname	*Female.firstname*
Dick	Linda
Dick	Mary
Necito	Linda
Necito	Mary
Pat	Linda
Pat	Mary
Scot	Linda
Scot	Mary

Figure 12.41 Listing all possible Male–Female pairs.

Department (deptCode, deptname)

Employee (empNr, empName, dept, gender) {M,F}

Employee:

empNr	*empName*	*dept*	*gender*
001	Hagar T	DV	M
002	Wong S	DV	F
003	Jones E	MK	F
004	Mifune K	SL	M

Department:

deptCode	*deptName*
DV	Development
MK	Marketing
RS	Research
SL	Sales

```
select empName, deptName
from  Employee join Department
      on dept = deptCode
where gender = 'F'
```

empName	*deptName*
Wong S	Development
Jones E	Marketing

Figure 12.42 A conditional join specifies the join condition in an **on** clause.

The same query may be formulated in SQL-89 as follows:

```
select empName, deptName
from   Employee, Department
where dept = deptCode and gender = 'F'
```

However, the formulation in Figure 12.42 is better, since it separates out the join condition (in the **on** clause) from the intratable condition (in the **where** clause). Although declaring the join conditions up front is logically cleaner, in practice, SQL optimizers typically evaluate the intratable conditions before the join conditions, since this reduces the size of intermediate tables created to compute the result (why?).

Column-list joins match values in the specified columns with the local same name in both tables. The new syntax uses the word **join** in the **from** clause, with the relevant columns listed in a **using** clause. As an example, consider the following schema. This is semantically equivalent to the previous schema, but different choices are made for some column names. The department code is denoted by "deptCode" in both tables, and the local identifier "name" is used for the department name, as well as the employee name.

Department (deptCode, name)

{M,F}

Employee (empNr, name, deptCode, gender)

Now consider the previous query: list the name of each female as well as their department name. How do we formulate this query for this new schema? The tables have two local column names in common: "deptCode" and "name". Here we want to perform an equi-join between only some of the columns with common names (just "deptcode", not "name"). The common column names used for the join are listed in a **using** clause, as shown here. Only one column is used for the join. To disambiguate the query, the column names in the **select** list are qualified by their table name.

```
select Employee.name, Department.name
from   Employee join Department
    using (deptCode)
where gender = 'F'
```

As you may have realized, the column naming choices in the previous two schemas are less than ideal. If two columns in different tables are intended to signify the same thing, it is better to give the columns the same name in those tables, wherever possible. Although informal, this makes it easier to see intended semantic connections between the tables, especially those without foreign key references. For the schema under discussion, "deptCode" may be uniformly used for the department code, and the names of employees and departments may be distinguished as "empName" and "deptName", as follows:

Department (deptCode, deptname)

Employee (empNr, empName, deptCode, gender {M,F})

With this new schema, the conditional join query considered earlier needs to qualify the column names used in the join, since their unqualified names occur in both tables. This gives

```
select empName, deptName
from   Employee join Department
  on Employee.deptCode = Department.deptCode
where gender = 'F'
```

For the new schema, the column-list query does not require any qualifications to the column names, as shown here.

```
select empName, deptName
from   Employee join Department
  using (deptCode)
where gender = 'F'
```

As you might have noticed, the underlying join operation in the previous two queries is now a *natural join*, since we are equijoining on *all* the columns with the same name (in this case there is only one such column: deptCode).

As discussed in relational algebra, *natural inner joins* perform an equijoin to match all columns with the same local names and then remove the extra copies of these columns. In SQL-89, natural inner joins are specified by first forming the Cartesian product (of the tables listed in the **from** clause) and then specifying any join conditions in a **where** clause, using the **select** list to filter out unwanted duplicate columns. In SQL-92 (and later), natural inner joins may be declared simply by inserting **natural** or **natural inner** before **join** in the **from** clause (qualified and natural joins are inner by default).

The new notation has three main advantages: it is more descriptive; the **natural join** operator gives direct support for the **natural inner join** operator of relational algebra ($\bowtie$); and it is often more concise. For example, the query just considered can be reformulated more concisely as

```
select empName, deptName
from   Employee natural join Department
where gender = 'F'
```

There is no need to explicitly declare the join condition Employee.deptCode = Department.deptCode, since this is implied by the use of **natural** and the fact that "deptCode" is the only common column name for the tables listed in the **from** clause. The compactness of this new notation is more obvious as the number of natural joins in a query grows. For example, consider the following relational schema.

Suppose we want to list the employee number and name of each employee who drives, as well as the model of their cars. Looking at the schema, the information is spread over three tables. As indicated in Figure 12.43, we can join the Employee and Drives tables by matching empNr, and join with the Car table by matching carRegNr (join values are shown in bold). Using the special syntax for natural joins, this can be formulated simply as shown.

Currently, most SQL dialects (including SQL Server) do not support the natural join syntax. Instead, the conditional join syntax is often used to handle natural joins. For example, the query in Figure 12.43 may be reformulated, thus:

```
select Employee.empNr, empName, carModel
from   Employee join Drives
    on Employee.empNr = Drives.empNr
  join Car
    on Drives.carRegNr = Car.carRegNr
```

Alternatively, SQL-89 syntax can be used as follows:

```
select Employee.empNr, empName, carModel
from   Employee, Drives, Car
where Employee.empNr = Drives.empNr
  and Drives.carRegNr = Car.carRegNr
```

Employee:

empNr	*empName*	*deptCode*	*gender*
001	Hagar T	DV	M
002	Wong S	DV	F
003	Jones E	MK	F
004	Mifune K	SL	M

Drives:

empNr	*carRegNr*
001	**ABC123**
003	**ABC123**
003	**TAH007**

Car:

carRegNr	*carModel*
ABC123	Ford
PKJ900	Ford
TAH007	Saturn

```
select empNr, empName, carModel
from  Employee natural join Drives
natural join Car
```

⇨

empNr	*empName*	*carModel*
001	Hagar T	Ford
003	Jones E	Ford
003	Jones E	Saturn

Figure 12.43 A projection on the natural join of three tables.

When the natural join operator is used, any join columns in the **select** list must be denoted by their local (unqualified) names. If a natural join is specified using other syntax (e.g., conditional join or SQL-89 syntax), the join columns must be qualified.

Although most joins in practice are natural inner joins, sometimes natural joins should be avoided. The natural join operation cannot be used to join columns with different names. For example, it can't join dept and deptCode in the query of Figure 12.42. Moreover, the natural join operation must not be used if different tables in the **from** clause include columns with the same name that should not be equated. For example, consider the table schemes for our column-list join example: *Department* (deptCode, name); *Employee* (empNr, name, deptCode, gender). A natural join between Department and Employee would equate not just deptCode but name as well. Any row in the result would equate the employee's name with the department's name, which is nonsense.

Sometimes we do wish to naturally join two tables on more than one column. For example, recall the following schema discussed in the relational algebra section.

To list the balance and client details for all the accounts requires a composite join on account (compound identifier) and a simple join on client. This may be specified with the natural join syntax as:

```
select * from Account natural join AcUser natural join Client
```

or as a conditional join:

```
select  Account.branchNr, Account.accountNr, balance,
Client.clientNr, clientName
from    Account join AcUser
   on Account.branchNr = AcUser.branchNr
      and Account.accountNr = AcUser.accountNr
   join Client
   on AcUser.clientNr = Client.clientNr
```

For conditional, column-list, and natural joins, **join** is assumed to mean inner join unless an outer join is explicitly specified (see later). In these cases, **inner** may be explicitly declared (e.g., **inner join** or **natural inner join**). If **natural** is declared, an **on** clause or a **using** clause must not be. If three or more tables are included in a join expression, the joins are normally evaluated in a left-to-right order. The join order can also be controlled by inserting parentheses. Since natural inner joins are associative, their order doesn't affect the actual result.

As a more complex example, consider the relational schema in Figure 12.44. As an exercise, you might like to draw the conceptual schema. In the first table, "mgrEmpNr" denotes the employee number of the department manager (where there is a manager).

The pair-subset constraint indicates that an employee manages a department only if he or she works in it. The (2, 1) item-list marker on this constraint is used here to indicate the reordering of the pair of values to (mgrEmpNr, deptCode) before comparing with (empNr, deptCode).

Now consider the following query: for each department with a manager, list its code and name, and the employee number and name of its manager. We must match employee numbers between Department and Employee tables. The department codes must also match, but do we need to specify this? No, because if the manager referenced in the department table is the same as the employee referenced in the Employee table, the matching of department codes is implied by the constraints in the schema (why?). So the query may be formulated with the following conditional join:

```
select Department.deptCode, deptName, mgrEmpNr, empName
from   Department join Employee
  on mgrEmpNr = empNr
```

Can we do this with a natural join? No, since the join columns have different names ("mgrEmpNr" and "empNr").

So far we have discussed four of the eight joins listed in Table 12.8. There are also three outer joins, as well as a union join. The union join is rarely used and is not discussed further. *Outer joins* may be *left*, *right*, or *full*. Left and right outer joins are often encountered in commercial applications. Given two tables *A* and *B*, their left/right/full outer join is formed by first computing their inner join and then adding the rows from the left/right/both table(s) that don't have a match in the inner join and padding them with nulls to fill the extra columns in the result table.

Figure 12.45 provides a simple example of a natural left outer join. The query lists the employee number, name, and car registration numbers (if any) of all the employees (including those who don't drive cars). Each of the two formulations shown is legal in SQL-92 (and later). The first uses the natural join syntax, but this is not yet supported by most SQL dialects. The second formulation uses the conditional outer join syntax, which is typically supported.

Figure 12.44 A relational schema about employees.

Employee:

empNr	*empName*	*deptCode*	*gender*
001	Hagar T	DV	M
002	Wong S	DV	F
003	Jones E	MK	F
004	Mifune K	SL	M

Drives:

empNr	*carRegNr*
001	ABC123
003	ABC123
003	TAH007

```
select empNr, empName, carRegNr
from  Employee natural left outer join Drives
```

⇨

```
select Employee.empNr, empName, carRegNr
from  Employee left outer join Drives
      on Employee.empNr = Drives. empNr
```

⇨

empNr	*empName*	*carRegNr*
001	Hagar T	ABC123
002	Wong S	?
003	Jones E	ABC123
003	Jones E	TAH007
004	Mifune K	?

Figure 12.45 Two equivalent formulations of a natural left outer join.

The natural inner join results in the three rows shown for the two drivers (employees 001 and 003). Employees 002 and 004 have no match in the Drives table. The left outer join adds their rows padded with a null value for car registration number. In this book, a null value is displayed as "?". Some systems display it instead as a blank or as "NULL". Many systems allow you to choose how you would like nulls to display.

Although the conditional join syntax requires qualified column names for join columns, when the result table is displayed, some systems (e.g., SQL Server) are clever enough to omit the table qualification from the column header (as in Figure 12.45). Some systems support an alternative outer join syntax that was introduced before outer joins became part of the SQL-92 standard. This alternative syntax often has different semantics and should never be used unless the standard syntax is unsupported.

SQL-92 (and later) are *orthogonal*. Anywhere that a value is legal, so is any expression that returns that value. This makes the language much easier to use and increases the expressive power of single queries. However, many commercial systems are not fully orthogonal. In particular, they often place restrictions on outer join queries. For example, at most one outer join might be allowed, or no other joins might be allowed after an outer join. With such systems, it is sometimes necessary to formulate a single standard query as a series of queries using intermediate tables to store results on the way.

Right outer join is analogous to left. Full join is the union of left and right. The word **outer** is assumed for left/right/full joins, so may be omitted (e.g., **natural left join** or **left join**). Outer joins are not associative, so be careful with the join order when outer joining three or more tables. As discussed later, outer joins can be emulated in the old SQL-89 syntax by means of **union** and subqueries.

When different rows of the same table must be compared, we need to join the table to itself. Such *self-joins* were discussed in Section 12.1. Recall that this requires introducing an alias for the table. In SQL, a *temporary alias* may be declared as a *tuple variable* in the **from** clause after the table it aliases. This variable may be assigned any

row from the table and is sometimes called a "correlation variable", "range variable", or "table label". For clarity, **as** may be used to introduce the alias.

Figure 12.46 provides a simple example. The base table Scientist stores the name and gender of various scientists, and the query lists pairs of scientists of opposite gender. Here the aliases S1 and S2 are declared in the **from** clause. To understand the self-join, it helps to think of S1 and S2 as copies of the original base table, as shown. The conditional join performs a <>-join on gender (to ensure opposite gender) and a <-join on PersonName (to ensure that each pair appears only once, rather than in both orders).

Some versions of SQL also provide a "create synonym" command for declaring a permanent alias definition. However, this is not part of the standard. In simplified form, the **from** clause syntax of SQL queries may be summarized in BNF as follows (for alternatives in braces, exactly one is required):

```
from table [ [as ] alias ]
   [ , | cross join table [ [ as ] alias ]
      | natural [ inner | [ outer] { left | right | full } ] join table [ [as ] alias ]
            | [ inner | [ outer ] { left | right | full } ] join table [ [ as ] alias ]
                        { on condition | using ( col-list ) }
            | union join table [ [ as ] alias ]
            [ ,... ] ]
```

"copies"

Scientist:

personName	*gender*
Curie, Marie	F
Curie, Pierre	M
Edison, Thomas	M
Lovelace, Ada	F

S1:

personName	*gender*
Curie, Marie	F
Curie, Pierre	M
Edison, Thomas	M
Lovelace, Ada	F

S2:

personName	*gender*
Curie, Marie	F
Curie, Pierre	M
Edison, Thomas	M
Lovelace, Ada	F

```
select S1.personName, S2.personName
from  Scientist as S1 join Scientist as S2
        on S1.gender <> S2.gender
            and S1.personName < S2.personName
```

⇨

S1.personName	*S2.personName*
Curie, Marie	Curie, Pierre
Curie, Marie	Edison, Thomas
Curie, Pierre	Lovelace, Ada
Edison, Thomas	Lovelace, Ada

Figure 12.46 A self-join is used to list pairs of scientists of opposite gender.

In formulating an SQL query, the guidelines discussed in relational algebra usually apply. First state the query clearly in English. Then try to solve the query yourself, watching how you do this. Then formalize your steps in SQL. This usually entails the following moves. What tables hold the required information? Name these in the **from** clause. What columns do you want, and in what order? Name these (qualified if needed) in the **select** list. If the **select** list doesn't include a key, and you wish to avoid duplicate rows, use the **distinct** option. If you need n tables, specify the *n - 1* join conditions. What rows do you want? Specify the search condition in a **where** clause. What order do you want for the rows? Use an **order by** clause for this.

The new join syntax introduced in SQL-92 is convenient, but adds little in the way of functionality. The most useful notations are those for conditional and natural joins (both inner and outer). If you are using a version of SQL that does not support the new syntax, you should take extra care to specify the join conditions in detail and to qualify column names when required.

Any SQL statement may include *comments*. These might be used for explanation, documentation, or to simply comment out some code for debugging (comments are ignored at execution time). In SQL-92, a comment begins with two contiguous hyphens "--" and terminates at the end of the line. In addition to these single-line comments, SQL:1999 introduced multiline comments, starting with "/*" and ending with "*/". SQL Server supports both these comments styles. Here's an example:

```
-- This query retrieves details about the employees who drive a car
select *        -- select all the columns
from   Employee natural join Drives
/* If the natural join syntax is not supported,
      then omit "natural", and specify the join condition in an on-clause */
```

Exercise 12.6

1. This question refers to the student database discussed in Exercise 12.1. Its relational schema is shown. Formulate the following queries in SQL.

   (a) List studentNr, name, degree, and birthYr of the students in ascending order of degree. For students enrolled in the same degree, show older students first.
   (b) For each student named 'Smith J', list the studentNr and the codes of subjects (being) studied.
   (c) List the studentNr, name, and gender of all students studying CS113.
   (d) List the titles of the subjects studied by the student with studentNr 863.
   (e) List the studentNr, name, and degree of those male students who obtain a rating of 5 in a subject titled "Logic". Display these in alphabetic order of name.

(f) List the code and credit points of all subjects for which at least one male student enrolled in a BSc scores a rating of 7. Display the subjects with higher credit points first, with subjects of equal credit points listed in alphabetic order of code. Ensure that no duplicate rows occur in the result.

(g) List the code and title of all the subjects, together with the ratings obtained for the subject (if any).

2. The relational schema shown is a fragment of an application dealing with club members and teams. Formulate the following queries in SQL.

Team (teamName, [captain], [coach])

Member (memberNr, memberName, gender) {M,F}

(a) List the teams as well as the member numbers and names of their captains.

(b) Who (member number) is captain and coach of the same team?

(c) Who (member number) captains some team and coaches some team.

To record who plays in what team, the following table scheme is used:

PlaysIn (memberNr, teamName)

(d) What intertable constraints apply between this table and the other two tables?

Formulate the following queries in SQL.

(e) List details of all the members as well as the teams (if any) in which they play.

(f) List details of all the members as well as the teams (if any) that they coach.

(g) Who (memberNr) plays in the judo team but is not the captain of that team?

(h) Who (memberNr and name) plays in a team with a female coach?

12.7 SQL: in, between, like, and is null Operators

This section examines four special operators (other than the comparators =, <, etc.) that SQL provides for use within search conditions: **in**; …**between**…**and**…; **like** and **is null.** These are sometimes called functions, and the conditions they are used to express are called "predicates" in the SQL standard.

A function is something that takes zero or more values as arguments and returns a single value as its result. You are probably familiar with functions from mathematics or programming, such as cos(x) or sqrt(x). As these examples illustrate, syntactically a function is usually represented as a function identifier preceding its arguments, which are typically included in parentheses.

When the action performed by a function is represented without bracketing all the arguments, we usually describe the notation as involving operators and operands rather than functions and arguments. Operators may be represented in infix, prefix, postfix, and mixfix notation according to whether the operator appears between, before, after, or mixed among the operands. For instance, the sum of 2 and 3 might be set out as:

```
sum (2, 3)          -- function
2 + 3               -- infix operator
+ 2 3               -- prefix operator
2 3 +               -- postfix operator
sum of 2 and 3      -- mixfix operator
```

The four operators we are about to discuss are used to express search conditions. The first three return the value True, False, or Unknown, while the **is null** operator returns True or False. Our initial treatment focuses on the SQL-89 version of these operators. Extensions for SQL-92 onwards are mentioned later. Most of our examples are based on the Person table, reproduced here as Table 12.9.

Suppose we want the names and birth years of the people born in 1950, 1967, or 1974. One way of requesting this information is shown in the following SQL query. As an exercise, check that this results in four rows.

```
select firstname, birthyr from Person
where birthyr = 1950 or birthyr = 1967 or birthyr = 1974
```

Imagine how tedious this way of phrasing the request would be if there were a dozen or more years involved. Partly to make life easier in such situations, SQL includes an **in** operator to handle *bag membership* and hence *set membership*. Using this infix operator, the aforementioned request may be formulated more briefly as:

```
select firstname, birthyr from Person
where birthyr in (1950,1967,1974)
```

Here the search condition is that the birthyr value is a member of the bag containing the values 1950, 1967, and 1974. The order in which these values are written does not matter, nor would it matter if any are duplicated. In general, if x is some expression (e.g., a column name) and a, b, etc. are data values (e.g., numeric or string constants), then the SQL condition shown here on the left is equivalent to the mathematical expression shown on the right:

x **in** $(a, b \ldots)$ means $x \in [a, b \ldots]$

Table 12.9 A relational table storing personal details.

Person:

firstname	*gender*	*starsign*	*birthyr*
Bob	M	Gemini	1967
Eve	F	Aquarius	1967
Fred	M	Gemini	1970
Norma	F	Aries	1950
Selena	F	Taurus	1974
Terry	M	Aquarius	1946

We use **in** instead of "∈", and parentheses "( )" instead of square brackets "[ ]" for *bag delimiters*. Since a set may be thought of as a bag with no duplicates, the **in** operator may also be used for set membership. Unlike many programming languages, SQL allows bags or sets to contain character strings (not just numbers). For example, the following query lists the Aquarians and Taureans (Eve, Selena, and Terry):

```
select firstname from Person
where starsign in ('Aquarius','Taurus')
```

To indicate that the value of an expression does *not* belong to a bag or set, the logical **not** operator may be used with the **in** operator. As with ordinary comparison operators, SQL gives the **in** operator higher priority than logical operators (unlike most programming languages). In SQL the condition "not (x in S)" may be rendered more briefly as **not** x **in** S. Even better, SQL also allows **not in** for "∉", so this may set out more naturally as x **not in** S.

x **not in** (*a, b* …) means $x \notin [a, b \ldots]$

For example, to obtain the names and birth years of those not born in any of the years 1950, 1967, or 1974 the following query may be used (yielding Fred and Terry):

```
select firstname, birthyr from Person
where birthyr not in (1950,1967,1974)
```

Sometimes we wish to determine whether an expression has a value occurring in a *range* of values. In mathematics, to say that some variable x has a value in the range from a to b we usually express this as: $a \leq x \leq b$. This notation is illegal in SQL. Instead we could say: a <= x and x <= b. For the sample population, the following query returns the name and birth year of all but Terry.

```
select firstname, birthyr from Person
where 1950 <= birthyr and birthyr <= 1974
```

As another way to specify range membership, SQL provides the ternary mixfix operator "**…between…and…**" which may be defined as shown, where x, a, and b may be scalar expressions:

x **between** *a* **and** *b* means $a <= x$ **and** $x <= b$

For example, the previous query may be reformulated as:

```
select firstname, birthyr from Person
where birthyr between 1950 and 1974
```

Note carefully that, in contrast to ordinary English, **between** in SQL is read in an *inclusive sense*. For instance, both 1950 and 1974 are included in the given range.

Notice also that here **and** is just part of the mixfix operator: it is not a logical operator. As already noted, the <= operator may be used to order strings as well as numbers. So strings may be used as operands. For example, the following query returns the bag ('Bob', 'Eve', 'Fred').

select firstname **from** Person
where firstname **between** 'Bob' **and** 'Fred'

Nonmembership in a range may be expressed with the help of the **not** operator, which has lower priority than **between**, and may be placed just before the word **between**. Thus each of the following conditions is equivalent to $x < a$ **or** $x > b$.

x **not between** *a* **and** *b* **not** (*x* **between** *a* **and** *b*) **not** *x* **between** *a* **and** *b*

The **not between** formulation is easier to read. For example, the following query returns just the tuple ('Terry', 1946).

select firstname, birthyr **from** Person
where birthyr **not between** 1950 and 1974

In the pure relational model, column values are atomic. However, SQL provides a number of ways of accessing substrings within character string values. In particular, the **like** operator may be used for *pattern matching* with character strings. The general form of the condition in SQL-89 may be set out in BNF thus:

char-col [**not**] **like** *quoted-string* [**escape** *quoted-char*]

Here char-col is the name of a column based on a character string data type. The quoted string is a character string, surrounded by single quotes, which may contain wildcard characters. You may be familiar with the use of wildcards for matching filenames at the operating system level (e.g., "*" and "?" in MS-DOS). In the absence of an escape clause, SQL gives the percentage character "%" and the underscore character "_" the following special meanings if included in a quoted string operated on by **like**:

%	means	zero or more characters
_	means	any single character

The "%"is generally more useful, but "_" is needed if the character's position in the string is important. Figure 12.47 provides a few examples based on Table 12.9. The **like** operator has priority over logical operators, so "*x* **not like** *s*" means "**not** (*x* **like** *s*)".

```
select distinct starsign from Person
where starsign like 'A%'
```

⇨

starsign
Aquarius
Aries

```
select distinct starsign from Person
where starsign like '%a%'
```

⇨

starsign
Aquarius
Taurus

```
select distinct starsign from Person
where starsign like '_a%'
```

⇨

starsign
Taurus

```
select firstname from Person
where name not like '%e%'
```

⇨

firstname
Bob
Norma

Figure 12.47 "%" and "_" wildcards for pattern matching with the **like** operator.

Note that the "%" and "_" are interpreted as wildcard characters only in the context of a **like** condition. For example, the following query returns the null set. There are no rows that satisfy the condition, since if "=" is used instead of **like** the expression "A%" is taken literally.

```
select starsign from Person
where starsign = 'A%'
```

As discussed earlier, to include a single quote in a string, we use two single quotes. For example, for the table scheme *Actor* (surname, firstname, gender) considered earlier, the following query returns each actor whose surname begins with "D'": **select** surname **from** Actor **where** surname **like** 'D"%'.

To see a few examples where the **like** operator is quite useful, consider Table 12.10, which indicates which subjects are offered in which semester at a given university. The sample population is small to save space. At this university, subject codes have the following meaning: the first two characters indicate the discipline area (e.g., "CS" denotes Computer Science and "PD" denotes Philosophy) and the third character denotes the year level (e.g., "1" for first year level).

Table 12.10 A relational table showing subjects offered and semesters.

Offering:

subject	*semester*
CS112	1
CS112	2
PD102	1
PD102	2
CS225	2
CS314	1

Try to formulate the following queries yourself before peeking at the answers.

List the computer science subjects.
List the first year level subjects.
List subjects higher than first level that are offered in semester 2.

The queries in order are:

```
select distinct subject from Offering where subject like 'CS%'

select distinct subject from Offering where subject like '__1%'

select distinct subject from Offering
where subject not like '__1%' and semester = 2
```

Another common example of a string that is often interpreted as having structure is a person name. Usually a person's surname, as well as either a first name or initials, is recorded (e.g., “Smith, James” or “Smith JB”). Suppose we wish to distinguish between these two parts of a person's name. One way to implement this is to include two columns, one for the surname and another for the firstname (or initials). The structure is then known to the system, and each of these two parts of the name can be accessed individually (this is especially handy for SQL’s **group by** facility). Alternatively, we might use just one column to store the whole name and then use the **like** operator to distinguish the two parts—the name structure then becomes derived rather than stored.

For example, consider Table 12.11, which provides details of players in a mixed doubles tennis match. Try the following queries and then check your answers.

List details of people with surname “Smith”.
List details of people with firstname “James”.

In order, the queries are:

```
select * from Player where personName like 'Smith, %'

select * from Player where personName like '%, James'
```

Table 12.11 A relational table concerning tennis players.

Player:

personName	*gender*	*height*
Smith, James	M	180
Smith, Sue	F	?
Smithers, James	M	175
James, Susan	F	170

Conditions using the **like** operator may optionally include an *escape character*. If chosen, it may be used as a lead-in character to have "%" and "_" interpreted literally rather than as wildcards. For example, the query in Figure 12.48 lists starships with an underscore character in their name. Here "\" is used as the lead-in character; any character that doesn't occur in the string being investigated could be used. Some SQL dialects extend the capability of the **like** operator with additional wildcards. For example, SQL Server uses [ ] to include a range of characters, and [^] to exclude characters.

Although nulls may be displayed (e.g., as "?"), they cannot be used with the usual comparison operators. Instead, a special postfix Boolean …**is null** operator is used. This operator is placed after the name of the column on which it operates. It returns True if the column value is null, and False otherwise. For example, the following query on the Player table returns 'Smith, Sue', since her height is unknown:

```
select personName from Player
where height is null
```

Neither of the expressions "height = null" nor "height = '?'" is allowed. To specify that a value is not null, use the logical **not** operator before the column name or the word **null**. For instance, either query shown here returns the names of the three other players.

Starship:

shipName	*maxSpeed*
Alpha_1	warp 5
Enterprise	warp 8
Epsilon_33	warp 7
Galactica	warp 5

```
select shipName from Starship
where shipName like '%\_%' escape '\'
```

⇨

shipName
Alpha_1
Epsilon_33

Figure 12.48 Here "\" is used as an escape character to detect an underscore.

```
select personName from Player where not height is null
select personName from Player where height is not null
```

Using "col" to denote a column, the BNF syntax for null comparisons in SQL-89 is:

col **is** [**not**] **null**

While the **is null** operator always returns True or False, other comparison operators return unknown when one of the arguments is the null value. Suppose we want the names of the players who are between 170 and 175 cm tall (inclusive). The following query returns just two names: 'Smithers, James', 'James, Susan'.

```
select personName from Player
  where height between 170 and 175
```

Although Sue Smith's height might be in this range in the real world, she is excluded from this result. The system does not know her height, so will not evaluate the condition to True in her case. Suppose now we want the names of those players whose height is not in the range 170..175. The following query returns just one name: 'Smith, James'.

```
select personName from Player
where height not between 170 and 175
```

Notice that Sue Smith is excluded from this result as well. As far as the system is concerned, in her case the truth value of the condition "height between 170 and 175" is neither true nor false; rather it is unknown.

A row is included in a result only if it satisfies the search condition, i.e., the condition is (known by the system to be) True for that row. If the condition is either False or Unknown, the row is excluded from the result. As an extreme example, the following query returns all the players except Sue Smith.

```
select personName from Player
where height = 170 or height <> 170
```

For Sue Smith's row, each comparison in this condition evaluates to unknown, and applying the **or** operator to unknowns also gives an unknown. The truth value of any condition is evaluated according to a *three-valued logic*, as set out in Figure 12.49. Here "1", "0", and "?" denote the values "True", "False", and "Unknown", respectively.

Although SQL-89 and SQL-92 allow only one kind of null value, Codd (1990) proposed a four-valued logic in "version 2" of the relational model to allow for two kinds of unknown (applicable and inapplicable), but this has not received popular approval.

p	**not** *p*
1	0
0	1
?	?

p	*q*	*p* **and** *q*
1	1	1
1	0	0
1	?	?
0	1	0
0	0	0
0	?	0
?	1	?
?	0	0
?	?	?

p	*q*	*p* **or** *q*
1	1	1
1	0	1
1	?	1
0	1	1
0	0	0
0	?	?
?	1	1
?	0	?
?	?	?

Figure 12.49 Truth tables for three-valued logic (1 = True, 0 = False, ? = Unknown).

A summary of the SQL-89 syntax of the four operators introduced in this section is shown. Here "expn" denotes an expression such as a column name or constant (or a combination of these connected by arithmetic operators +, −, *, / ... see later).

expn [**not**] **in** (*constant-list*)
expn [**not**] **between** *expn* **and** *expn*
char-col [**not**] **like** *quoted-string* [escape *quoted-char*]
col **is** [**not**] **null**

In addition, SQL-89 allows the **in** operator to be used with subqueries, and the use of quantified comparisons and the **exists** quantifier in search conditions (see later). SQL-92 (and later) significantly extended the kinds of expressions and operators allowed in search conditions.

For example, SQL-92 allows a *row value expression* (e.g., a tuple of values) as the left argument of any comparator (e.g., =, **in**) and includes **unique**, **match**, and **overlaps** predicates. **Unique** is used to check whether each null-free row returned by a subquery is unique, **match** is used to check whether a row matches a row returned by a subquery, and **overlaps** is used to test whether two datetime periods overlap.

SQL:1999 goes further, adding **similar**, **distinct**, and **type** predicates for use in search conditions. **Similar** allows character strings to be compared by means of a regular expression, **distinct** tests whether two row values are distinct, and **type** tests whether a user-defined type value expression conforms to a user-defined type specification. Dialects of SQL differ in their support for these standard predicates. However, all SQL dialects now support the SQL-89 syntax discussed in this section.

Exercise 12.7

The table contains data about computer languages. Phrase the following queries in SQL.

(a) Which languages were released in the years 1959, 1975, or 1979?
(b) Which languages were released in the period 1959 – 1979?
(c) Which languages do not have their release year recorded?
(d) List the name and release year of any Goodo language.

(e) List the name of any language ending with “OL”.
(f) List the name and release year of any Pascal language.
(g) List language names that are five characters long.
(h) List language names with 'o' as the second character.
(i) List the name of any language containing the letter “O” or “o”.
(j) List the name and release year of languages not starting with “P” that were released after 1959 and before 1979.
(k) List language names that either are six characters long and have “a” as the last or second last character or are five characters long and end with “OL”.

CompLanguage:

title	*releaseYr*
FORTRAN	1957
ALGOL	1958
COBOL	1959
Logo	?
Pascal	1971
Prolog	1972
SQL	?
Modula	1975
Modula 2	1979
Goodo Pascal 1.0	1983
Goodo Pascal 2.0	1984
Goodo Pascal 3.0	1985
Goodo Prolog	1986

2. Suppose that in later years computer languages with the following titles were released: Modula_3; Oberon_2; Ada%93; COBOL%93. Write an SQL query to list all languages whose title includes an underscore or a percentage sign.

12.8 SQL: Union and Simple Subqueries

This section discusses two ways in which **select** queries may be used as components within a larger **select** query. First we examine how to specify the operations of union, intersection, and difference in SQL. Next we look at basic subqueries in SQL. Let’s begin with the union operation. The basic syntax is shown here. The reserved word “union” is allowed only between **select** queries, not between table names.

select-query
union [**all**]
select-query
…

The **select** queries must be union compatible (i.e., same number of columns, and corresponding columns are based on compatible data types). By default, a set is returned

which is the union of the tables returned from the individual **select** queries, with any duplicate rows eliminated. To return a bag, specify **all** to include duplicate rows. Note that this is the opposite of the SQL **select** list syntax, which returns a bag by default and uses **distinct** to remove duplicates. SQL:1999 allows the **distinct** option to be explicitly declared after **union**; if omitted it is assumed by default. The explicit use of **union distinct** is not supported by SQL Server.

As an example, the query in Figure 12.50 lists each person who is an American or drives a Ford. The first **select** finds the Americans ('JimO', 'Lance'), the second **select** finds the Ford drivers ('Colleen', 'Lance'), and the **union** includes any person in either or both of these intermediate results. Although Lance appears in both of the tables to be unioned, he appears just once in the final union since duplicates are removed by default.

If *duplicates* are desired, the **all** option must be specified after **union**, yielding a bag. For example, **union all** in the previous query would cause Lance to appear twice in the result. In general, if a row appears m times in one result table and n times in another, then it appears $m + n$ times in the bag union of those results.

In our example, the corresponding columns (shortName and driver) are compatible but have different names. Since column headings in the final result are chosen from the first query in the union, the heading "shortName" appears here. Although this query may be formulated in other ways (see subqueries later), the union operation provides the most natural method of merging results from compatible result tables.

Since base tables are rarely union compatible, the union operator is mostly used on derived tables that have been made union compatible by projection, as in this example. The **union** operation may be applied several times in a query. If an **order by** clause is used, it must come after the last query in the union. For example, appending the clause **order by** shortName sorts the result into 'Colleen', 'JimO', 'Lance', 'Walter'.

In SQL-92 (and later), a **corresponding** option may be added to specify the common columns used for the union. For example, if the driver column in Figure 12.50 were instead named "shortName', the query could be declared as **select** * **from** Person **union corresponding** Drives or as **select** * **from** Person **union corresponding by** (shortName) Drives. This option is not yet supported by most SQLs, including SQL Server.

Person:

shortName	*nationality*
Colleen	Irish
JimO	American
Lance	American
Norma	Aussie
Terry	Aussie
Walter	American

Drives:

driver	*carModel*
Colleen	Ford
JimO	Honda
Lance	Ford
Lance	Honda
Terry	Saturn

Who is an American or drives a Ford?

```
select  shortName from Person
where nationality = 'American'
union
select  driver from Drives
where carModel = 'Ford'
```

⇨

shortName
Colleen
Lance
Walter
JimO

Figure 12.50 By default, union eliminates duplicates.

Now consider the query: Who drives a Ford, Holden, or Honda (i.e., who drives at least one of these car models)? Although we could formulate this using two unions, the query can be answered with one scan of the rows from the Drives table. So it is simpler (and more efficient) to use the **or** operator, thus:

```
select driver from Drives
where carModel = 'Ford' or carModel = Holden' or carModel = 'Honda'
```

When the operands of a **union** operator are disjoint (i.e., mutually exclusive), the **union** operator is sometimes used to add descriptions to the output even when only one base table is involved. As an example, consider the Expert and Novice tables in Figure 12.51. Suppose that we want to list the names of the judo players, as well as a label to indicate who are the experts and who are the novices. To obtain the names, we can union a **select** query for the judo experts with another **select** query for the judo novices. We could then identify the expert and novice judoka by including the strings "expert judoka" and "novice judoka" in the relevant **select** lists. We can improve on this by introducing "judo level" as an alias for the labels. This provides a meaningful column header for the labels and allows us to shorten the labels to just "expert" and "novice", as shown. Notice the use of double quotes in "judo level" to allow an embedded space.

Although this technique of adding descriptive strings in unions is handy when the operands are disjoint, it is not of much use if they overlap. Any originally overlapping rows become disjoint when the string values are added into the rows (since they will differ in that string value). Hence, such rows would appear more than once in the union.

In SQL-92 (but not SQL-89), the **intersect** operator is provided for *set intersection* and the **except** operator handles *set difference*. These work analogously to the **union** operator. They are allowed only between **select** queries that return union-compatible tables, and duplicates are removed from the result unless the **all** option is specified.

Expert:

person	*sport*
Ann	aikido
Ann	judo
Bill	judo
John	bojutsu

Novice:

person	*sport*
Ann	karatedo
Bill	karatedo
Cathy	aikido
Cathy	judo
David	kendo

```
select  person, 'expert' as "judo level"
from    Expert
where sport = 'judo'
union
select  person, 'novice' as "judo level"
from    Novice
where sport = 'judo'
order by "judo level", person          ⇨
```

person	*judo level*
Ann	expert
Bill	expert
Cathy	novice

Figure 12.51 Constants may be used with union to indicate the source of the rows.

Refer back to the tables *Person*(shortName, nationality) and *Drives*(driver, carModel) in Figure 12.50. Now consider the query: Which Americans drive a Ford? This can be reworded: Who is an American and drives a Ford? Since the nationality facts and carModel facts are in separate tables, this logical "and" operator can naturally be implemented by a table intersection operator, as follows, yielding the result {'Lance'}.

```
select shortName from Person
where nationality = 'American'
intersect
select driver from Drives
where carModel = 'Ford'
```

Similarly the query "Which Americans do not drive a Ford?" can be formulated as a set difference as follows, yielding the result {'JimO', 'Walter'}.

```
select shortName from Person
where nationality = 'American'
except
select driver from Drives
where carModel = 'Ford'
```

Unlike union and intersection, the difference operator is not commutative. For example, reversing the order of the operands in the previous query results in {'Colleen'}, since the reversed query has the different meaning "Which Ford drivers are not American?".

If the **all** option is used after **intersect** or **except**, it returns a bag intersection or bag difference, respectively. In general, if a row appears m times in one table and n times in another, then it appears $\min(m, n)$ times in the bag intersection and $\max(m - n, 0)$ times in the bag difference.

Although all the main commercial SQL dialects support the **union** operator, several do not yet support the **intersect** and **except** operators. As we now discuss, these two operators can be emulated using subqueries, so their absence does not entail a loss of functionality.

SQL subqueries are like component queries within nested formulations in relational algebra. Although SQL subqueries are queries in their own right, they must be enclosed in parentheses (unlike **select** queries that are unioned), and they have some limitations on their use. Ideally, a language ought to be orthogonal—wherever a value or result is legal, an expression that returns that value or result should be legal. In SQL-89, however, subqueries are allowed only in a few places where their result is legal. Although SQL-92 removed these restrictions, few commercial SQLs are fully orthogonal.

A subquery is a parenthesized **select** query embedded in another query. Let Θ denote one of the comparison operators (=, <>, <, >, <=, >=). In SQL-89, a subquery is legal only in the following three contexts:

```
expression [not] in (subquery)                -- membership subquery
expression Θ [all | some | any] (subquery)    -- [quantified] comparison subquery
exists (subquery)                             -- existential subquery
```

A *membership subquery* follows an **in** operator. In SQL-89, a membership subquery must be a *singleton query* (i.e., it must return a single column). This is consistent with our earlier use of the **in** operator with a bag of values (a column is a just bag of values, displayed vertically). In SQL-92, the **in** operator may be preceded by a row value expression, in which case the subquery may return a multicolumn table.

A *comparison subquery* follows a comparator (e.g., =). If the comparator is modified by one of the quantifiers **all**, **some**, or **any**, this is a *quantified comparison*. In SQL-89, a comparison subquery must return a single column; if the comparison is not quantified, the subquery must return a single value. This is consistent with the normal use of comparison operators. In SQL-92, a comparator may be preceded by a row value expression, in which case the subquery may return a multicolumn table or, if unquantified, a row value. An *existential subquery* appears after the **exists** quantifier and may return any table.

In SQL-92, subqueries may be used in other places (e.g., **from** clauses and **check** clauses), as long as their result table makes sense there (orthogonality). **From** clause subqueries require an alias for the derived table even if not needed. The syntax is:

```
... from (subquery) [as] alias
```

SQL Server supports subqueries in a **from** clause, but not in a **check** clause. Some SQL dialects limit their subquery support to the three SQL-89 cases, each of which embeds the subquery in a **where** clause of an outer query, thus:

```
select ...          -- outer query
from ...
where ...
  ( select ...      -- subquery
    from ... )
```

If the subquery is self-contained (i.e., it makes no reference to the outer query) it is said to be *uncorrelated*. In this case, it is evaluated first and replaced by its result (e.g., a bag of values) before the outer query is processed. In contrast, *correlated subqueries* refer to their outer query. This section considers only simple, uncorrelated membership and comparison subqueries. Existential and correlated subqueries are covered later.

Let's now consider *membership subqueries*. The condition *expression* **in** (*subquery*) returns True if and only if the value of the expression is included in the subquery result. If the subquery returns the null set, the condition evaluates to unknown. The outer query result includes only the rows where the condition evaluates to True.

To discuss some examples, consider the tables shown in Figure 12.52. Assume that there is a subset constraint from Mountain.countryName to Country.countryName. The opposite constraint does not apply (e.g., The Netherlands has no mountains). To save space, only some mountains are shown for the countries.

Country:

countryName	*population*
Australia	21 000 000
India	1 128 000 000
Netherlands	16 000 000
USA	303 000 000

Mountain:

mtName	*countryName*
Kosciusko	Australia
Mana	India
McKinley	USA
Rainier	USA

Figure 12.52 Some countries and some of their mountains (if any).

Consider the query: Which mountains are located in a country with more than 100 million people? We could begin by finding the countries with the relevant populations, and then finding the mountains in those countries. This may be set out in SQL as follows:

```
select mtName from Mountain
where countryName in
  (select countryName from Country
  where population > 100000000)
```

The subquery is evaluated first, resulting in the bag: ('India', 'USA'). After the subquery has been replaced by this intermediate result, the outer query is evaluated, finally resulting in a list of three mountains: Mana, McKinley, and Rainier. Of course, this query could have been phrased instead using a join. As an exercise, you may wish to do this.

Subqueries may be nested (i.e., their search conditions may include subqueries). To improve readability, subqueries should always be indented. The deeper the level of nesting, the greater the indentation should be.

Often, queries using subqueries may be replaced by **join** queries, and vice versa. Some joins cannot be reformulated in terms of subqueries. In particular, *if the* **select** *list includes columns from more than one table, a join must be used* (self-joins must be used if the select columns are from different copies of the same table). For example, the following query cannot be reformulated as a subquery (what columns would the result table contain?).

```
select * from Country natural join Mountain
```

Now consider the relational schema for our software retailer UoD, which was discussed in Exercise 6.3 and is reproduced in Figure 12.53. Suppose we wish to issue the query: List the customer number, customer name, and invoice number of each invoice that records a shipment of the product titled 'Quinquo'. You might like to try this yourself in SQL before reading on.

The customer name, customer number, and invoice number are to be selected, but do not all occur in the same table. Hence we must include a join in the query to bring these together on the same row of the result table.

Figure 12.53 The relational schema for an invoice application.

Looking at the schema, the minimum join we require is to join the Customer and Invoice tables together on customerNr. The rest of the navigation between tables can be done with subqueries, because we don't need to include any other columns from those tables in the **select** list of the final result. This approach leads to a formulation with one join and nested subqueries as follows:

```
select customerNr, customerName, invoiceNr
from   Customer natural join Invoice
where invoiceNr in
   (select invoiceNr from Lineitem
    where itemCode in
          (select itemCode from Item
           where title = 'Quinquo')
```

Alternatively, we could replace any of the subqueries by a join. For example, the following formulation uses three joins and no subqueries:

```
select customerNr, customerName, invoiceNr
from   Customer natural join Invoice
       natural join Lineitem
       natural join Item
```

If your SQL dialect doesn't support the natural join syntax, use either the conditional join syntax or the old SQL-89 join syntax instead.

As a negated example of subqueries, the following query lists the countries with no mountains. For the given population, this results in just one country: Netherlands.

```
select countryName from Country
where countryName not in
   (select countryName from Mountain)
```

Can you replace a negated subquery with an inner join? No! However, you can replace it with an outer join. For example, the aforementioned query may be reformulated as:

```
select countryName
from Country natural left outer join Mountain
where mtName is null
```

The query may also be formulated using a set difference operation, thus:

```
select countryName from Country
except
select countryName from Mountain
```

To list those countries that do have mountains, the following query could be used:

```
select countryName from Country
where countryName in
   (select countryName from Mountain)
```

This performs a set intersection and is equivalent to:

```
select countryName from Country
intersect
select countryName from Mountain
```

So, when just single columns are required, *the use of* **in** *and* **not in** *with subqueries enables set intersection and set difference operations to be formulated* even in SQL-89. To discuss this further, recall the *Person*(shortName, nationality) and *Drives*(driver, carModel) tables from the previous section. For convenience the sample population for these tables is repeated in Figure 12.54.

Recall that the following queries were used to list the Americans who drive a Ford or do not drive a Ford, respectively:

```
select shortName from Person
where nationality = 'American'
intersect
   select driver from Drives
   where carModel = 'Ford'
```

```
select shortName from Person
where nationality = 'American'
except
   select driver from Drives
   where carModel = 'Ford'
```

Person:

shortName	*nationality*
Colleen	Irish
JimO	American
Lance	American
Norma	Aussie
Terry	Aussie
Walter	American

Drives:

driver	*carModel*
Colleen	Ford
JimO	Honda
Lance	Ford
Lance	Honda
Terry	Saturn

Figure 12.54 Two relational tables.

For a dialect that does not support the **intersect** or **except** operators, these queries may reformulated using membership subqueries, thus:

```
select shortName from Person
where nationality = 'American'
  and shortName in
   (select driver from Drives
    where carModel = 'Ford')
```

```
select shortName from Person
where nationality = 'American'
  and shortName not in
   (select driver from Drives
    where carModel = 'Ford')
```

This technique will not work in SQL-89 if we want to obtain the intersection or difference of tables with more than one column. However, the exists quantifier may be used with subqueries to handle such cases, as discussed later. In SQL-92, the multiple column case can be handled with membership subqueries, because the **in** operator works with row value expressions.

Now suppose that we tried to list the Americans who did not drive a Ford by using the following query. Is this correct? Work out the result table for yourself before reading on.

```
select shortName
from Person join Drives
on shortName = driver
where nationality = 'American' and carModel <> 'Ford'
```

The result from this query lists both JimO and Lance. The query is incorrect for two reasons. As formulated, the query actually asks for the Americans who drive a car other than a Ford. This is very different from the intended query. Lance is wrongly included since he drives a Honda (even though he does drive a Ford). Walter is wrongly eliminated because he drives no car. The condition of never matching a specified value (on any row) is quite different from the condition of matching a different value on some row.

Subqueries may be used with the **union** operator to emulate *outer joins*. For example, consider the query: "List the empNr and name of each employee, as well as their car models (if any)". In SQL-92 syntax this may be set out as:

```
select Employee.empNr, empName, carModel
from   Employee left outer join Drives
  on Employee.empNr = Drives.empNr
```

In SQL-89 syntax, this may be rephrased as follows. Here a string constant '?' (padded with blanks if needed) is used to emulate a null value, since the standard forbids **null** as a **select** item. However, many dialects, including SQL Server, do allow us to say "**select null** …", which can be handy at times.

```
select Employee.empNr, empName, carModel
from   Employee, Drives
where Employee.empNr = Drives.empNr
```

```
union
select Employee.empNr, empName, ' ? '
from   Employee
where empNr not in
   (select empNr from Drives)
```

The remainder of this section discusses *quantified comparison subqueries*. Recall that comparison subqueries may be used to formulate conditions of the form:

expression Θ [**all** | **some** | **any**] (*subquery*)

where Θ is a comparison operator (e.g., =, >) and **all**, **some**, and **any** are quantifiers. In the absence of a quantifier, the subquery must return a single value, which can be guaranteed by using a function (see next section). If such a quantifier is used, we have a quantified comparison subquery. The quantifiers have the following meanings:

some	means	*at least one*
any	means	*at least one*
all	means	*each* (taken one at a time)

In consequence, the following equivalences hold:

in	≡	= **some**	≡	= **any**
not in	≡	<> **all**		

To avoid confusion, **in** and **not in** should be used instead of their equivalent quantified comparisons, and **some** should always be used instead of **any**. In SQL, **some** and **any** are treated as synonyms. However, this is not always the case in English. For example, "<> any" means "not in" in English, but not in SQL! To appreciate this point, compute the result of the following query for yourself.

```
select countryName from Country
where population <> any (select population from Country)
```

After evaluating the subquery, we may express the condition as "population <> **any** (190000000, 1000000000, 16000000, 270000000)". In English, "<> any" means "= none". Since each population equals itself, we would expect no rows to be returned. But the query returns all the populations, since each is not equal to some (i.e., at least one) of the populations, and SQL treats "<> any" to mean "<> some".

As a proper use of a quantified comparison, consider the query: Which country has the largest population? This may be formulated as follows.

```
select countryName
from   Country
where population >= all (select population from Country)
```

Remember that **all** is interpreted distributively as "each, taken individually" rather than collectively. The query result correctly lists India. As discussed in the next section, a **max** function could alternatively be used to reformulate the search condition as: population = (**select max**(population) **from** Country). As discussed later, however, quantified comparisons can also handle some cases not amenable to functions (e.g., group extrema).

Exercise 12.8

1. A database is used to store information on diets. The relational schema and a sample population are provided. Weight (or, more correctly, mass) is measured in kilograms.

Eater:

person	*weight*	*gender*
Ann	70	F
Bill	70	M
Humphrey	150	M
Sue	60	F

Food:

foodName	*foodClass*
apple	fruit
beef	meat
chicken	meat
orange	fruit
peas	vegetable
potato	vegetable

Eats:

person	*foodName*
Ann	apple
Ann	beef
Ann	potato
Bill	apple
Bill	potato
Humphrey	apple
Humphrey	beef
Humphrey	chicken
Humphrey	orange
Humphrey	peas
Humphrey	potato
Sue	apple
Sue	chicken
Sue	orange
Sue	peas

For this database, formulate each of the following queries in SQL.

(a) Who is either a male or a person weighing over 60 kg?
(b) Who is either a male or a person who eats peas?
(c) Who is female and weighs over 60 kg?
(d) Who is female and eats potatoes?
(e) Who weighs over 60 kg but does not eat beef?
(f) List all pairs of people who are of the same weight but opposite gender.
(g) Who eats vegetables?
(h) List the name and weight of those males who eat at least one kind of meat.
(i) Who eats some food other than meat?

(j) Who are vegetarians (i.e., who does not eat meat)?
(k) List the name, weight, and gender of all eaters, identifying each as either a vegetarian or meat eater, and listing vegetarians first, then ordering by gender (females first), then by weight, and finally by name.

2. Consider the following relational schema and sample population (for simplicity, assume that academics are identified by their surname).

BornIn (academic, country)

Attended (academic, conference)

HeldIn (conference, country)

BornIn:

academic	*country*
Everest	America
Smith	America
Meersman	Belgium

Attended:

academic	*conference*
Everest	ER-99
Everest	ORM-1
Meersman	ORM-1
Meersman	CAiSE-00

HeldIn:

conference	*country*
CAiSE-00	Sweden
DAMA-00	America
ER-99	France
ER-00	America
ORM-1	Australia

Formulate queries (a)–(f) in SQL-89:
(a) For conference attendees, list their name, birth country, and conferences attended.
(b) As for (a), but include all academics.
(c) For those academics who attended conferences, list their surname and the name and place of their conferences.
(d) As for (c), but include the conferences not attended by the listed academics.
(e) List the name and birth country of academics born in a country that held a conference, as well as the conferences held there.
(f) As for (e), but include all countries (each additional country will have birth or conference details, but not both).
(g)–(l) Formulate queries (a)–(f) in SQL:1999, making use of its extra join operators.

12.9 SQL: Scalar Operators and Bag Functions

A *scalar* is a data value (e.g., a number, string, or date). This section provides a brief discussion of the scalar operators and bag functions available in SQL-89, as well as some of the further operators and functions included in later versions.

In SQL-89, the scalar operators are the four *arithmetic operators* +, −, *, and /. As unary operators, + and − provide the sign of the number (e.g., +3 or −3). If omitted, + is assumed (e.g., 3 = +3). As binary operators, + and − perform *addition* and *subtraction*. The binary operators * and / perform *multiplication* and *division*; they have no

unary version. Each is a number-forming operator on numbers. Their operands may appear as column names, constants, function calls, or expressions formed from these by the use of arithmetic operators and perhaps parentheses. Scalar expressions may be used as items in a **select** list or a search condition.

Let's begin by seeing how these operators may be used to provide a simple calculator facility in SQL to compute the value of an arithmetic expression. We simply include in our database a dummy table with just one row and then include the expression as the sole item in a **select** list for this table. Some DBMSs have dummy tables built in, but let's create our own by issuing the command **create table** C (c **char**) and then populating this table with the dummy value "c", thus: **insert into** C **values** ('c'). We can now use this table to do simple calculations. As a trivial example, to compute the sum obtained by adding 2 to the product of 3 and 4, we can issue the following query.

```
select 2 + 3 * 4 from C   →   14
```

For such calculations, the contents of C are not accessed, and hence are irrelevant (except that by confining C to one row, the desired value displayed just once). In SQL Server, the **from** clause is optional, so you can simply use **select** 2 + 3 * 4 without having to create a dummy table. The arithmetic operators obey the usual priority convention. Multiplication and division have top priority, with addition and subtraction second. Operators on the same level are left associative (i.e., are evaluated left to right in the order in which they appear). Parentheses may be used to override this evaluation order (parenthesized expressions are evaluated before being operated on from outside). For example: 2 + 8 / 2 * 4 = 18.

For the division operation on exact numeric data, the SQL standard leaves it implementation defined as to the scale (number of digits after the decimal point) of the result. In many SQL dialects (including SQL Server), this scale is 0, so that the division operator performs integer division if its operands are integers. For example:

```
select 14/3 from C        →   4
```

In this case, real division can be obtained by making one of the operands real. For example:

```
select 14.0/3 from C      →   4.666666
```

In some dialects, this displays as 4.666666E+000. Here "E" is read "times 10 to the power", so the aforementioned value is read "4.666666 times 10 to the power 0". Since $10^0 = 1$, this result boils down to just 4.666666.

SQL-92 introduced the **cast** function to enable data to be recast or converted into data of another type (with some restrictions). This can be used to control the scale of a division result. For example, in SQL Server the following query results in 4.67. Whether results are rounded (as here) or truncated is implementation dependent.

```
select cast(14.0/3 as numeric(9,2)) from C   →   4.67
```

When arithmetic operands are of different types, the result is typically coerced into the "greater" type. For example, the result of the division was coerced into real by including a real operand. When column names rather than constants are involved, coercion to real can be achieved by multiplying by 1.0 (e.g., **select** 1.0 * quantity/2).

The result of an arithmetic operation between two numbers of the same type (e.g., **smallint**) may be coerced into a larger type (e.g., **integer**) to avoid possible overflow problems. As not all SQL dialects behave in the same way, you should check the numeric computation rules for your own dialect.

As a practical example using scalar operators, consider the table scheme *LineItem*(invoiceNr, itemCode, qty, unitPrice). The items and line totals in Australian dollars for invoice 502 where the unit price is above 100 AUD may be requested as follows, assuming that unit prices are stored in USD and a conversion rate of 1 AUD = 0.80 USD.

```
select itemCode, qty * unitPrice/0.80 as lineTotalAUD
from   LineItem
where invoiceNr = 502
  and unitPrice/0.80 > 100          -- unitPrice > 100 AUD
```

SQL-92 introduced just one more scalar operator, "||", for *concatenating strings* (character strings or bit strings). For example, suppose that we are using the table scheme *Pupil*(pupilNr, surname, firstname, gender, [iq]) to store information about school students. Figure 12.55 shows a sample population, as well as a query to list the full name of each student in the way it is normally stated in English, as well as how much above average their IQ is. Some dialects, including SQL Server, use "+" instead of "||" for the concatenate operator.

Pupil:

pupilNr	*surname*	*firstname*	*gender*	*iq*
103	Adams	Ann	F	120
101	Brown	Tom	M	120
106	Brown	Chris	F	100
102	Collins	Don	M	?
104	Dancer	Ernie	M	95
105	Evans	Ann	F	115

```
select firstname || ' ' || surname as fullname,
       iq - 100 as iqDelta
from   Pupil
order by fullname
```

⇨

fullname	*iqDelta*
Ann Adams	20
Ann Evans	15
Chris Brown	0
Don Collins	?
Ernie Dancer	-5
Tom Brown	20

Figure 12.55 String concatenation using the "||" operator (sometimes rendered as "+").

In the result, each fullname appears as the concatenation of the firstname, a single space character, and the surname. This assumes that the firstname and surname columns are defined using a **varchar** data type. If instead these columns used a fixed length **char** type, extra spaces would appear between the fullname components, aligning the surnames. Note that the IQ delta for Don Collins is unknown ("?" denotes null). This is because his IQ is unknown, and *a scalar operation returns unknown if any of its operands is null*.

SQL-92 extended the use of the +, −, * and / operators to dates, times, and intervals (e.g., subtracting one date from another yields an interval). SQL:1999 extended the use of the string concatenation operator to work with large object strings (**clobs**, **nclobs** and **blobs**) and arrays. No further scalar operators have yet been introduced in standard SQL. The precedence order for the main SQL operators is summarized in Table 12.12.

SQL-89 provides the following five *bag functions:* **count**, **sum**, **avg**, **max**, and **min**. Each operates on a bag of rows and returns a single, scalar value. In SQL-89, the return value must be a number in the case of **count**, **sum**, and **avg** and is either a number or a character string in the case of **max** and **min**. Recall that a bag or multiset is a set in which repetition (although not order) is significant. Each function takes a bag as its argument. For **count**, this may be a table (bag of rows) or a column (bag of data values); for the others, it must be a column.

Table 12.13 summarizes the cases. The **distinct** option may also be applied with the **max** and **min** functions, but obviously would have no effect. SQL-92 and SQL:1999 introduced no new bag functions, but more were added in SQL:2003. Commercial dialects often support further bag functions (e.g., SQL Server includes **stdev** and **stdevp** functions to compute standard deviations).

A function is called simply by naming it and placing its argument in parentheses after its name. In SQL-89 the argument must be an expression of the following kind: a column name, a constant, or an arithmetic expression formed from column names and constants with the use of arithmetic operators and perhaps parentheses. Any duplicate values are included in the computation unless the keyword **distinct** is placed inside the parentheses just before the expression.

Table 12.12 Operator priority (1 = highest).

Priority	*Operator type*	*Operator(s)*
1	Arithmetic/temporal	* /
2		+ −
	String/array	||
3	Comparison	= <> < > <= >= **in** **between** **like** **null**
4	Logical	**not**
5		**and**
6		**or**
7	Relational	**union** **intersect** **except**

Table 12.13 The five bag functions in SQL-89.

Bag function	*Result*
count (*) **count** (**distinct** *colname*)	number of rows in bag number of distinct values in column
sum (*numeric-expn*) **sum** (**distinct** *colname*)	sum of *numeric-expn* values in bag sum of distinct values in column
avg (*numeric-expn*) **avg** (**distinct** *colname*)	average of *numeric-expn* values in bag average of distinct values in column
max (*expn*)	maximum of *expn* values in bag
min (*expn*)	minimum of *expn* values in bag

Function calls may be included in a **select** list. In this case, every other item in the **select** list must also include a function call, unless the item is a constant or grouping is being performed (see later).

The **sum** and **avg** functions require numeric expressions, whereas **max** and **min** accept number or string expressions. **Count** (*) treats nulls like actual values. All other function calls remove the nulls before computing the result. If the argument bag is empty (either initially or because nulls were removed), **count** returns 0 and the other functions return null.

The count function may be used in only two ways. **Count** (*) returns the number of rows in the specified table, whereas **count** (**distinct** colname) returns the number of distinct values in the named column. If **distinct** is specified, any duplicates are excluded. If duplicates are wanted, the keyword **all** may be used; however, since this is the default, it is often omitted.

A few examples based on the Pupil table in Figure 12.55 are shown in the following queries. With the third example, a **where** clause is used to filter out unwanted rows before the function is called.

```
select count ( * ) from Pupil                    →  6

select count ( distinct gender ) from Pupil  →  2

select count ( * ) from Pupil
where iq > 100                               →  3
```

The **sum** function returns the sum of the values in the column, and **avg** returns the average of these values. For example:

```
select sum ( iq ) from Pupil                     →  550
select sum ( distinct iq ) from Pupil        →  430
select avg ( iq ) from Pupil                     →  110
```

Note the exclusion of nulls here. Since Don Collins's IQ is unknown, there are only five IQ values to be considered. Since these total 550, their average is 110.

If the argument of the **avg** function is exact numeric, the SQL standard leaves the scale of the result implementation defined. For example, the following query returns different results depending on the SQL dialect used.

```
select avg ( distinct iq ) from Pupil   →   107.50    -- in some SQLs
                                        →   107       -- in SQL Server
```

In SQL Server the scale here is 0, so any fraction in the average result is lost. To include the fraction, you can multiply the argument by 1.0 to coerce it to real. For example:

```
select avg ( distinct 1.0 *iq ) from Pupil   →   107.500000
```

The functions **max** and **min** return, respectively, maximum and minimum values. If the data type is character string rather than numeric, these values are computed according to ordinal positions in the character collating. For example:

```
select max ( iq ) from Pupil                 →   120
select min ( iq ) from Pupil                 →   95
select max ( surname ) from Pupil            →   Evans
```

In SQL-89, the bag functions may be called only in a **select** list (or in a **having** clause—see later). For example, consider the request: Who has an IQ above the average pupil IQ? The following formulation is illegal:

```
select firstname, surname from Pupil
where iq > avg ( iq )                        →   Error!
```

Instead, we need to embed the function call within a subquery as follows. This query correctly lists Ann Adams, Tom Brown, and Ann Evans in its result.

```
select firstname, surname from Pupil
where iq >
  ( select avg ( iq ) from Pupil )
```

A few more examples are given in the following SQL queries. The first query computes the difference between the highest and the lowest IQ. The second query determines the ratio of highest to lowest IQ (note the multiplication by 1.0 to ensure that the decimal fraction is included). Similarly, in the third example, to compute the mean of the highest and lowest IQ, the divisor is 2.0 rather than 2. The fourth example lists those pupils where 80% of their IQ is greater than 90% of the minimum IQ. As an exercise, explain why it would be wrong in SQL-89 to express the search condition as: "0.8 * IQ > 0.9 * (**select min** (IQ) **from** Pupil)".

```
select max( iq ) – min( iq ) from Pupil        →   25

select 1.0 * max( iq ) / min( iq )
from Pupil                                     →   1.263157


select ( max( iq ) + min( iq ) ) / 2.0
from Pupil                                     →   107.500000

select firstname, surname from Pupil
where 0.8 * iq >
  ( select 0.9 * min( iq ) from Pupil )        →   Ann  Adams
                                                   Tom  Brown
                                                   Ann  Evans
```

This example illustrates the fact that in SQL-89, unlike a function call, an arithmetic expression may be used as a term to be compared in a search condition.

Our next example additionally illustrates the fact that built-in functions may take an arithmetic expression as an argument. Consider the following relation scheme: *Window* (windowNr, height, width). The following query may be used to list details about those windows whose height exceeds their width by the greatest amount.

```
select * from Window
where height – width =
    ( select max( height – width ) from Window)
```

SQL-92 and SQL:1999 extended the range of expressions that may feature as arguments of bag functions. For example, **sum** and **avg** may apply to interval types, and **max** and **min** may be applied to user-defined types. Although SQL-92 and SQL:1999 did not add further bag functions, they did add many scalar functions. For example, SQL-92 introduced the following scalar functions, each of which takes zero or more arguments and returns a scalar: **bit_length**, **char_length**, **octet_length**, **case**, **cast**, **collate**, **convert**, **translate**, **current_date**, **current_time**, **current_timestamp**, **current_user**, **session_user**, **system_user**, **user**, **extract**, **lower**, **upper**, **position**, **substring** and **trim**. Most dialects support some of these, as well as other scalar functions of their own.

The **lower** and **upper** functions return uppercase and lowercase versions of the argument, while the **cast** function converts the argument to another data type. Examples of these were given earlier. The **substring** function returns a part of a string, using the syntax **substring** (*string-expn* **from** *start-posn* [**for** *length*]). For example:

```
select substring(surname from 2 for 3)
from  Pupil
where surname = 'Adams'                        →   dam
```

Some dialects use a nonstandard syntax for this function. For example, SQL Server uses commas instead of the **from** and **for** keywords, e.g., **substring**(surname, 2, 3).

The **case** function is quite useful for arranging different output for different cases. Conditions are placed after **when**, with their return option after **then**, and an **else** clause is used to specify the return option if all conditions fail. For example:

```
select firstname, surname,
   case
        when iq > 100 then 'above average IQ'
        when iq = 100 then 'average IQ'
        when iq < 100 then 'below average IQ'
        else 'IQ is unknown'
   end
from Pupil                                  →      Ann   Adams      above average IQ
                                                   Tom   Brown      above average IQ
                                                   Chris Brown      average IQ
                                                   Don   Collins    IQ is unknown
                                                   Ernie  Dancer    below average IQ
                                                   Ann   Evans      above average IQ
```

That covers the most important scalar operators and functions in SQL. A discussion of other scalar functions can be found in references cited in the chapter notes.

Exercise 12.9

1. This question refers to the Diet database used in Exercise 12.8. The table schemes are: Eater(person, weight, gender), Eats(person, foodName), and Food(foodName, foodClass). Formulate SQL queries for the following.
   (a) How many males are there above 100 kg in weight?
   (b) How many different weights are there?
   (c) What is the sum of the weights of the males?
   (d) What is the average weight of the females?
   (e) What is the heaviest weight of those who eat beef?

   For each of the next two questions, give two equivalent solutions, one of which uses a function while the other uses an "all" or "some" quantifier.
   (f) Which females are lighter than every male?
   (g) State the name and weight of those females who are as heavy as at least one male.

   The Log table shown is used by a hardware store to record details about wooden logs.

Log:

code	*diameter*	*len*	*mass*	*cost_price*	*retail_price*
2A	15	2	10	4.00	7.00
3B	20	3	20	6.00	9.50
5C	20	5	30	9.00	13.00
5D	20	5	25	8.00	2.00

The schema is: *Log* (code, diameter, len, mass, cost_price, retail_price).
The diameter, length (here called "len"), mass, and price of the logs are measured in cm, m, kg, and $, respectively. Formulate the following in SQL.

(a) List the absolute mark-up (i.e., retail price "cost price) for all the logs.
(b) List the volume of each log in cubic meters, with the column heading "Volume (cubic m)". Use the formula $V = \pi D^2 L/4$ for the volume of a cylinder, approximating π as 3.14. Express each volume as a fixed-point number, truncated to three decimal places.
(c) List the density (i.e., mass/volume), in kg/m^3, of those logs that are 5m long. Use an appropriate heading and show each density as a fixed-point number truncated to two decimal places.
(d) Which log has the highest relative mark-up (use ratio R.P./C.P.)?
(e) Which log has a less than average ratio of length to diameter? Include this ratio (dimensionless, two decimal places) in the output.

3. The following database is used by a library to record details of books. Authors are identified by name, and books by their International Standard Book Number(ISBN).

A sample population is shown. Formulate the following queries in SQL.
(a) Who are the Australian male authors?
(b) List the titles of books published in the period 1984..1986, ordered by title. (Explain the position of "dBaseIII" in the result.)
(c) List all details of books with "SQL" as part of the title, showing the most recently published ones first.
(d) Who wrote a book titled "Databases" published in 1980?
(e) List the ISBN and title of all books with at least one Australian author.
(f) List the name and nationality of author(s) of any book(s) titled "Informatics" published by Hall in 1986.
(g) List, with suitable descriptors, names of the Australian male authors and the American female authors, with the former shown first.

Book:

isbn	*title*	*publisher*	*yrPublished*	*nrCopies*
101	Databases	Possum	1985	4
202	SQL Primer	Hall	1984	4
246	Databases	West	1980	1
345	dBase III	West	1986	1
400	Modula 2	Possum	1986	4
444	Advanced SQL	Hall	1986	2
500	Informatics	Hall	1986	2

WrittenBy:

isbn	*authorName*
101	Brown J
101	Collins T
202	Adams A
246	Smith JB
345	Jones S
400	Smith JA
444	Adams A
500	Brown J
500	Smith JA

Author:

authorName	*gender*	*nationality*
Adams A	F	Aussie
Brown J	M	Aussie
Collins T	M	Kiwi
Jones S	F	American
Smith JA	M	Aussie
Smith JB	M	American

4. Specify the following SQL queries for the Pupil table discussed in this section.
   (a) List the names of the female pupils in a single column, with surname first, followed by a comma, space, and firstname, e.g., "Adams, Ann".
   (b) List the number of pupils, as well as the number of pupils whose IQ is known.
   (c) List the pupilNr, surname, firstname, and gender of each pupil whose IQ is known, specifying gender as a name ("Female" or "Male") instead of a code ("F" or "M").

12.10 SQL: Grouping

Sometimes we wish to partition the rows of a table into a number of *groups* and display properties that apply to each group as a whole. SQL provides a **group by** clause that may be included in the **select** statement to retrieve such grouped data. *Each group is a bag of rows with the same value(s) for the grouping criteria* listed in the **group by** clause. *The final query result has at most one row for each group.* Figure 12.56 pictures grouping by column *a*. Here there are three groups, one for each of the *a* values. The result may include some group property or function of *a*, depicted here as *f(a)*.

Since the final result includes just one row for each group, the **select** list may include group properties only. In other words, for each group, each **select** item has only one value. To ensure this, each group property must be a grouping criterion (a column name used for grouping), a function call (e.g., **count**(*)), a constant, or an expression formed from these terms (e.g., 2 * groupcol).

Figure 12.56 Partitioning rows into groups with the same value for column *a*.

The basic syntax is as follows:

select *group-property1*, …
from …
[**where** …]
group by *group-criterion1*, …

Table 12.14 will be used to help explain most of these ideas. The relation scheme is *Particle* (particleName, family, charge, mass). To help you visualize the grouping into families, the rows have been sorted by family, and a blank line has been added between the groups. The table lists the main atomic particles belonging to the lepton (light), meson (middle), and baryon (heavy) families. The charge of each particle is given in elementary charges (elementary charge = charge of proton = 1.6×10^{-19} Coulomb). The mass of each particle is expressed as a multiple of the electron mass (9.1×10^{-31} kg).

Table 12.14 Some atomic particles.

Particle:

particleName	*family*	*charge*	*mass*
neutrino	lepton	0	0
mu neutrino	lepton	0	0
electron	lepton	-1	1
mu –	lepton	-1	207
pi 0	meson	0	264
pi +	meson	1	273
pi –	meson	-1	273
K +	meson	1	966
K –	meson	-1	966
K 0	meson	0	974
eta	meson	0	1074
proton	baryon	1	1836
neutron	baryon	0	1839
lambda	baryon	0	2183
sigma +	baryon	1	2328
sigma 0	baryon	0	2334
sigma –	baryon	-1	2343
xi –	baryon	-1	2585
xi 0	baryon	0	2573

Consider the following query: List the families and the number of particles in each. This may be formulated in SQL as shown. For this section, the result tables are listed with the column headers underlined by hyphens (this is typical of most SQLs).

```
select family, count(*)      →   family      count(*)        -- function header
from   Particle                  ---------------------       -- may be omitted
group by family                  baryon        8             -- in some SQLs
                                 lepton        4
                                 meson         7
```

Here the **group by** clause indicates that the particles are to be *grouped* into families (i.e., particles with the same family name are placed in the same group). The *items in the* **select** *list are then applied to each group as a whole*.

If **count** (*) is used, it returns the number of rows in the group. In this case we have eight baryons, four leptons, and seven mesons. If an expression other than a column name is used in the **select** list, some SQLs use the expression as a column header in the result (as shown here), whereas others, including SQL Server, omit the column header. It's normally best to use an **as** clause after the expression, to provide your own header.

In many versions of SQL, the use of **group by** causes a default ordering equivalent to a matching **order by** clause (e.g., **order by** family in this case). However, this is not part of the SQL standard. The syntax of the **group by** clause is:

group by *col-name* [, ...]

Members of the same group must have matching values for all of the columns named in the **group by** clause.

The next example places particles into groups having both the same family name and the same charge.

```
select family, charge, count(*) as tally   →   family      charge       tally
from   Particle                                ---------------------------------
group by family, charge                        baryon        - 1          2
order by family, charge                        baryon          0          4
                                               baryon          1          2
                                               lepton         -1          2
                                               lepton          0          2
                                               meson         - 1          2
                                               meson           0          3
                                               meson           1          2
```

If more than one column is used for grouping, the order in which these columns are specified is irrelevant to the content of the result. This is because the same set of groups must be formed. For example, if we group by attributes *a, b,* or *b, a* then the group with values a = a1, b = b1 is the same as the group with values b = b1, a = a1, and so on. However, if the row order is based on the group order, the order of rows in the result

may be different. For example, the following query returns the same rows as the previous query, in a different order. You can adjust the **order by** clause to change the order:

```
select family, charge, count(*) as tally   →  family      charge      tally
from   Particle                               ----------------------------
group by charge, family                       baryon        - 1         2
order by charge, family                       lepton        - 1         2
                                              meson         - 1         2
                                              baryon          0         4
                                              lepton          0         2
                                              meson           0         3
                                              baryon          1         2
                                              meson           1         2
```

If a function call in the **select** list is used as an ordering criterion, you should introduce an alias for it so that you can reference it by name in the **order by** clause. For example, the following query lists each family and the mass of its lightest particle, ordering by that minimum mass. As an undesirable alternative, you can reference it by its position in the **select** list (e.g., **order by** 2).

```
select family, min (mass) as minMass   →  family     minMass
from   Particle                           ----------------------
group by family                           lepton            0
order by minMass                          meson           264
                                          baryon         1836
```

Use of a **group by** clause implies that items in the **select** list are to be applied to each group as a whole, returning just one value for each group. So any columns mentioned in the **select** list must either be grouped columns or must appear there as an argument to a built-in function (e.g., family and mass in the aforementioned query). So when grouping is being performed it makes no sense, and is in fact illegal, to include a nongrouped column as a whole item in the **select** list. For example:

```
select family, mass from Particle   →   Error!
group by family
```

Since the same family may include several particles of different mass, this request is plain silly (which mass would we choose?). We can, however, obtain a single grouped value related to mass by using a built-in function, such as **min** (mass) or **avg** (mass), as we did with the previous query.

Although the **select** list must not include any nongrouped columns, it is legal to group by columns that are not mentioned in the **select** list. For example, the following query lists the minimum mass, but not the name, of each family.

```
select min(mass) as minMass        →   minMass
from   Particle                        -----------
group by family                        0
                                       264
                                       1836
```

Within the **select** statement, *a* **where** *clause may be included to filter out unwanted rows before the groups are formed and functions are applied.* Sensibly, the **where** clause must be placed before the **group by** clause. For example, the following query lists each family and the average mass of its uncharged particles. To cater for dialects such as SQL Server that truncate averages, the mass has been coerced to real by multiplying by 1.0, and the **cast** function is used to display just two digits after the decimal point. Note that it's legal to apply a scalar function to a bag function, e.g., **cast**(**avg**(…)). It is, however, illegal to apply a bag function to a bag function, e.g., **min**(**avg**(…)).

```
select family, cast(avg(1.0*mass) as decimal(6,2)) as avgUnchargedMass
from   Particle
where charge = 0                   →   family      avgUnchargedMass
group by family                        ----------------------------------
                                       baryon      2232.25
                                       lepton         0.00
                                       meson        770.65
```

Sometimes we wish to restrict our attention to only those groups that satisfy a certain condition. Just as a **where** clause may be used to impose a search condition on individual rows, a **having** clause may be used to impose a search condition on individual groups. In this case, the **having** clause must be placed straight after the **group by** clause to which it is logically connected. The syntax is:

group by *col-name* [, ...]
having *group-condition*

The usual rules for search conditions apply. Logical and comparison operators may be included in the condition. Each comparison compares some property of the group, such as **min**(mass), with a constant or another group property. (Although rare, it is possible to include a **having** clause without an associated **group by** clause: in this case the entire table is treated as one group.) As a simple example, the following query lists the families with more than four particles, and the number of particles for each. Although it makes sense to allow aliases (e.g., tally) in the **having** clause, this is not allowed in many SQLs (e.g., SQL Server).

```
select family, count(*) as tally   →   family      tally
from   Particle                        ------------------
group by family                        baryon        8
having count(*) > 4                    meson         7
```

The next example is harder. To help clarify how it works, the numbers 1 through 6 have been added to indicate the order in which the query lines are processed. These numbers are not part of the query. Trace through the steps yourself to make sure that you follow this order. With respect to the **having** clause, note that the charged leptons are included (uncharged leptons with zero mass are filtered out by the **where** clause) and that the charged baryons are excluded (their maximum mass exceeds 2 000).

Considering only the charged particles, form groups with the same family name, where the group's lightest particle has a mass above 0 and the group's heaviest particle has a mass below 2000. List the family name, number of particles, and largest mass for each group, showing groups with more members first.

```
5 select family, count(*) as chargedTally, max(mass) as maxChargedMass
1 from   Particle
2 where charge <> 0
3 group by family
4 having min(mass) > 0 and max(mass) < 2000
6 order by chargedTally desc
```

```
→  family      chargedTally      maxChargedMass
   ---------------------------------------------
   meson       4                 966
   lepton      2                 207
```

One trivial use of grouping is simply to avoid duplicates. For example, the following query may be used as an alternative to **select distinct** family **from** Particle:

```
select family from Particle      →   family
group by family                      --------
                                     baryon
                                     lepton
                                     meson
```

Now suppose we issue the following query. What will be the result?

```
select family from Particle
group by family
having count(*) = 4
```

Comparing this with the previous query, you may feel that the count of each family will be 1 and hence no rows will be returned (i.e., that the result is the null set). This is wrong! When the **group by** clause does its work, each group still has all its members (which of course all have the same group value). The **having** clause now filters out those groups that do not satisfy its search condition. Finally the **select** list determines which group properties are listed. With this query, the groups are first sorted into three families, two of these families are eliminated (the baryons and mesons), and finally just the family name "lepton" of the remaining group (the four leptons) is listed.

Speaks:

country	*language*
Australia	English
Belgium	Dutch
Belgium	French
Canada	English
Canada	French
Cuba	Spanish
Dominica	English
Dominica	French

Which countries speak
all *the languages spoken in Canada?*

```
select country from Speaks
where "language" in
  (select "language" from Speaks
    where country = 'Canada')
group by country
having count(*) =
          (select count(*) from Speaks
           where country = 'Canada')
```

⇨

country
Canada
Dominica

Figure 12.57 Grouping can be used to perform relational division.

As a further application of grouping, let's consider how simple cases of *relational division* can be handled. Figure 12.57 shows an SQL query for an example handled earlier in Figure 12.27 by relational division. Here the column name "language" is double quoted because it is a reserved word.

Although SQL provides no operator for relational division, we can achieve the same effect by grouping as shown. The first subquery yields ('English', 'French'), so the **where** clause filters out the Dutch and Spanish rows before the groups are formed. Because of the uniqueness constraint on Speaks, the number of Canadian languages spoken by a country is now the number of rows in the group for that country. So the left **count**(*) of the **having** condition returns the values 1, 1, 2, and 2 for Australia, Belgium, Canada, and Dominica, respectively. The second subquery evaluates to 2, so the overall query returns Canada and Dominica.

This approach can be adapted to handle cases when the relevant column association is not unique by using the **distinct** option. For example, consider the relation schemes *Localization*(productCode, "language") and *LanguageUse*("language", country, nrSpeakers). The query "Which products are localized into all languages spoken by a country where at least 10 million speak that language?" requires **count**(**distinct**("language")) instead of **count**(*) in the subquery of the **having** clause. As discussed later, this grouping approach can be extended to cater for queries that involve set comparisons other than those covered by relational division.

We have now covered most of the basic working of SQL's **select** statement. The BNF syntax covered so far may be summarized as shown. Unlike the syntax shown here, the SQL standard actually allows a **having** clause without a **group by** clause (effectively meaning "group by no columns") but the semantics are problematic so we'll ignore this option.

```
select * | [all | distinct] expression [[as] column-alias] [,...]
  from table [[as] alias]
        [,| cross join table [[as] alias]
                | natural [inner | [outer] {left | right | full}] join table [[as] alias]
         | [inner | [outer] {left | right | full}] join table [[as] alias]
```

```
                    { on condition | using (col-list) }
            | union join table [[as] alias]
                [,...]]
[where col-condition]
[group by colname [,...]
    [having group-condition]]
[union | intersect | except ... ]
[order by col [asc | desc] [,...] ]
```

Of the eight table operations of relational algebra, we have discussed the equivalent SQL formulation for the general case of five (projection, selection, Cartesian product, Θ-join, and union) and for special cases of three (intersection, difference, and division). The general cases of intersection, difference, and division can be handled using joins, and also with correlated and existential subqueries, as discussed in the next section.

Exercise 12.10

1. This question refers to the library database discussed in Exercise 12.9. The relational schema is repeated here.

   Assume ISBN is based on a character data type. Formulate SQL queries for the following requests.
   (a) How many authors of each gender are there?
   (b) Place authors into groups of the same gender and nationality, indicating how many there are in each group, with the larger groups shown first.
   (c) Considering only the books published after 1980, list the publishers and the total number of copies of their books in the library.
   (d) For each publisher having an average number of library copies per book above two, show the earliest publication year.
   (e) List the ISBN and the number of authors for each book published before 1986 that has more than one author.
   (f) Restricting attention to male authors of a book published by Hall or Possum, list the number of such authors for each nationality that has at least as many authors of this kind as there are copies in the library of the book with ISBN "444".

2. The relational schema for a dietary database is shown (for a sample population, see Exercise 12.8). Formulate the following queries in SQL.

(a) Who eats all the foods?
(b) Which foods are eaten by all eaters?

12.11 SQL: Correlated and Existential Subqueries

This section extends the subquery work in Section 12.8 by considering correlated and existential subqueries. The subqueries discussed earlier were simple, uncorrelated subqueries. An uncorrelated subquery is computed once, replaced by its result, and the outer query is run. In contrast, a *correlated subquery* relates its search condition to each row of a table named in the outer query and is effectively recomputed for each outer row (see Figure 12.58).

Because of repeated computation, correlated subqueries can sometimes be slow to run. However, they significantly extend the range of queries that can be expressed as a single SQL query and are well worth mastering. Let us refer to the condition inside the correlated subquery as the correlation condition. This condition typically takes the form:

$$a \ \Theta \ b$$

where a is an *inner column* (listed in the **from** clause of the subquery), b is an *outer column* (listed in the **from** clause of the outer query), and Θ is a *comparator* (e.g., $=$, $>$).

To avoid ambiguity, the column names for a or b may need to be qualified by a table name. The three basic cases are summarized in Table 12.15. Since local scope is assumed by default, we may normally leave a unqualified (unless it occurs in two inner tables). However, if a and b have the same unqualified name, we need to distinguish b by qualifying it with its outer table name. If the outer table has the same name as the inner table, we must introduce an alias for it to distinguish it from the inner table.

Figure 12.58 A correlated subquery relates its condition to each row of the outer query.

Table 12.15 Correlation between an inner column *a* and an outer column *b*.

Case	*Situation*	*Declaration*
1	*a* is the name of a column in inner table only *b* is the name of a column in outer table only	$a \Theta b$
2	*a, b* are the same column name but in different tables *T* is the base name of the outer table	$a \Theta T.b$
3	inner and outer tables have the same base name *X* is an alias introduced for the outer table	$a \Theta X.b$

For example, consider the table scheme *Human* (firstname, gender, height). For simplicity, we assume that people in this UoD can be identified by their firstname. A sample population is provided in Figure 12.59, along with a correlated subquery to retrieve the name and height (in centimeters) of those who are above average height for *their* gender. In English, correlations are often made with a *pronoun*, such as "their", "his/her", "its", and "that". In this context, pronouns effectively function as object variables.

Here the same table name "Human" appears in the **from** clause of both inner and outer queries. So to distinguish the gender column in the inner table from the gender column in the outer table, the alias *X* is introduced for the outer table. This allows us to specify the correlation condition as "gender = *X*.gender", rather than "gender = gender" (which, of course, would always be true). Here *X* is said to be a *correlation variable*. You can think of *X* as an extra copy of the Human table.

Human:

firstname	*gender*	*height*
Ann	F	160
Bill	M	180
Carol	F	170
David	M	175
Emily	F	165
Fred	M	170

List the name and height of those above average height for ***their*** *gender*

correlation pronoun

```
select firstname, height
from  Human as X
where height >
    (select avg(height)
     from  Human
     where gender = X.gender)
```

⇨

firstname	*height*
Bill	180
Carol	170

Figure 12.59 Correlation between inner and outer tables with the same base name.

Figure 12.60 In the outer query, X is successively assigned one of the six rows.

Alternatively, you can think of it as a tuple variable that ranges over the rows of the Human table, as depicted in Figure 12.60. Despite its set-oriented syntax, SQL queries are processed one row at a time. Here *X* is successively assigned each row of the Human table, passing through six states (in this example), in each of which it holds one row. For each of these states, the inner query is evaluated, computing the average height for the gender currently assigned to *X*. If you're familiar with programming languages, this should remind you of a nested **for** loop.

Let's walk through the execution of the query. At the outer level, we start at the top row, so *X* is assigned the tuple ('Ann', 'F', 160). Hence in the outer level, gender = 'F' and height = 160. In other words, *X*.gender = 'F' and *X*.height = 160. In the subquery, the correlation condition "gender = *X*.gender" now becomes "gender = 'F'". So the subquery returns the average height of the females, which for the given data is 165. The condition in the outer query now becomes "160 > 165", which is false. Since the top row does not satisfy the condition, it is filtered out, so Ann is excluded from the result.

At the outer level, we now move on to the second row, so *X* is assigned the tuple ('Bill', 'M', 180). Hence *X*.gender = 'M' and height = 180. In the subquery, the condition "gender = *X*.gender" becomes "gender = 'M'". So the subquery returns the average height of the males, which is 175. The condition in the outer query now becomes "180 > 175", which is true. So Bill's firstname and height ('Bill', 180) are included in the result.

Similarly we move through rows 3, 4, 5, and 6 at the outer level, recomputing the subquery each time. Carol is accepted (170 > 165 is true) but the others are rejected (because the conditions 175 > 175, 165 > 165, and 170 > 175 are all false). So the final result lists the firstname and height of just Bill and Carol as shown in Figure 12.59.

Now consider the table scheme *Employee* (empNr, empName, pay, [bossNr]), where pay stores the weekly pay and bossNr is the employee number of the boss of the employee on that row. The foreign key constraint bossNr **references** empNr also applies. A sample population is shown in Table 12.16. Now consider the query: Who earns at least 80% of his/her boss's pay?

Table 12.16 Who earns at least 80% of his/her boss's pay?

Employee:

empNr	*empName*	*pay*	*bossNr*
1	Davis, J	1000	?
2	Jones, E	900	1
3	Smith, J	700	1
4	Brown, T	780	2

Let's try answering the question first, using the sample data: Employee 1 is eliminated because he/she has no boss. Employee 2 earns 900/1000 or 90% of his/her boss's pay, employee 3 earns 70% of his/her boss's pay, and employee 4 earns 780/900 or 86.67% of his/her boss's pay. So the result should list employees 2 and 4. How can we formulate the query in SQL to work with any set of data? You might like to try it yourself before reading on.

The inclusion of the pronoun "his/her" in the question is a clue that the query can be formulated with a correlated subquery, as set out later. *E* is used to denote the employee at the outer level. Although not needed, a second alias *B* has been used to help indicate that the inner employee is a boss.

```
select empNr
from Employee as E
where pay >=
  (select 0.8 * pay
   from  Employee as B
   where B.empNr = E.bossNr)
```

As an alternative, you can formulate the query using a self-join as follows:

```
select E.empNr
from Employee as E join Employee as B
  on E.bossNr = B.empNr
    and E.pay >= 0.8 * B.pay
```

To help you understand the join approach, the following list shows rows formed from the join before the final projection is made on E.empNr.

```
E                                   B
empNr  empName   pay    bossNr     empNr     empName   pay    bossNr
-----------------------------------------------------------------------------
2      Jones, E  900     1         1         Davis, J  1000     ?
4      Brown, T  780     2         2         Jones, E   900     1
```

As this example suggests, computing a correlated subquery is basically equivalent to computing a join, so some care is needed in the formulation of correlated subqueries to avoid unacceptable performance.

Now let's consider *existential subqueries*. In predicate logic, the proposition "There is a frog" may be formulated as "$\exists x\ Fx$ ". Here "$\exists$" is the existential quantifier, with the meaning "there exists at least one", "x" is an individual variable standing for some object, and "F …" abbreviates the predicate "… is a frog". An existential subquery in SQL is somewhat similar. It appears in an *existential condition* of the form:

exists (*subquery*)

Here **exists** is used instead of "$\exists$" for the existential quantifier, and subquery is effectively a tuple variable ranging over rows returned by the subquery. The **exists** quantifier may be thought of as a Boolean function, *returning True if the subquery returns a row and False otherwise.* The **exists** quantifier may be preceded by the logical **not** operator. The possible cases are shown here. The term "a row" means "at least one row". Since the system always knows whether or not a row is returned, a simple two-valued logic applies—existential conditions evaluate to True or False, never Unknown.

exists (*subquery*)	→	True	-- a row is returned (by the subquery)
	→	False	-- no row is returned
not exists (*subquery*)	→	True	-- no row is returned
	→	False	-- a row is returned

Existential subqueries may be used within search conditions in a **where** clause, leading to overall queries of the following form:

```
select …
from  …
where [not] exists
   (select * …)
```

Since the mere existence of a row is all that counts in the subquery result, it is normal to simply use " * " instead of a column list in the **select** list for the existential subquery. Prior to SQL-92, existential subqueries were the only subqueries that could return multiple columns, since in SQL-89 both membership and comparison subqueries had to return a single column. Some SQL dialects still have this restriction.

Member (memberNr, memberName, gender)

Ranked (memberNr, art, rank)

Member:

memberNr	*memberName*	*gender*
1	Adams, Ann	F
2	Brown, Ann	F
3	Halpin, John	M
4	Halpin, Terry	M

Ranked:

memberNr	*art*	*rank*
2	kendo	3 kyu
3	judo	2 dan
3	karatedo	6 dan
4	judo	1 dan
4	karatedo	3 kyu

Figure 12.61 Martial arts club members and their ranks (if any).

Let's look at some examples using the populated schema in Figure 12.61. The Member table stores details about members of a martial arts club. The Ranked table indicates what rank (if any) members have achieved in what martial arts. Below black belt level, *kyu* grades are used, whereas *dan* grades are used for black belt ranks.

Consider the query: Who (member number and name) is a black belt in at least one art? We can formulate this in SQL with an existential subquery as shown here. This is also a correlated subquery since the subquery condition refers back to the outer table (Member). Because the outer table has a different name from the inner table (Ranked), there is no need to introduce an alias for it. The existential condition checks to see if the member has a row in the Ranked table with a dan rank.

```
select memberNr, memberName
from   Member                                  →   memberNr      memberName
where exists                                       ----------------------------------
   (select * from Ranked                               3         Halpin, John
     where memberNr = Member.memberNr                  4         Halpin, Terry
       and rank like '% dan')
```

The same query could have been formulated using a membership query instead, e.g.,

```
select memberNr, memberName
from   Member
where memberNr in
   (select memberNr from Ranked
    where  rank like '% dan')
```

To list the members who do *not* have a black belt, we simply substitute **not exists** for **exists** and **not in** for **in** in the previous queries. Would the following query also work for this?

```
select memberNr, memberName
from Member natural join Ranked
where rank not like '% dan'
```

As you no doubt realized, this fails because it also returns members with both a dan rank and a nondan rank. As a more complex example, suppose that club members are identified instead by combining their surname and firstname. The populated schema for this is shown in Figure 12.62.

Now consider the query: Who is not ranked in judo? The existential subquery solution is straightforward, matching on both surname and firstname:

```
select surname, firstname from Member
where not exists
   (select * from Ranked                       →   surname       firstname
    where surname = Member.surname                 ------------------------------
      and firstname = Member.firstname             Adams         Ann
            and art = 'judo')                      Brown         Ann
```

Member (surname, firstname, gender)

Ranked (surname, firstname, art, rank)

Member:

surname	*firstname*	*gender*
Adams	Ann	F
Brown	Ann	F
Halpin	John	M
Halpin	Terry	M

Ranked:

surname	*firstname*	*art*	*rank*
Brown	Ann	kendo	3 kyu
Halpin	John	judo	2 dan
Halpin	John	karatedo	6 dan
Halpin	Terry	judo	1 dan
Halpin	Terry	karatedo	3 kyu

Figure 12.62 Members have a composite reference scheme.

Although we can do this with a membership subquery, in SQL-89 the logic becomes more complex because we cannot use a column-name pair (here surname, firstname) as an argument for the **in** operator. In the following query, the surname column is compared using the **in** operator and the firstname is matched using a correlation:

```
select surname, firstname from Member
where surname not in
  (select surname from Ranked
   where firstname = Member.firstname
         and art = 'judo')
```

For such cases, the existential solution is easier to understand.

Yet another solution to this query is to use the concatenate operator to combine the name parts into a single string (recall that some dialects use "+" instead of "||"):

```
select surname, firstname from Member
where surname || firstname not in
  (select surname || firstname from Ranked
   where art = 'judo')
```

Further examples of correlated and existential subqueries are discussed in the context of set-comparison queries in the advanced SQL supplement (Appendix C).

Exercise 12.11

1. A population for the table scheme *Pupil* (pupilNr, surname, firstname, gender, [iq]) was provided in Figure 12.55. Formulate the following queries in SQL:
   (a) Which person has the highest IQ?
   (b) Who has the highest IQ for his/her gender?
   (c) List each male, female pair where both genders have the same IQ.

2. For the Member and Ranked tables discussed in Figure 12.62, formulate the following in SQL .
   (a) Who holds a kyu rank?
   (b) Which genders hold a kyu rank?
   (c) Who is not ranked?
   (d) Who does not hold a kyu rank?

3. The following schema refers to part of the student database considered in Exercise 12.1. Specify the following queries in SQL.

Student (studentNr, studentName, degree, gender, birthYr)

Result (studentNr, subjCode, [rating])

   (a) List the student number and name of each student who did not score any 7s. (Use a subquery.)
   (b) Same question, but use a join instead of a subquery. (*Hint*: 7 is the highest rating.)

12.12 SQL: Recursive Queries

Hierarchical data structures are found quite often in business. Some common examples are organizational structures with departments and subdepartments, and bill-of-material information involving assemblies and subassemblies. Recursion provides a powerful way to query such structures.

In SQL, recursive queries involve a three-phase process:

1. Create an initial result set (sometimes known as the *anchor*)
2. Recursively add results from further queries to the initial result set
3. Carry out a final query on the accumulated results

Recursion generally requires some termination condition. Here, the termination is implicit: the recursion ends when there are no more results to add in Phase 2. Particular SQL implementations may provide facilities for controlling the maximum depth of recursion to avoid infinite loops.

Creating the initial result set involves the concept of a *common table expression* (CTE), first introduced in SQL:1999. A common table expression can be imagined as a view definition with a scope that is limited to the query that immediately follows. The next chapter discusses *views*, which are basically named **select** statements stored in the system. The result of a view can be visualized as a table, although such a table is not normally stored in any physical sense. The basic syntax for a CTE is:

```
with cte-name [ (col-list) ] as cte-query-definition
query-statement
```

The col-list is not required as the cte-query-definition returns distinct columns. The query-statement immediately following the **with** clause uses the results of the cte-query-definition, referencing the cte-name. Attempts by any other query to refer to this CTE will meet with failure, unlike a view definition. This construct has other uses in SQL, but we're interested here in the role it plays in forming a recursive query. The anchor part of the recursive query can involve multiple subqueries with results combined using the standard SQL set operations (**union**, **intersect**, and **except**), but for simplicity we'll assume here that we have a single query.

Once the anchor results are formed, results from the recursion are added using SQL's **union all** operation. At each stage, the recursive part of the query takes the accumulated results as input, runs the query, and adds the results on to the result set. This continues until no more results are produced. Typically, this happens when there are no more unexplored nodes in the hierarchy.

A skeleton of the recursive CTE looks like the following:

```
with cte-name [ (col-list) ] as anchor-query-definition
union all
recursive-query-definition
query-statement
```

To make this more concrete, we'll look at a simple example. Figure 12.63 shows a schema for a table that holds hierarchical information about departments in an organization, together with a sample data population.

As shown in Figure 12.63(b), this organization has three departmental levels. We may want to write various queries that exploit the intrinsic parent–child relationships in the hierarchy, and recursive queries are a natural way to do this. Given the simple nature of the scenario in Figure 12.63, using recursive queries may seem overkill, but we can easily demonstrate the essential features.

Department (deptId, deptName, totalStaff, [parentDept])

Department:

deptId	*deptName*	*totalStaff*	*parentDept*
101	Management	3	?
201	Sales	2	101
202	Production	6	101
301	Sales - West	2	201
302	Sales - Central	3	201
303	Sales - East	4	201

(a)

(b)

Figure 12.63 (a) Table schema and data (b) Departmental hierarchy.

We'll begin with a simple level-by-level listing of the departments to illustrate how the result is produced. Here is the query:

```
with myCTE (deptId, deptName, parentDept, "level") as
(  select anchor.deptId, anchor.deptName, anchor.parentDept, 1 as "level"
   from Dept as anchor
   where parentDept is null
         union all
               select child.deptId, child.deptName, child.parentDept, "level" + 1
               from Dept as child join myCTE as parent
               on child.parentDept = parent.deptId
)
select deptId, deptName, "level" from myCTE
```

When executed, the query produces the result:

deptId	*deptName*	*level*
101	Management	1
201	Sales	2
202	Production	2
301	Sales - West	3
302	Sales - Central	3
303	Sales - East	3

Here the "level" column indicates the relative depth in the hierarchy, taking 1 as the top level ("level" is a delimited identifier here because it's a recognized keyword in SQL).

We've taken as our anchor the topmost level by specifying **where** parentDept **is null**, so the anchor tuple is <101, 'Management', null, 1>. At the next level, we're now looking for departments that have a deptId of 101. The **union all** operation adds the tuples <201, 'Sales', 101, 2> and <202, 'Production', 101, 2>. The join in the recursive part of the CTE can now add the departments under Sales. Production has no subdepartments and so the join produces no results. After adding the three Sales subdepartments there are no other possibilities for the join (because no department has a parent deptId of 301, 302, or 303), so the recursion ends.

When the **select** statement after the CTE is executed, it creates the result set for myCTE and then applies its own query, which in this case just extracts three of the columns produced by the recursive CTE. Of course, the final result depends on the query that operates on the CTE results and it does not have to list all of the CTE rows. For instance, the following recursive query finds the total number of staff below a particular point in the hierarchy—again, we've taken the top node, but a with simple change to the query defining the anchor we could have taken any point in the hierarchy.

```
with myCTE (deptId, totalStaff, parentDept) as
(  select anchor.deptId, anchor.totalStaff, anchor.parentDept
   from Dept as anchor
   where parentDept is null
         union all
               select child.deptId, child.totalStaff, child.parentDept
               from Dept as child join myCTE as parent
               on child.parentDept = parent.deptId
)
select sum(totalStaff) as totalStaff from myCTE
```

which produces the expected result:

totalStaff
20

Clearly, we could have produced this particular result using a much simpler query, but, in general, recursion allows us to frame queries on hierarchical structures that would be difficult (or clumsy) to achieve in other ways. Most commercial dialects of SQL support common table expressions and support recursion, but the details may vary from the outline we've given here.

12.13 SQL: Updating Table Populations

Generically, an "update" of the population of the database tables may involve any of three relational operations: insert a row into a table, delete a row from a table, or modify one or more values on a row. In SQL, the modify operation is called "update" so the term is sometime used in this restricted sense.

Let's first consider how to **insert rows** into a table. For this, SQL provides an **insert** statement in two basic forms. The first adds a *single row of data* to a table. Its SQL-89 syntax is:

```
insert into tablename [ (col-list) ]
  values (constant-list)
```

If no columns are listed, values for all columns must be given. Character strings must be delimited by single quotes. A null value may be entered as **null**. SQL-92 introduced the following form of the **insert** statement: **insert into** *tablename* **default values**. This adds a single row where each column is given its default value (possibly **null**). An error results if any column has no default. SQL:1999 added options for using constructors and overrides.

As a simple example, consider the table scheme *Employee* (empNr, surname, firstname, gender, [phone], [email]). Suppose a new female employee named "Eve Jones" is assigned the employee number 715. To add these details of the new employee to the table, we could use either

```
insert into Employee
   values ( 715, 'Jones', 'Eve', 'F', null, null )
```

or

```
insert into Employee ( empNr, surname, firstname, gender )
   values ( 715, 'Jones', 'Eve', 'F' )
```

Commercial systems also provide commands for fast bulk load of data from files (e.g., in delimited ASCII format). Data entry from users is typically via screen forms rather than directly to tables. Because of constraints on forms or between tables, a single row insert might require other data to be inserted in the same transaction. A transaction is a series of SQL statements, terminated by a **commit** statement:

```
commit [work]
```

If not committed, a transaction may be cancelled, thus:

```
rollback [work]
```

SQL:1999 allows inclusion of one or more *savepoints* within a transaction. Work can then be rolled back to a specific savepoint rather than rolling back the whole transaction. Releasing a savepoint acts like a tentative commit of the part of the transaction preceding the savepoint. Many SQL vendors do not yet support savepoints.

The most powerful form of the **insert** statement allows us to insert *multiple rows* into a table in one go. Its basic syntax is:

```
insert into tablename [ (col-list) ]
   select-query
```

This inserts the set of rows returned from the subquery. This has many uses, such as making copies of tables, or temporarily storing join results to speed up later queries. For example, consider the software retailer database schematized earlier in Figure 12.63. Now suppose we want to run many queries about sales of the database products. To facilitate this, we could create an appropriate table called "DBsales" and then populate it, thus:

```
insert into DBsales
   select *
   from  Invoice natural join LineItem natural join Item
   where category = 'DB'
```

For analysis purposes for a given period (e.g., 1 week), it may be acceptable to ignore updates to the underlying base tables. Queries can now be formulated on DBsales without the performance overhead of the original joins. This approach is used in data warehousing.

Now let's consider how to **delete rows**. In SQL, the **delete** statement is used to delete some or all rows from a table. After deletion, the table structure still exists (until the relevant **drop table** command). The basic syntax of the **delete** statement is:

```
delete from tablename
[ where condition ]
```

Here are two examples, based on our retailer database:

```
delete from Item            -- deletes all rows from the Item table

delete from Item
where category = 'DB'       -- deletes all the database item rows from Item
```

Next, let's consider how to **update rows** (i.e., change one or more values in existing rows). In SQL, the **update** statement is used to modify rows. Its basic syntax is shown. In SQL-89, expression must be a scalar expression, or **null** or **default**. A scalar expression may include column names, constants, and scalar operators.

```
update tablename
  set colname = expression  [, ...]
  [where condition]
```

As an example, consider the table scheme *Employee*(empNr, …, job, salary). The following statement gives all modelers earning less than $50 000 a 5% increase in salary:

```
update Employee
  set salary = salary * 1.05
  where job = 'Modeler'
    and salary < 50000
```

Because of the bulk updating power of **insert**, **delete**, and **update** commands, it is critical to have facilities for undo (rollback) and security (e.g., granting relevant access rights).

12.14 Summary

The data manipulation aspect of the relational model includes relational algebra and relational assignment (:=). Apart from the use of logical and comparison operators in expressing conditions, the algebra includes eight basic table operations. Comparison operators have precedence over logical operators, which have precedence over table operators. Table 12.3 listed the six priority levels of the operators: =, <>, <, >, <=, >=; **not**; **and**; **or**; selection (**where** …) and projection (*[ ]*); and finally **union** ∪, **intersection** ∩, difference − (termed **except** in SQL), Cartesian product ×, natural join ⋈ and division ÷. Parentheses may be used to override this order.

$\sigma_c(T) \equiv T$ **where** c
$\pi_{a,b,\ldots}(T) \equiv T[a,b,\ldots]$
$A \bowtie_c B \equiv (A \times B)$ **where** c

Figure 12.64 The eight basic table operations of relational algebra.

Operators on the same precedence level are left associative (evaluated in left to right order as they appear in an expression). Our priority convention is designed to minimize use of parentheses and is stronger than some other conventions in use. If in doubt, or purely for the sake of clarity, feel free to add extra parentheses.

The table operators are summarized visually in Figure 12.64. Many different notations exist for selection, projection, and joins. A notation common in academic journals is shown on the left, with our notation on the right.

The laws for the comparison and logical operators are well known. Table 12.17 sets out most of the main laws for the table operators. Some of these have not been discussed: their proof is left as an easy exercise. Here A, B, and C are tables, c is a condition, and p is a projection list of attributes. Various other distributive laws could be stated; recall, however, that *[ ]* does not distribute over $\cap$ or $-$.

A *relational DBMS* has the relational table as its only essential data structure and supports relational selection, projection, and join without needing to specify access paths. SQL is the dominant language used in relational DBMSs, and various versions have been standardized over the years, with the most recent being SQL:2003. Most commercial SQL dialects support all of SQL-89, some of the extra features in SQL-92 and SQL:1999, and some of their own special features.

Table 12.17 Main laws for the relational operators.

Commutative laws:	$A \cup B$	=	$B \cup A$
	$A \cap B$	=	$B \cap A$
Associative laws:	$A \cup (B \cup C)$	=	$(A \cup B) \cup C$
	$A \cap (B \cap C)$	=	$(A \cap B) \cap C$
	$A \times (B \times C)$	=	$(A \times B) \times C$
	$A \bowtie (B \bowtie C)$	=	$(A \bowtie B) \bowtie C$
Distributive laws:	$A \cup (B \cap C)$	=	$(A \cup B) \cap (A \cup C)$
	$A \cap (B \cup C)$	=	$(A \cap B) \cup (A \cap C)$
	$(A \cup B)$ **where** c	=	A **where** $c \cup B$ **where** c
	$(A \cap B)$ **where** c	=	A **where** $c \cap B$ **where** c
If A, B are compatible:	$(A \cup B)$ *[p]*	=	A *[p]* $\cup B$ *[p]* -- false for $\cap$
Exportation:	A **where** $c1$ **where** $c2$	=	A **where** $c1$ **and** $c2$
Delayed Projection:	A *[p]* **where** c	=	(A **where** c) *[p]*

Database objects such as tables and columns may be named using standard identifiers or delimited identifiers (allowing any characters inside double quotes). Database fields may be defined in terms of predefined data types (e.g., **varchar**(20)). SQL:1999 added user-defined types, row types, reference types, and arrays. The SQL **select** statement is used for queries and has the following basic syntax.

```
select * | [all | distinct] expression [[as] column-alias ] [,...]
  from table [[as] alias]
        [, | cross join table [[as] alias]
                | natural [inner | [outer] {left | right | full}] join table [[as] alias]
          | [inner | [outer] {left | right | full}] join table [[as] alias]
             { on condition | using (col-list) }
          | union join table [[as] alias]
            [,...]]
[where col-condition]
[group by colname [,...]
   [having group-condition]]
[union | intersect | except ...]
[order by col [asc | desc] [,...] ]
```

The **select** list chooses the result columns, the **from** clause declares the source tables and joins, the **where** clause is used to filter out rows, the **group by** clause forms groups of rows, the **having** clause is used to filter out groups, and the **order by** clause determines the order of rows.

A *subquery* is a parenthesized **select** query used inside an outer query. If the subquery includes a condition that refers to the outer query, it is a *correlated* subquery. A subquery after an **exists** quantifier is an existential subquery. An **exists** condition is true if the subquery returns a row, and false otherwise. The three most common forms of conditions with an embedded subquery are as shown, where Θ is a comparison operator.

```
expression [not] in (subquery)              -- membership subquery
expression Θ [all | some | any] (subquery)  -- [quantified] comparison subquery
exists (subquery)                           -- existential subquery
```

Table populations may be changed using **insert**, **update**, and **delete** statements, with the following basic syntax.

```
insert into tablename [ (col-list) ]
   {values (constant-list) | select-query}

update tablename
  set colname = expression  [, ...]
  [where condition]

delete from tablename
[ where condition ]
```

Chapter Notes

Relational algebra is covered in most database texts (e.g., Elmasri and Navathe 1994; Date 2000). Arithmetic operators, set functions, semijoin and other operators have been proposed to extend the relational algebra (e.g., Date 2000, sec. 6.7). As an alternative to the relational algebra, the relational calculus may be used (e.g., Date 2000, ch. 7). Codd (1990) proposed version 2 of the relational model, with 333 rules, but this has not achieved any significant adoption.

The SQL standards may be purchased online from *www.ncits.org*. The SQL standard is published by the International Standards Organization (*www.iso.org*) as ISO/IEC 9075. SQL:2003 is available now, with SQL:2008 to follow soon. For clear, thorough account of the SQL-92 standard, see Date and Darwen (1997), and for SQL:1999 see Melton and Simon (2002).

The original SEQUEL language, described in Chamberlin and Boyce (1974), was itself based on an earlier relational language called "SQUARE". Chamberlin (1998) provides a very readable coverage of DB2, with section 1.3 sketching an authoritative history of SQL, including an explanation for why SQL permits duplicate rows in base tables. For a comprehensive coverage of the latest version of Oracle, see Greenwald et al (2007). A good description of DB2 is given by Chong et al (2008). SQL Server 2005 is covered in many books: see, for example, Vieira (2007). *SQL Server Magazine* is a good source of technical articles on this product (*www.sqlmag.com*).

For a readable and practical discussion of many kinds of advanced queries in SQL, see the series of books by Celko (1999, 2000, 2004, 2005). An in-depth critique of SQL and an alternative theoretical foundation for object-relational databases are provided by Date and Darwen (1998).

This page intentionally left blank

13 Other Database Features

13.1 SQL: The Bigger Picture

13.2 SQL: Defining Tables
- Creating Tables
- Maintaining Referential Integrity
- Changing Table Definitions

13.3 SQL: Views

13.4 SQL: Triggers

13.5 SQL: Routines
- User Defined Functions (UDFs)
- Stored Procedures

13.6 More Database Objects
- Sequences
- Indexes
- Cursors

13.7 Transactions and Concurrency

13.8 Security and Metadata

13.9 Exploiting XML
- SQL/XML
- Using XQuery

13.10 Summary

13.1 SQL: The Bigger Picture

In the previous chapter we looked at SQL statements that retrieve data (**select**…) and modify data (**insert**…, **update**…, **delete**…). This set of statements is traditionally known as the *data manipulation language* (DML). We now turn our attention to other aspects of SQL. We'll look at statements used to define the various elements of a database schema, which uses a subset of SQL that is traditionally known as the *data definition language* (DDL).

Tables are, of course, fundamentally important to any schema, but we'll also discuss several other types of database object and the roles that they play in the realization of a conceptual schema. In particular, database objects such as *views* and *triggers* are often used to implement constraints and derivation rules that may have been defined initially at the conceptual level.

The widespread adoption of XML has been fuelled primarily by its use for the interchange of structured data. However, this has created a corresponding need for better integration between relational and XML data. We'll discuss the ways in which the SQL standard has been adapted and extended to add XML-related features and at how non-SQL data manipulation (such as in XQuery) can be accommodated in a relational framework.

13.2 SQL: Defining Tables

Creating Tables

The population of a *base table* resides in the database (unlike result tables and views, which we'll look at shortly). Base tables must be created before they are populated. In SQL-92 onward, base tables (and other aspects of the database) are defined within the scope of a *database schema* that is created using a **create schema** statement. The schema may be deleted using a **drop schema** statement. Commercial SQLs often provide a different syntax for creating a new database (e.g., **create database**).

In the SQL standard, tables are created using a **create table** statement. In addition to this, most DBMSs provide a graphical interface for users to create tables by entering text and selecting options. The main part of a table definition is a list of columns, identifying for each column the column name, its data type, and whether *nulls* are prohibited in the column (i.e., the column is mandatory). Constraints may also be declared for some of the columns.

The SQL-89 syntax of the **create table** statement may be summarized as shown. Here *literal* denotes a numeric or string constant, *col-list* denotes a list of one or more column names separated by commas, and *unique-col-list* denotes a column list that has collectively been declared to be the primary key or to be unique. If a constraint applies to a single column it may be declared at the end of the column definition. Multicolumn constraints must be declared at the end of the table definition. **Check** clauses are restricted to conditions that involve only one row of the table.

```
create table tablename (
  colname  data-type  [ not null ]
                      [ default { literal | null | user } ]
                      [ primary key | unique ]
                      [ references tablename ]
                      [ check (col-condition-on-same-row) ]
  [,...]
  [, primary key (col-list) ]
  [, unique (col-list) ]
  [, foreign key (col-list) references tablename [ (unique-col-list) ] [,...] ]
  [, check (table-condition-on-same-row) [,...] ] )
```

SQL-92 extended this syntax in many ways. Constraints may be named by inserting **constraint** *constraint-name* at the start of their definition. This facilitates meaningful error messages and dropping of constraints. **Check** clause conditions may range over many rows or tables, and general constraints may be declared using **create assertion** (mainly for intertable constraints). Foreign key constraints may include full/partial match conditions and referential actions. Constraints may be declared immediate or deferred. Macros for constrained data types may be declared with a **create domain** statement. SQL:1999 went much further. For example, an **of** clause may be used to declare a table to be of a user-defined type, previously declared as a row type with a **create type** statement. Subtables may be defined using an **under** clause to reference the supertable.

Commercial SQL dialects typically support all the SQL-89 functionality, possibly with some different syntax. They tend to support only some of the syntax from more recent versions of the standard. For example, constraint names and referential actions are widely supported, but assertions and unrestricted **check** clauses are rarely supported. Most vendors also provide extra features and syntax of their own.

As an example, let's see how to declare the relational schema for our software-retailer database. The graphical version of this schema is reproduced in Figure 13.1, along with the schema declaration in SQL-92 syntax. You can adapt this where needed to your SQL dialect of choice, and add constraint names as you see fit.

Notice the four intertable subset constraints (the equality constraint shows two of these). Unless we add these intertable constraints later, using an **alter table** or **create assertion** statement, the *order* in which we create the tables is important. If you wish to minimize the need for these later additions, you should wherever possible create the referenced table before the referring table. With our example, this entails creating the Customer table before the Invoice table and the Item table before the LineItem table.

Columns are optional by default, but may be declared mandatory using **not null**. SQL requires **primary key** columns to be declared not null. A **unique** constraint is assumed for primary keys, but may be explicitly declared for other columns or column lists, with the meaning that nonnull values are unique. For example, if the optional phoneNr column had instead been declared **unique**, an actual phone number may appear at most once in that column, but there may be many nulls in that column. Some commercial systems may consider null to be a value under these circumstances, in which case there could be at most one null value in the phoneNr column.

```
create schema SoftwareRetailer;

create table Customer (
  customerNr     smallint       not null    primary key,
  customerName   varchar(20)    not null,
  address        varchar(40)    not null,
  phonenr        varchar(10),
            unique (customerName, address) );

create table Item (
  itemCode       char(4)        not null    primary key,
  title          varchar(20)    not null    unique,
  category       char(2)        not null
                                      check (category in ('DB','SS','WP')),
  stock          smallint       not null    default 0,
  listPrice      decimal(6,2)   not null );

create table Invoice (
   invoiceNr     smallint       not null    primary key,
   customerNr    smallint       not null    references Customer,
   issueDate     date           not null,
   paidDatepaid  date );

create table LineItem (
   invoiceNr     smallint       not null    references Invoice,
   itemCode      char(4)        not null    references Item,
   qty           smallint       not null,
   unitPrice     decimal(6,2)   not null,
            primary key ( invoiceNr, itemCode ) );

create assertion each_invoice_has_a_line_item
  check ( not exists
                (select * from Invoice
                 where invoiceNr not in
                      (select invoiceNr from LineItem)) );
```

Figure 13.1 The relational schema for the software-retailer database.

Foreign keys are declared with a **references** clause, indicating the referenced table, optionally followed by the referenced column list, which must already be declared to be the primary key or unique. If no referenced column list is given, the primary key is assumed to be the target.

The equality constraint in the graphical version of the software-retailer schema is equivalent to two subset constraints in either direction between LineItem.invoiceNr and Invoice.invoiceNr—a referential cycle. The subset constraint from LineItem.invoiceNr to Invoice.invoiceNr is a foreign key constraint, since the target is a primary key. So this may be declared using a **references** clause as shown. However, the subset constraint from Invoice.invoiceNr to LineItem.invoiceNr is *not* a foreign key constraint, since its target is just one part of a composite primary key. In SQL-92 onward this constraint may be declared using an **assertion**, as shown. In SQL dialects that do not support assertions (such as SQL Server), assertions can be coded instead using triggers or stored procedures. We'll look at referential integrity in more detail shortly.

A **check** clause specifies a condition to be satisfied. Value constraints are often implemented using **check** constraints. A **check** constraint is violated if and only if its condition evaluates to false (not true and not unknown). Some people find this unintuitive when compared with the **where** clause in a **select** statement, which must evaluate to true (not false and not unknown) in order for a row to be included in the result set.

Default options may be declared with a **default** clause. If a user inserts a row without specifying a value for a column, then the system automatically inserts the default value defined for that column. In SQL-89, the only allowed defaults were a literal value, **user** or **null**. In the absence of a default clause for an optional column, **null** is assumed to be the default. SQL-92 expanded the possible defaults to include the nullary functions: **current_user**, **session_user**, **system_user**, **current_date**, **current_time**, **current_timestamp**. SQL:1999 added **current_role**, **current_path** and any implicitly typed value specification (e.g., to deal with collection types).

Here is a suggested procedure for deciding where to declare constraints in SQL that you may find useful if you want to minimize keystrokes:

```
If a constraint applies to only one table
            or is a foreign key constraint
then if it applies to only one column
      then declare it with the column
      else declare it at the end of the table definition
else declare it with an assertion (or trigger etc.)
```

SQL:2003 added another column option to deal with the creation of *identity columns*, for which the column syntax is

```
colnamedata-type   [ generated { always | by default } as identity ]
                        [ ( start with start-value | increment by increment-value |
                          { maxvalue value | no maxvalue } | { minvalue value | no minvalue } |
                          { cycle | no cycle } [ ,... ] ) ]
```

There can be at most one identity column in a table and its data type must be numeric with a scale of zero (e.g., integer). When a row is inserted into the table, the system gen-

erates the next value in sequence, as defined by the rather complicated set of options. If **always** is specified, it is an error to try to insert a value into the identity column. If **by default** is specified, the system will generate a value if none is supplied, otherwise it will use the supplied value. If a row is subsequently deleted, its identity value is not reused.

The default value for both **start with** and **increment by** is 1, unless explicitly specified otherwise. The increment value can be positive or negative. The maximum and minimum values possible for a sequence are implementation dependent, and these will be adopted unless a specific **maxvalue** or **minvalue** is defined. The sequence stops and no more values will be generated once **maxvalue** (counting up) or **minvalue** (counting down) is reached or exceeded. Depending on the relationship between the start value and increment size, the sequence may or may not land exactly on a **maxvalue** or **minvalue**. A sequence will not cycle back and duplicate values already generated unless the **cycle** option is specified. If **cycle** is specified the sequence will stop at **maxvalue** and reset to **minvalue** if counting up, and the converse if counting down. The start value can be anywhere in the range defined by these two values.

Identity columns are supported in most implementations, although not necessarily with the standard syntax. The guarantee that the generated value will be unique within a table makes them attractive to use as a primary key. In cases where the natural primary key for the table is clumsy, for example, a combination of multiple **varchar** columns, an identity column is often introduced as a surrogate key.

The SQL:1999 standard allowed the creation of tables based on existing table definitions. This capability was refined in SQL:2003 to provide more precise control over the columns that should be included or excluded in the new table. The basic syntax is as follows.

```
create table new-tablename like existing-tablename
[ { including identity | excluding identity } |
  { including defaults | excluding defaults } |
  { including generated | excluding generated } [,..] ]
```

The result of the **create table** … **like** … statement is an empty table with the same column names, data types, and nullability as the original table. No dependency is established between existing and new tables. The options specify what action should be taken for "special" columns.

Maintaining referential integrity

Let's now discuss the notion of *referential integrity* in more detail. Recall that referential integrity is a special case of a subset constraint. The relational schema outline here shows a typical case, where *S.a* is a foreign key referencing *R.a*. Basically the subset constraint is satisfied if each value in *S.a* either occurs as a value of *R.a* or is null.

The subset constraint may be violated in any of the following four ways:
(1) insert a row into *S* with a value for *S.a* that is not a value in *R.a*
(2) update a row of *S* to replace the value for *S.a* with a value that is not in *R.a*
(3) delete a row from *R* where the value of *R.a* is also a value in *S.a*
(4) update a row of *R* to replace a value of *R.a* that is also a value in *S.a*

Instead of declaring the subset constraint as "*S.a* **references** *R.a*", we could enforce the constraint by writing trigger code to handle each of these four cases. A *trigger* is basically a chunk of procedural code that is defined for a single table to carry out some action whenever a specified kind of event (insert, delete, or update) occurs for that table. We'll look at triggers in more detail a little later in this chapter. SQL:1999 includes a full procedural trigger language with the power of a typical third generation programming language.

Although triggers can be very useful for enforcing rules (such as constraints or derivation rules), if a declarative version of the rule is supported by the system, then the declarative approach is normally preferred. First, the declarative version of the rule is more easily understood, and hence easier to validate with domain experts. Second, the declarative version is shorter and easier to formulate than the procedural version. For example, the declaration "*S.a* **references** *R.a*" is trivial compared to the code for the corresponding triggers. Finally, the system can often do a better job of optimizing declarative code than procedural code. Having said that, there are still many cases where triggers or stored procedures provide the only way of getting the system to carry out some action. Until recently, referential actions fitted into this category.

A *referential action* is an action to be taken in response to a violation of a referential integrity constraint. In SQL-89, if an attempted update (insert/delete/update) violates a foreign key constraint, the update is simply rejected—in other words, no action is taken. In SQL-92, this "no action" default may be replaced for update and delete operations by other options by declaring the appropriate referential action(s) at the end of the **references** clause. SQL-92 also allows a **match** clause, with **full** or **partial** options for finer control over null matching. SQL:1999 added the **simple** option to allow the default option for matching to be explicitly declared, but otherwise uses the same syntax for the **references** clause, the full syntax of which is

references *table* [(*col-list*)]
[**match** { **full** | **partial** | **simple** }]
[*delete-action* [*update-action*] | *update-action* [*delete-action*]]

where *delete-action* has the syntax:

on delete { **no action** | **cascade** | **set null** | **set default** }

and *update-action* has the syntax:

on update { **no action** | **cascade** | **set null** | **set default** }

If an attempt is made to change the database with an insert, delete, or update operation that would violate the referential constraint, the default **no action** option rejects the change. The other three referential action options accept the change, but also take some compen-

sating action to keep the database consistent. The **cascade** option propagates the change to all referencing foreign key values; **set null** sets the relevant foreign key values to null; and **set default** sets the relevant foreign key values to the column default. For example, consider the following relational schema:

Here we have a chain of two subset constraints. Suppose "**on delete cascade**" is declared for both constraints. If we attempt to delete the model "Ford T" from the CarModel table, the delete is accepted. But if there are any rows in the Car table with the model "Ford T", they will be deleted also, which in turn causes any rows in the Drives table with matching registration number to be deleted also.

SQL-92 introduced the **match** operator as a Boolean operator to test whether a tuple belongs to a set of tuples, i.e., does $(a_1, \ldots, a_n) \in \{(b_1, \ldots, b_n), \ldots\}$? Except for its Boolean nature, it is a generalization of the **in** operator and may be used in search conditions of the form (a_1, ..., a_n) **match** [**unique**] [**full** | **partial** | **simple**] (*subquery*). The **unique** option requires the tuple to match exactly one row. By default, or if **simple** is declared, the match operation returns True if all values $a_1..a_n$ match or *some* are null; otherwise it returns False. If full is specified, the match operation returns True if all values $a_1..a_n$ match or *all* are null; otherwise it returns False. If partial is specified, the match operation returns True if all *nonnull* values match; otherwise it returns False. The match operator may also be used in a **references** clause in a similar way.

Although the **references** clause syntax for SQL-89 is now widely supported, most SQL dialects currently have less support for the further referential syntax in SQL-92 or SQL:1999. For example, SQL Server 2005 does not support the **match** operator.

Changing Table Definitions

After a base table has been created and even populated, we may wish to change its definition. This was not possible in SQL-89. However, SQL-92 allows a table scheme to be changed using an **alter table** statement. SQL:1999 extended this slightly to cater for column scoping. Each change requires a separate statement. The basic syntax is as follows:

alter table *tablename*
add [**column**] *col-defn*
| **alter** [**column**] *colname* **set** *default-defn* | **drop default**
| **drop** [**column**] *colname* **restrict** | **cascade**
| **add** *table-constraint-defn*
| **drop constraint** *constraint-name* **restrict** | **cascade**

Dropping a column is not allowed if **restrict** is specified and the column is referenced in constraint or view definitions. The **cascade** option causes the other definitions to be dropped too. Commercial SQLs typically do not support **restrict** or **cascade** options for dropping columns or constraints.

Although the standard forbids the data type of a column to be altered, some dialects allow a column's data type to be changed to a "larger type", e.g., change char(2) to char(4) for a roomNr column. Some alter actions are not supported by many vendors, e.g., **drop column**. To simplify automated table creation, some vendors prefer to add most table constraints in **alter table** statements. The following examples are self-explanatory:

```
alter table Person
   add column email_address varchar(20)

alter table Person
   add foreign key starsign references starsignPeriod
```

SQL-92 allows a table (including its population) to be deleted from the database using a **drop table** statement. Although this is not allowed in SQL-89, all vendors support it. The syntax is:

```
drop table tablename  cascade | restrict
```

Dropping a table is not allowed if **restrict** is specified and the table is referenced in constraint or view definitions. The **cascade** option causes the other definitions to be dropped too. Most commercial SQLs do not require cascade/restrict here.

As a first step on the way to user-defined types, SQL-92 introduced statements for creating, altering, and dropping syntactic *domains*. Once a domain has been defined, it can be used in column definitions. For example:

```
create domain Money as decimal ( 9, 2 )
create domain Gendercode as char
constraint ValidGendercode check ( value in ( 'F' , 'M' ) )
   create domain Centimeters as smallint check ( value > 0 )
   create domain Kilograms as smallint check ( value > 0 )

create table Employee (
   empNr     smallint      not null  primary key,
   gender    Gendercode    not null,
   salary    Money         not null,
   tax       Money,
   height    Centimeters,
   weight    Kilograms )
```

These domains are merely macros or abbreviations for their definitions. So nonsense queries such as "**select** * **from** Employee **where** height = weight" are still legal. However SQL:1999 does support true user-defined types, with strong type checking. For example, if we replace the previous domain definitions for Centimeters and Kilograms by the following type definitions, the condition "height = weight" generates an error. If you ever want to compare just the numbers rather than the height and weight entities, you can use the

cast function to convert to compatible data types. The declaration of "**final**" means the type cannot have proper subtypes.

```
create type Centimeters   as smallint   final
create type Kilograms     as smallint   final
```

Centimeters and Kilograms are defined in terms of a predefined scalar type, but are treated as distinct from their base type, and are said to be "*distinct types*". SQL:1999 also allows user-defined *structured types* involving many attributes. In SQL:1999, *array types* and *row types* may be introduced within a table definition. The type definition features in SQL:1999 are vast, but not yet widely supported. There is no space here to discuss them in detail, but here is an example:

```
create table Voter (
   voterId        smallint                primary key,
   preferences    smallint array[3],
   address        row( streetAddress      varchar(20),
                       cityname           varchar(20),
                       zip                row( main char(5), ext char(4))))
```

13.3 SQL: Views

In addition to defining base tables, SQL also allows us to define **views**, or "virtual tables", which are basically *named, derived tables*. Their definition is stored, but their contents are not. By means of the view definitions, queries and updates on views are automatically translated into equivalent queries/updates on base tables. Hence, unlike working tables that you might create to temporarily store results, views are always up to date.

Views have many uses. They may be used for security reasons by allowing a particular group of users access to views but not to their underlying base tables. They can also be used to hide unwanted details, simplify queries or updates, implement derivation rules, and allow base table reconstruction with minimum impact. Their main limitations are that some views cannot be updated, and queries on views sometimes lower performance. Further discussion on the implemetation of derivation rules using views and generated columns can be found on the Web site that accompanies this book.

A view is created using the **create view** statement. The SQL-89 syntax is as follows:

```
create view viewname [ (col-list) ] as
   select-query
   [ with check option ]
```

Since the result of a view is a derived table, the **select** statement in a view can reference other views as well as base tables. In fact, from the perspective of a **select** statement, there is little difference between a query on a table and a query on a view.

SQL-92 allows the check option to be qualified as local or cascaded, and SQL:1999 allows views to be recursive and to be based on user-defined types. Although SQL-89 does not allow views to be dropped, SQL-92 does, using the syntax:

drop view viewname { **restrict** | **cascade** }

If the **restrict** option is specified, the view is not dropped if it is referenced in another view or a constraint. If the **cascade** option is specified, the view is dropped as well as any views or constraints that reference it. Many dialects, including SQL Server, do not yet support these two options. Although the standard does not provide an **alter view** statement, some dialects, including SQL Server's T-SQL, do. One advantage of altering a view, instead of dropping it and recreating it, is that any privileges associated with the view are preserved.

Some restrictions apply to the **select** query used to define a view, although details depend on the SQL dialect. Typically no **order-by** clause or any of the **union**, **intersect**, or **except** operators are allowed. An *updatable view* can be used to insert, delete, or modify rows. Many more restrictions apply to updatable views (see later).

Consider the table scheme *Academic*(empNr, empName, dept, [gender], grade) used to store details about academic employees. Sample data are shown in Table 13.1. The dept column stores the code for the academic's department (e.g., CS = Computer Science, MA = Mathematics). The grade column is the academic's rank (e.g., Lec = Lecturer, Prof = Professor).

The following statement may be used to create a view about just the female computer scientists. With this definition stored, we may now act as if a table corresponding to the view really exists, as shown in Table 13.2.

```
create view FemaleCompScientist ( empNr, name, rank ) as
   select empNr, empName, grade
   from   Academic
   where dept = 'CS' and gender = 'F'
```

For example, to retrieve the female professors of computer science, we request:

```
select  empNr, name
from   FemaleCompScientist
where rank = 'Prof'                    →    Rolland, C
```

Table 13.1 A base table about academics.

Academic:

empNr	*empName*	*dept*	*gender*	*grade*
101	Aldave, L	CS	F	Lec
102	Jones, E	MA	?	Lec
103	Codd, E	CS	M	Prof
104	Rolland, C	CS	F	Prof

Table 13.2 A view.

FemaleCompScientist:

empNr	*name*	*rank*
101	Aldave, L	Lec
104	Rolland, C	Prof

When this query is processed, the view definition is accessed and used to automatically translate the query into the following equivalent query on the underlying base table. The base table query is then processed to produce the result:

```
select empNr, empName
from   Academic
where dept = 'CS' and gender = 'F'
and grade = 'Prof'                    →    Rolland, C
```

That view was based on just a projection and selection from a single table. Let's look at a harder case based on a join. Consider the table scheme *ParentOf* (parent, child). A sample population is shown in Figure 13.2. For simplicity, assume people are identified by their first name.

Now consider the query: Who are the grandparents of Fred? To make it easy to answer the question, Figure 13.2 includes a graph in where a dotted arrow from one node to another means the first is the parent of the second. Starting at Fred, we can easily find Chris as Fred's only recorded parent, and then move up to Chris's parents. So the answer is the set {Ann, Bill}. We can also follow this procedure using the table. The query is easy to formulate in SQL using a membership subquery, thus:

```
select parent from ParentOf
where child in
   ( select parent from ParentOf
   where child = 'Fred' )
```

But now suppose we often wanted to list the grandparents of different people on different occasions. We could save ourselves some work by defining a view for the *GrandparentOf* (grandparent, grandchild) relation thus:

```
create view GrandparentOf ( grandparent, grandchild ) as
   select A.parent, B.child
   from  ParentOf as A join ParentOf as B
   on A.child = B.parent
```

ParentOf:

parent	*child*
Ann	Chris
Ann	David
Bill	Chris
Bill	David
Chris	Ernie
Chris	Fred
David	Gina
Gina	Helen

Figure 13.2 Who are the grandparents of Fred?

Table 13.3 A view based on a join.

GrandparentOf:

grandparent	*grandchild*
Ann	Ernie
Bill	Ernie
Ann	Fred
Bill	Fred
Ann	Gina
Bill	Gina
David	Helen

This view definition involves a projection on a self-join, where we match the child of the grandparent to the parent of the grandchild. Table 13.3 shows the virtual population, which could be listed using the command: **select** * **from** GrandParentOf.

The grandparents of Fred can be listed using **select** * **from** GrandParentOf **where** grandchild = 'Fred'. The grandparents of any other person can be requested by using their name in place of 'Fred'. Although the view is obviously useful for queries, can it also be used for updates?

As discussed in the previous chapter, rows can be inserted into base tables using an **insert** command. Suppose we try to insert a row into the view, thus:

```
insert into GrandparentOf
   values ( 'Bernie', 'Selena' )
```

An update on a view must be translated into an equivalent update on the base table(s). But since we don't know the intermediate parents (i.e., Selena's parents) the best we could hope for in this case is to add the following two rows into the ParentOf table:

```
ParentOf:    parent      child
             -------------------
             Bernie        ?
               ?         Selena
```

But this is unacceptable because it doesn't really tell us that Bernie is a grandparent of Selena. In fact, the two rows would not satisfy the view definition anyway, since they fail the join condition A.child = B.parent. Since both values are null here, the condition evaluates to unknown, and the row ('Bernie', 'Selena') is excluded from the view. So the GrandparentOf view is *nonupdatable*.

SQL-92 places many restrictions (more than logically required) on what kinds of view are updatable. In particular, an updatable view must contain no joins, no union (or intersect or except), no **distinct** option, and no **group by** clause. SQL Server gets around many such restrictions by use of **instead-of** triggers on views (see later).

Even with no joins, similar problems with nulls arise if the view excludes a column used in its search condition. For example, suppose we try the following insertion on our FemaleCompScientist view:

```
insert into FemaleCompScientist
   values ( 105, 'Bird, L', 'Lec' )
```

If we allow this update, is 'Bird, L' included in the result? Given our earlier view definition, if the update were accepted it would translate into the following row being inserted into the Academic table: (105, 'Bird, L', **null**, **null**, 'Lec'). The new row fails the view definition since the condition (dept = 'CS' **and** gender = 'F') evaluates to unknown. So if the update is accepted, it is not part of the view!

It is better to reject any update that fails to satisfy the view definition. This is done by appending "**with check option**" to the view definition. For example, the following view may be used for computer science professors:

```
create view CSProfessor ( empNr, name, dept, rank ) as
  select empNr, empName, dept, grade
  from Academic
  where grade = 'Prof' and dept = 'CS'
        with check option
```

The following attempts to update this view would now have the results shown.

```
insert into CSProfessor
  values ( 106,'Wirth N','CS','Prof' )        →   accepted

insert into CSProfessor
  values ( 107,'Russell B','MA','Prof' )      →   rejected

update CSProfessor
  set rank = 'Lec'
  where empNr = 103                           →   rejected
```

The type of view we have considered so far is basically a predefined query on one or more base tables. The result of the query is not stored in the view, only the query definition. A subsequent query on the view is combined with the query in the view definition, and the composite query is run against the base table(s). This ensures that the result is always up to date with the content of the base tables, but may be inefficient if a view is queried frequently.

Another option is to execute the query defined for the view and cache the result. Now the view represents an actual data structure held in physical storage, not just the definition of a data structure. This is known as a *materialized* view. Such views have obvious advantages in efficiency, but subsequent changes to data in the base table(s) can lead to the view becoming increasingly unrepresentative of the "real" data. Commercial systems that support materialized views normally provide some mechanism for periodically refreshing them.

SQL:2003 added a new option to the **create table** statement to allow for the creation of tables based on the result of a **select** statement. This provides functionality somewhat similar to a materialized view. The basic syntax is:

```
create table tablename as select-statement
{ with data | with no data }
```

If the intention is to populate the table with data at the time of its creation, the **with data** option must be specified. If it is not, an empty table will be created with the appropriate

columns but no data. A better description of the result is a *materialized query table* because no dependency is established between the new table and the underlying query expression. After the table is initially populated, updates to the tables in the query expression will not automatically be reflected in the new table.

Exercise 13.3

1. Specify SQL DDL statements to create the following relational schema.

2. A conceptual subschema, its relational tables, and a sample population are shown.

ParentOf: (≤ 2)

parent	*child*
Ann	Chris
Ann	David
Bill	Chris
Bill	David
Chis	Ernie
Chris	Fred
David	Gina
Gina	Helen

BrotherOf:

brother	*sibling*
Ian	Ann
John	Ann
Ken	Bill
Bill	Ken
David	Chris
Ernie	Fred
Fred	Ernie

(a) Complete the following derivation rule: $Person_1$ is an uncle of $Person_2$ **if** …
(b) Specify this rule as a view *UncleOf* (uncle, child) in SQL.
(c) Formulate the query "Who is an uncle of whom?" in SQL, using the view.
(d) Formulate (c) in SQL without using the view.
(e) Formulate in SQL: Who is recorded as having a parent but not an uncle?

3. The following schema shows part of a database that holds information about repair technicians and the types of assemblies that they are competent to repair. Different technicians may be competent at repairing different assemblies. Assemblies may be parts of other assemblies, unless, of course, they are a top level assembly.

(a) Create the tables for this schema in the most convenient SQL form. Assume that empNr is some unique integer, assemblyCode is some unique 10-character string, complexityNr is an integer, and firstName, lastName, and certificationRef are varchar strings of appropriate length. isTopLevel and hasReport are flags that reflect a Boolean condition. For this exercise, specify certificationRef simply as a unique column.
(b) Is the order of creation of the tables important?
(c) Both of the Technician and Assembly tables could represent hierarchies (how can you tell?). What would be the distinguishing characteristics of a node at the top of a Technician hierarchy and a node at the top of an Assembly hierarchy?

4. Given the schema defined in Q1, we want to identify technicians with high level skills.
   (a) Define a view for all technicians that are competent to work on an assembly of complexity 6 or higher. This should list only the Technician's firstName and lastName, assemblyCode, and complexityNr.
   (b) Is this view updateable? Explain your reasoning.

5. Do you agree or disagree with the following statements about ordinary SQL views? Explain your reasoning.
   (a) Views create temporary tables that can be queried using select statements.
   (b) A select statement associated with a view may not reference other views.
   (c) Defining a large number of views will create additional demands for data storage.
   (d) A view defined on exactly one base table is always materialized.
   (e) A view can be queried in much the same way as a table, but updates are more restricted.

13.4 SQL: Triggers

Triggers provide yet another way to change table populations. Although not included in the standard until SQL:1999, triggers have been used by commercial dialects for many years. As well as enforcing constraints, triggers may be used to perform other actions, including computation of derived columns, maintaining audit or summary data in other tables, and initiating external actions such as sending email.

A trigger specifies a set of SQL statements that will be executed automatically when a given event takes place on a table (or, in some circumstances that we'll discuss shortly, a view). The event causing the execution of the trigger can be an attempt to *insert* new

rows, *delete* existing rows, or *update* existing rows in the table. The trigger actions can be specified to take place either immediately *before* or immediately *after* the related operation (insert, update, or delete) takes place.

Multiple triggers can be defined on a table. All **before** triggers will execute before any operation is carried out on the table, including constraint checking. This provides an opportunity to examine the action that is about to take place and to modify it if necessary. All **after** triggers will execute after constraint checking has been performed and the action has taken place, but before any changes are committed. This provides an opportunity to examine changes that have been made and to rollback the associated transaction if required. Different vendors may provide facilities for specifying the trigger sequence if there are multiple **before** triggers, and similarly for **after** triggers.

Within the body of a trigger it's possible to reference the contents of one or two *transition tables*. These contain rows that are being affected by the trigger event. The columns of the transition tables match the column definitions of the table on which the trigger is based. For a delete trigger, there is an *old* transition table containing the rows being deleted. For an insert trigger there is a *new* transition table containing the rows being inserted. An update trigger has both an *old* transition table containing the row data prior to the update and a *new* transition table containing the row data after the update. In the update case, every row in the *old* transition table will have a corresponding row in the *new* transition table. Transition tables only exist for the duration of the trigger and cannot be accessed from outside the trigger.

The trigger action can be specified to take place either at the *row level* or at the *statement level*. In the former case the action takes place for each row of the transition table. In the latter case the action occurs just once, when the trigger event takes place. Depending on whether the trigger has been defined at the row level or the statement level, the trigger code can access *transition variables* that are either individual rows from the old/new transition tables or the complete transition tables.

In SQL:1999, the statements for creating and dropping triggers have the following syntax:

```
create trigger triggername
  { before | after } { insert | delete | update [ of col-list ] } on tablename
  [ referencing     { old | new } [row] [ as ] correlation-name |
                    { old | new } table [ as ] table-alias          [, ...] ]
  [ for each { row | statement } ]
        [ when ( condition ) ]
        triggered-SQL-statement

drop trigger triggername
```

Although most SQL dialects support triggers, they do not necessarily conform to the standard syntax just shown for creating triggers. For example, SQL Server does not currently support **before** triggers.

In addition to **before** and **after** triggers, SQL:2003 introduced a third kind of trigger: the **instead of** trigger. As the name suggests, the trigger action is carried out *instead of* the triggering event. An important application for **instead of** triggers is to intercept an attempt

to modify a view and replace it with an appropriate set of modifications to the table(s) on which the view is based.

As an example, we can return to the *FemaleCompScientist* view defined earlier.

```
create view FemaleCompScientist ( empNr, name, rank ) as
   select empNr, empName, grade
   from   Academic
   where dept = 'CS' and gender = 'F'
```

Recall that the base table, Academic, had columns (empNr, empName, dept, [gender], grade). An attempt to insert a new row into the view, such as

```
insert into FemaleCompScientist
   values ( 105, 'Bird, L', 'Lec' )
```

would fail because we have no values for the dept and gender columns: null entries would violate the view definition. However, we can use an **instead of** trigger to intercept the attempted insertion, and instead make the correct insertion in the base table, as follows:

```
create trigger AddFCS
instead of insert on FemaleCompScientist
referencing  new row as Inserted
   for each row
   insert into Academic ( empNr, empName, grade, dept, gender )
         select empNr, empName, rank, 'CS', 'F' from Inserted
```

We obtain the appropriate values for empNr, empName, and grade from the empNr, empName, and rank columns in the *new transition variable*. We already know from the view definition that the values for dept and gender must be 'CS' and 'F', respectively. This allows us to correctly insert a new row into the Academic base table. A subsequent **select** command on the view would show that the insert has apparently succeeded, with the new row now appearing in the view.

Exercise 13.4

The following schema shows part of a database that holds information about repair technicians and the types of assemblies that they are competent to repair. Different technicians may be competent at repairing different assemblies.

1. The foreign key reference from Competence to Technician would normally be implemented by a **references** clause. Assume that this is omitted, and create the equivalent functionality using SQL triggers to prevent violations of referential integrity (only **restrict** is required).

2. Values for certificationRef must be unique if they exist, but some implementations of SQL do not allow multiple nulls in a unique column. Instead of specifying the certificationRef column as unique, write a trigger that ensures that nonnull values are unique but allows multiple nulls.

3. From time to time the complexityNr of an assembly may be adjusted up or down, but only by a single level. For example, a complexityNr of 6 may only be changed to 5 or 7. Write an update trigger to limit changes to complexityNr to at most one level on a single update.

4. Reports are produced detailing faults that have been found on assemblies in the field. These are stored in a table with the schema *Report* (reportId, assemblyCode, reportDoc), where reportId and reportDoc have the data types **int** and **clob**, respectively. A report can be deleted if it is subsequently found that it does not reflect a genuine fault. For efficiency, we want to set a flag in the Assembly table for any assembly that has a report. Write a trigger for the Report table that will maintain the hasReport flags in the Assembly table.

13.5 SQL: Routines

The SQL standard uses the generic term *routine* to describe a body of code that can be invoked from some other code—this covers functions, procedures, and the methods of user-defined data types.

User Defined Functions (UDFs)

We'll look first at functions, which, in simple terms, are bodies of code that return a result. The SQL standard defines a large number of built-in functions for calculating logarithms, generating random numbers, transforming character strings into all uppercase or all lowercase, and so on. These can be used wherever appropriate in SQL code. It's also convenient to be able to define new functions to encapsulate pieces of code that are likely to be reused. In SQL, functions can return scalar (single) values of a given data type. The SQL:2003 standard extended the concept to include functions that return tables.

The basic SQL:2003 syntax for creating a function is as follows.

```
create function functionname ( [ parameter-list ] )
  returns { data-type | table col-name data-type [,…] }
  function-body
```

The parameter-list is a (possibly empty) comma-separated list of parameter name, parameter type pairs. The data-types for the returned value can be any legitimate SQL type. The function body can be composed of any valid combination of SQL statements, but must include a **return** clause that returns a parameter of the type specified in the **returns** clause. UDFs can be removed by using **drop function** *function-name*.

Commercial DBMSs may place other restrictions. For example, in SQL Server, UDFs cannot use SQL statements that would produce different results if the function was invoked repeatedly with the same parameters. Examples of such statements include getting the current time, and generating random numbers. SQL Server also prohibits side effects in UDFs so, for instance, a function cannot be used to update a base table.

The following example shows a scalar UDF for SQL Server that takes a currency amount (defined as a decimal value) and a currency symbol as parameters, and returns a string consisting of the currency symbol concatenated with a string version of the amount. SQL Server uses "+" for string concatenation, as opposed to the standard "||".

```
create function moneyString ( @amount decimal( 9, 2 ), @symbol char ( 1 ) )
  returns varchar ( 11 )
  return @symbol + cast ( @amount as varchar(10) )
```

Once defined, functions can generally be used at any point where a value of the return type could be used. So taking the example just given, we could use a call to moneyString( ) at any point where a varchar(11) value would be legitimate.

Stored Procedures

Stored procedures, commonly referred to as "sprocs", are bodies of code that are explicitly called from other code. In this section we'll assume that the body of the procedure is written in SQL, but it's possible for the procedure actions to be written in some other language, such as 'C', and then called from SQL. Stored procedures can (and often do) have side effects, such as updating tables during their execution. Encapsulating code in a stored procedure offer several potential benefits.

Modularity. Code in a stored procedure is readily reusable in other code. A well-maintained library of stored procedures can be a valuable resource when developing new applications. Modularity also helps to ensure consistency. For example, a business rule can be implemented once in a stored procedure and called as needed, instead of having the same rule implemented in perhaps slightly different ways in different places. The same argument applies to maintainability. If a change is needed, it need only be applied in one place. As long as the previous interface is maintained, other programs that call the stored procedure need not be aware of any changes that have been made.

Security. Users of a stored procedure are offered only the functionality that the procedure provides. Limiting access in this way drastically reduces the possibility of damage to the database, whether deliberate or accidental. The authority given to users of a stored procedure can be different from the authority to access any underlying tables, so any actions can only be taken in the context of the procedure. For example, all modifications to an "Orders" table can be channeled through a stored procedure that imposes business rules and other constraints on what changes can be made, instead of giving users direct access to the base table.

Performance. Stored procedures are typically compiled, optimized, and cached ready for use when needed. The details vary from one implementation to another, and extensions to the basic syntax are usually provided to address issues such as the need to redo compilation and optimization if the usage circumstances change. Stored proce-

dures can also reduce network traffic: instead of constructing a query in an application and passing the resulting SQL across a network, only the parameters need be passed to a stored procedure.

The basic syntax for creating and dropping a stored procedure is as follows.

```
create procedure procedurename
  ( [ { in | out | inout } parmname parmtype [,...] ] )
      procedure-body

drop procedure procedurename
```

A stored procedure takes zero or more parameters. Each parameter can be specified as **in** (external values are provided to the procedure), **out** (the procedure makes its results available externally), or **inout** (the input parameter may be modified by the procedure for external access). As usual, commercial systems may vary from the standard syntax. For example, in SQL Server any parameters of a stored procedure are considered as **in** parameters unless they are explicitly denoted as **out**, and there is no **inout** option.

The procedure body is made up of any legal combination of SQL statements. Stored procedures may make calls to other procedures, and a procedure may call itself recursively. An implementation may place restrictions on the number of levels of nesting permitted, for example, SQL Server allows nesting up to 32 levels deep. Stored procedures can also be called from within a trigger, which gives both the advantage of automatic invocation (when the relevant trigger event occurs) and the modularity offered by a stored procedure. For instance, we may want to apply the same constraint under several different circumstances, which we could do by having several different triggers call the same stored procedure.

One SQL statement that requires some attention in a stored procedure is the **return** statement. Once a **return** is encountered, the stored procedure is exited immediately, and so a **return** is often associated with a branch of a test condition, along the lines of: "*if x is the case then return, otherwise, …*". Although not necessarily enforced by an implementation, a recommended practice is to use a value associated with the **return** statement to indicate status. Typically, a return code of zero would indicate "all is well", whereas a nonzero value could be used to indicate a specific error condition that had been encountered. The results of any computation carried out by the stored procedure should be returned through **out** parameters rather than using the **return** value.

Exercise 13.5

The following schema shows part of a database that holds information about repair technicians and the types of assemblies that they are competent to repair. Different technicians may be competent at repairing different assemblies.

1. Write a stored procedure that inserts a new row into the Technician table and returns a result of zero if it succeeds. If the new row would create a loop in the technician–supervisor hierarchy the insert should not be performed and the procedure should return the result code 99. Use recursion to check for loops in the hierarchy. You can assume that no hierarchy will be more than 10 levels deep.

2. Most commercial systems provide a function that returns the last identity value used in the update of a table. The empNr column in the Technician table would be a typical application for an identity column. Write a stored procedure that adds a new row to the Technician table and returns as an **out** parameter the identity value that was generated for the new row.

3. Assume that we have a table with the schema *Currency* (code, symbol, rate), where code identifies a particular currency, symbol is a single Unicode character, and rate is a conversion rate. Write a function that takes a currency code and a decimal number as parameters and returns a string consisting of the number provided multiplied by the rate for the currency code appended to the appropriate currency symbol. For example, if the Currency table contains a row ('USD', '$', 2.0), given the parameters 'USD' and 100.00 the function should return '$200.00'. The returned amount should have two decimal digits.

13.6 More Database Objects

Sequences

In SQL, a sequence is a database object that automatically generates a succession of values—note that this is a completely different meaning of the word "sequence" from its usage in XQuery, as we'll see later. Sequence objects were first introduced in SQL:2003 and are useful whenever it's necessary to produce a unique key value. Having the unique value generated inside the database, instead of in an external application, reduces the possibility of issues arising in areas such as performance and concurrency. Some SQL implementations may precompute the next values in a sequence and cache these ready for immediate use.

In some respects, a sequence is similar to an identity column in a table. However, unlike an identity column, a sequence is not tied to one specific table, and sequence val-

ues are unique within a database, rather than being unique within a table. Sequences can be created using the following syntax.

```
create sequence sequence-name
  [ as data-type | start with start-value |
  increment by increment-value |
  maxvalue value | no maxvalue
  minvalue value | no minvalue |
  cycle | no cycle  [ ,… ] ]
```

The data type must be a numeric type with a scale of zero (e.g., integer). The options are similar to those for identity columns, which were discussed earlier. One additional feature of sequences is that some implementations also allow a zero increment, so that the sequence effectively becomes a global constant value.

The standard syntax for obtaining the next value from a sequence is:

```
next value for sequence-name
```

but some implementations may have variations on this. They may also provide some means of accessing the last sequence number generated without generating a new one, such as:

```
previous value for sequence-name
```

SQL also defines **drop sequence** and **alter sequence** commands, but since these work in the obvious way, we won't discuss them here.

Indexes

An *index* is typically an ordered binary tree of key values, with pointers to the locations of the full records used to store the data. Because indexes are relevant only for physical modeling, not logical modeling, the SQL standard will never mention them. However, most commercial systems allow users to explicitly create and drop indexes, using syntax such as **create** [**unique**] **index** indexname **on** tablename (index-column [**asc** | **desc**] [,…]) and **drop index** indexname. Optimizers use indexes to improve performance and may create and drop indexes themselves. If supported, the unique index option provides one way of enforcing uniqueness constraints. However, most uniqueness constraints are better declared using **primary key** or **unique** declarations, although uniqueness constraints on optional columns usually require triggers. Indexes are often created automatically for primary and foreign keys.

Indexing can dramatically improve query performance, especially for large tables, since indexes often allow random access in main memory rather than sequential disk access. Some commercial systems may also allow a chosen index to be specified as *clustered*. This creates an alignment between the logical sequence of values in the indexed column(s) and the order in which rows are physically stored, potentially offering improved performance in retrieving row data. Obviously, if a table has multiple indexes, only one can be selected as the basis for clustering; typically this will be the index on the primary key column(s).

Although query efficiency (and hence perhaps constraint enforcement performance) can be improved by indexing, there is a price to be paid. Indexing may slow down the updating of a table, since any change to the data will involve not only a change to the base table, but also to any indexes that have been defined. However, this trade-off may still be acceptable, depending on the situation. Individual DBMSs may offer many different kinds of index, and strategies for choosing indexes may also vary from one DBMS to another. Since this book is primary concerned with conceptual and logical modeling, index strategies will be not be considered further.

Cursors

A cursor is essentially a mechanism to provide row-at-a-time access to the results of a prespecified **select** statement. The result is retained for as long as the cursor is being used (during which time the cursor is said to be *open*). Using a cursor is a four-stage process.

1. The cursor is *declared*, which defines the **select** statement and the mode of operation of the cursor.
2. The cursor is *opened*, at which time the associated **select** statement is executed and the temporary result table is formed.
3. The application navigates through the result table, typically row by row, to *fetch* the data needed for any required operations.
4. When the application has completed its work, the cursor is *closed*, which results in deletion of the results of the original **select** statement.

It's fairly common practice for these steps to be controlled directly from an application program. This is one reason why cursors have a reputation for sluggish operation, which is discussed in more detail later.

Most vendors support the basic SQL-92 syntax for cursor declaration, which is shown here. Some details were added in later versions of the standard, but these did not affect the basic SQL cursor concept.

```
declare cursor-name [ sensitive | insensitive ] [ scroll | no scroll ] cursor
  for select_statement
  [ for { read only | update [ of column_name [ ,... ] ] } ]
```

Optionally, the cursor can be declared as **for read only** (which improves performance if updating is not required) or **for update**, possibly restricted to specific columns. A cursor that is declared as **insensitive** makes a temporary copy of the data to be used, and all data requests are answered from this temporary table. Modifications therefore cannot be made to base tables. If **insensitive** is omitted, or the sensitive option is chosen, changes made to the underlying tables may be reflected in subsequent fetches. The **scroll** option specifies that a range of navigation operations should be enabled for the cursor. If omitted, or if the **no scroll** option is chosen, the cursor is limited to just fetching the next row of the result. Once a cursor is defined, it can be opened. The syntax could hardly be simpler:

```
open cursor-name
```

There's no option for the open clause to change the **select** statement; it simply executes the one already defined in the associated cursor declaration. It also positions the cursor to point just before the first row of the result set. Given an open cursor, data can be fetched from the results produced by the **select** statement.

```
fetch [ [ next | prior | first | last | { absolute | relative } ] from ] cursor-name
   into target-specification [ ,... ]
```

In the navigation options, **next** fetches the next row from the current cursor position, **prior** fetches the previous row, **first** and **last**, respectively, fetch the first and last rows of the result set, **absolute** retrieves a row by its row number in the result set, and **relative** moves the cursor the specified number of rows backward or forward in the result set. If none of the navigation options is specified in the **fetch** clause, **next** is assumed. If the cursor declaration did not include **scroll**, the only option available is **next**. Under these circumstances, if we want to move to a previous row we have to close the cursor, reopen it, and then **fetch** repeatedly until we reach the required row.

The target-specification varies somewhat depending on the environment that the data is being fetched into. The main restriction is that the number of variables waiting to receive values must be the same as the number of columns produced by the **select** statement associated with the cursor. Closing a cursor is as simple as opening it.

```
close cursor-name
```

Once the cursor is closed, its result set becomes undefined. Cursors are automatically closed at the end of the transaction they are contained within, even if a **close** command is not explicitly used. The following code fragment illustrates a typical series of cursor activities. The host language here is 'C'.

```
/*  variables dnum etc. will have been declared previously */

exec sql declare c1 cursor for
   select deptno, deptname, mgrno from tdept
   where admrdept = 'A00';

exec sql open c1;

while ( SQLCODE==0 ) {
   exec sql fetch c1 into :dnum, :dname, :mnum;
   /*     --
          -- do something interesting with the values
          --
   */
}

exec sql close c1;

/* carry on */
```

Cursors have a reputation for poor performance. One reason for this is that they are often used to retrieve data for use in an application program running on a different server to the database system. This requires some processing effort on the application server to

construct the cursor and submit it to the databases server. More importantly, the network overhead (serializing, transmission, and deserializing, both ways) introduces a significant lag in operations. This can be partly addressed by moving the cursor to a stored procedure in the database server. A better solution is to rethink the data requirements in terms of set operations rather than row-at-a-time operations, using the inherent relational power of the database server.

13.7 Transactions and Concurrency

Operations affecting data in a DBMS are contained in transactions. A transaction may be composed of many individual actions, which, in some sense, can be considered together as an atomic unit of activity. In order to preserve the integrity of the database, we can make positive decisions about whether to accept the results of a unit of activity or not. The corresponding database operations are **commit**, which makes the results permanent, and **rollback**, which aborts the activity and removes the results of the transaction, leaving the database in the state it was in before the transaction began. In some situations, the **commit** or **rollback** is carried out implicitly, depending on whether a program terminates normally or abnormally.

If database systems handled transactions one at a time this mechanism would be sufficient to maintain integrity. We could call this idealistic situation *serial* execution. However, a realistic database system could be expected to have many concurrent users, each carrying out their own individual transactions. It's therefore entirely likely that several users could be working in the same part of the database: for example, writing and reading data to and from the same table. Although we may expect integrity within a transaction, some potential problems arise when transactions overlap. The following situations are possible.

Dirty reads (also known as uncommitted reads). Transaction1 modifies a row in a table. Transaction2 reads that row before Transaction1 performs a commit. If Transaction1 subsequently performs a rollback, Transaction2 will have read a row that never really existed.

Nonrepeatable reads. Transaction1 reads a row from a table. Transaction2 modifies the row, or perhaps deletes it, and then performs a commit. If Transaction1 attempts to read the row again it will find that it has changed, or perhaps even disappeared completely!

Phantoms. Transaction1 reads some rows that satisfy a search condition. Transaction2 then creates some rows that would match the search condition used by Transaction1. If Transaction1 attempts to read the rows again, using the same search condition, it will obtain the original result plus some additional rows that have now mysteriously appeared.

Interactions between transactions can be controlled using *locks* on resources. A transaction that wants to use a resource, such as a row or a table, can acquire a lock that makes it the temporary owner of the resource. Other transactions are obliged to wait until the first transaction has relinquished the lock before they can compete for the same resource. A typical sequence is shown in Figure 13.3.

1. Applications *A* and *B* connect to the database

2. Application *A* starts to update Table 1
   update Table1 **set** num = 0
   where alpha = 'R'

3. The application acquires an *exclusive lock* (**X**) on the relevant row

4. Application *B* tries to retrieve all rows
   select * **from** Table1

5. Application *B* can retrieve the first two rows but is forced to wait for the row with alpha = R

6. Application *A* commits, and the lock is removed

7. Application *B* can now retrieve the row for alpha = R and beyond

Table1:

	alpha	num
	P	1
	Q	2
X	R	3
	S	4
	T	5

Table1:

alpha	num
P	1
Q	2
R	0
S	4
T	5

Figure 13.3 A typical locking sequence.

There are two main kinds of lock: *shared* and *exclusive*. A shared lock is typically acquired by a transaction when it wants to read some data but does not want another transaction to change it. The resource can be shared with other transactions that have the same intentions. An exclusive lock is typically acquired when a transaction wants to modify the resource, and wants to prevent other transactions from either writing to or reading from the resource until the operation is complete. Commercial systems may have many subtly different variations on these two main kinds of lock.

Obviously, locking is likely to have an impact on how quickly transactions can complete their business as they compete for resources. We have a potential trade-off here between better isolation between transactions but reduced performance, or reduced isolation but better performance, as shown in Figure 13.4.

Figure 13.4 Isolation versus performance.

Table 13.4 Standard isolation levels.

Isolation level	*Dirty reads*	*Nonrepeatable reads*	*Phantoms*
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not possible	Possible	Possible
REPEATABLE READ	Not possible	Not possible	Possible
SERIALIZABLE	Not possible	Not possible	Not possible

The choice will be dependent on the nature of the application, and so the SQL standard defines several *isolation levels* that represent different points along this spectrum. The standard isolation levels are summarized in Table 13.4. At the highest level, SERIALIZABLE, we can treat transactions as though they are running in a one-at-a-time serial mode, albeit with a certain cost in terms of performance.

Most database vendors support at least some of these isolation levels, but not necessarily under the same names. For example, DB2 refers to these same four levels as UNCOMMITTED READ, CURSOR STABILITY, READ STABILITY, and REPEATABLE READ (the last of these being particularly confusing when compared to the standard). Other vendors may offer additional isolation levels. For example, SQL Server 2005 supports the four standard levels plus an additional SNAPSHOT level.

An application can specify the isolation level required for a connection. If no level is specified, the DBMS will assume some default, such as READ COMMITTED, that represents a reasonable compromise between isolation and performance. It may also be appropriate to switch temporarily to a different isolation level to improve either isolation or performance. For example, an application reading data from a table that is read only could temporarily drop down to the READ UNCOMMITTED level to improve performance—in this case without compromising isolation, because dirty reads cannot occur on a table that is not being updated concurrently.

A practical DBMS also has to deal with a number of side effects of locking. One is the *deadlock* situation. Transaction1 requires resources *A* and *B*, and successfully acquires a lock on *A*. Meanwhile, Transaction2, which requires the same two resources, has acquired a lock on *B*. The result is that Transaction1 is unable to proceed because it is waiting for *B*, and Transaction2 is unable to proceed because it is waiting for *A*. Most DBMSs have a built-in background process to detect deadlock situations. Typically deadlock situations are resolved automatically by rolling back one of the transactions. Another potential problem is a poorly designed application that acquires a lock and retains it for an unduly long period. Again, an automatic process can be included to detect such conditions and roll back the offending transactions.

13.8 Security and Metadata

This section looks briefly at two topics that any database administrator needs to be familiar with: security and metadata in SQL. Let's begin with security. A database is *secure* if and only if operations on it can be performed only by users authorized to do so. Each

system user is assigned a user identifier (AuthID). The user who "owns" or creates a database has all access privileges on it and can grant and revoke access privileges on it to other users. SQL provides a *grant statement* for granting various kinds of privileges to users. The SQL-92 syntax is:

grant all privileges |
select | **insert** [(*col*)] | **update** [(*col*)] | **delete** | **usage** | **references** [(*col*)] [,...]
on *object*
  to *user-list* [**with grant option**]

where *object* is one of [**table**] tablename, **domain** domainname, etc. The user list may include **public**, meaning all users. SQL:1999 extended the statement to cater for "roles" (user groups). If **with grant option** is included, the user list has the power to grant the same privilege to others. For example, the following statement grants all users read access to the Stock table: **grant select on** Stock **to public**. The following grants the user with authorization id "Mgr" read access to the Stock table, as well as the right to update values in the price column: **grant select, update**(price) **on** Stock **to** Mgr. Privileges may be removed with the *revoke statement*:

revoke [**grant option for**] *privilege-list*
on *object*
from *user-list* [**restrict** | **cascade**]

For example, to revoke the update right granted earlier, use: **revoke update** (price) **on** Stock **from** Mgr. The table name in **grant** or **revoke** statements may be the name of a base table or view. Granting access to views but not the base tables is one convenient way of enforcing security. The **restrict** and **cascade** options are not yet widely supported.

In addition to database privileges, commercial systems also provide controls over operations related to database servers and instances of a DBMS. One server may contain multiple instances and one instance may support many individual databases. Typically, roles are arranged hierarchically, with lower levels of the hierarchy having fewer privileges than higher levels. For example, a lower level administrator may be able to carry out maintenance and utility operations but not access the data stored in the databases.

Now let's move onto *metadata* (data about data). SQL systems automatically maintain a set of *system tables* holding information about the database schema itself (e.g., base tables, views, domains, constraints, and privileges). In commercial DBMSs, such tables are collectively referred to under various names such as "system catalog". SQL-92 uses the term "catalog" in a different sense (for a collection of schemas within an environment). System tables of commercial systems also include physical information (e.g., about indexes) that is used by the optimizer. Users with access to the system tables may query them in SQL (just like the application tables).

From SQL-92 onward, the SQL standard defines metadata in terms of tables and assertions in a special schema named DEFINITION_SCHEMA. This provides a reference model for metadata, but there is no intention that this should be implemented in this way in any commercial realization—it's sufficient for an implementation to simulate its existence. The intended access to metadata is though *views* defined in another schema, named

INFORMATION_SCHEMA. Views in the INFORMATION_SCHEMA are defined in terms of the model in the DEFINITION_SCHEMA. Most of the views are defined with a **select** privilege granted to **public**, but no other privileges are granted, so the views cannot be updated. These schemas have been extended as new features of SQL, such as triggers, have been added in subsequent versions of the standard.

To conform to the standard, a DBMS should support these views, and many popular DBMSs are at least in the process of doing so. Commercial systems may still use their own specific names (e.g., SysTables, SysCatalog, etc.) and structures for the metadata. These generally provide equivalent features to the standard schemas, but with some variation in the detailed syntax.

As a simple example, the following query uses the standard Columns view to request the number of columns in the Customer table:

```
select count ( * )
from Columns
where table_name = 'CUSTOMER'
```

To give an idea of the variation to be found among commercial systems, the equivalent query in SQL Server 2005 might look like:

```
select count ( * )
from sys.Columns
where object_id = object_id ( 'CUSTOMER', 'U' )
```

whereas in DB2 we might see:

```
select count ( * )
from sysibm.Columns
where table_name = 'CUSTOMER'
```

13.9 Exploiting XML

SQL/XML

SQL:2003 introduced XML as a new built-in data type. XML data is strongly typed: XML values are assumed to be tree-like data structures that are distinct from their representation in the form of a text string. The XML type can be used wherever other SQL types are allowed: in table columns, as parameters of a routine, as variables, and so on. Although relational data and XML data are both structured, there are some significant differences between them. The Table 13.5 identifies a few distinctive points.

XML data can be stored in a database either in a text column (i.e., **char**, **varchar**, or **clob**) or in a column of data type **xml**. A textual representation might be preferred if it is important to retain the textual fidelity of a document (which may include text that has no relevance to the data structure) or if the XML is always entered and retrieved as a complete document. However, storing the data in an XML column allows efficient processing of the XML data in the database, rather than in a middle-tier application.

Table 13.5 Some differences between relational and XML structured data.

Feature	*Relational (SQL)*	*Basic XML*
Primary structure	"Flat" sets	Hierarchical
Data sequencing	Row order is not significant	Nodes are ordered
Primary access mechanism	Set operations	Navigation through hierarchy
Data typing	Always typed	Typed or untyped
Missing values	NULL	Absent or empty element

In addition to the XML data type, the SQL:2003 standard also introduced some related features, such as new XML operators, functions, and rules for mapping between SQL concepts and XML concepts. Most vendors broadly support these features, or their equivalents, but the need to integrate the additional XML features into different preexisting frameworks has inevitably resulted in some differences in syntactic detail. The fact that XML support is an area of keen competition for vendors is also likely to produce a variety of nonstandard extensions. The picture is further complicated by the need for interoperability with other standards, such as XQuery; discussed in more detail shortly. Fortunately, this competition is limited to the variety of operators, functions, and so on that are provided to integrate XML with SQL. The standards governing the structure of XML data are defined outside SQL, and so the XML values themselves are reasonably consistent from one database system to another.

In 2006, the part of the SQL standard dealing with XML (Part 14) was revised and extended to consolidate the features of SQL/XML, as it has come to be called. (The standard is sometimes referred to as SQL:2006, although this is a little misleading since only the XML part was revised in 2006.) Here we will use SQL/XML:2006 to specifically distinguish the more recent version where necessary. You also need to be aware that Microsoft's XML database technology, as used in SQL Server, is referred to as SQLXML, but is not the same as SQL/XML!

In SQL/XML:2006 the **xml** data type is based on the XQuery data model[1]. Legal values are either **null** (indicating the absence of any XML) or any legal value of the XQuery data model. An instance of the XQuery data model is a *sequence* of zero or more items, which are either atomic values of one of the defined data types, or a *node*. There are seven kinds of node: *document*, *element*, *attribute*, *text*, *namespace*, *processing instruction*, and *comment*. The **xml** data type can take optional modifiers that give a finer-grained definition of the XML structure. The general sequence structure is indicated by **xml** (**sequence**). A sequence containing exactly one document node is termed *XML content*, with the type modifier **xml** (**content**). A special type of content is where the document node contains exactly one element node (plus, possibly comment and processing instruction nodes), and this is termed an *XML document*, with the type modifier **xml** (**document**). A secondary modifier can be added for **xml** (**content**) and **xml** (**document**) to indicate whether the content or document can contain both typed and untyped elements (**any**), only untyped elements (**untyped**), or elements defined by a schema (**schema** name). If a schema is de-

1 To be more precise, SQL/XML is based on the XQuery 1.0 and XPath 2.0 Data Model.

fined, the element and attribute nodes are guaranteed to be valid according to the schema. These distinctions are summarized in Figure 13.5. If the type **xml** is specified without a modifier, any default assumption about these more specialized subtypes is implementation dependent.

The SQL:2003 standard introduced the predicate **is document** to test whether or not an XML expression is a document. The basic syntax is:

```
xmlexpression is [ not ] document
```

Although the syntax of the predicate remained unchanged from SQL:2003, the meaning shifted somewhat because of the realignment of SQL/XML:2006 with the XQuery data model. Three more predicates were added: **is content**, **xmlexists**, and **is valid**. The **is content** predicate tests whether or not an XML expression is legitimate XML content, using the syntax:

```
xmlexpression is [ not ] content
```

To test whether an expression contains valid XML, the **xmlexists** predicate can be used. The syntax, simplified a little, is as follows:

```
xmlexists ( XQuery-expression
   [ passing  value-expression
   [ as identifier ] [ { by ref | by value } ] ] [,...] )
```

Figure 13.5 XML Data Type Modifiers.

If the XQuery evaluation is of a value that turns out to be null (perhaps passed as an argument from SQL) then the predicate result is unknown. For nonnull values, if the XQuery evaluation returns an empty sequence, the predicate's value is false, otherwise the result is true. The following example shows the use of **xmlexists** in a select statement to retrieve "id" values for customers in a given city, with the customer details being stored in a column of the **xml** type.

```
select c.custId from customer as c
   where xmlexists ( ' $d/*:customerinfo/*:addr [ *:city = "Salt Lake City" ] '
   passing info as "d" )
```

The **is valid** predicate checks to see if an XML value is valid according to a given XML schema. The details of validity assessment are a little complex and are not covered here. The simplified syntax is:

```
xml-value-expression is [ not ] valid
   [ identity-constraint-option ]
   [ according to xmlschema { uri | registered-name } ]
   [ valid-element-clause ]
```

The xml-value-expression is checked according to the schema identified by the choice of namespace. If none is specified, each element is validated according to the schema associated with its namespace. If specified, the schema can be identified either by URI or by its registered name. The optional identity-constraint-option and valid-element-clause parameters provide for more detailed specification of exactly how schema checking should be done.

A number of built-in XML operators were introduced in the SQL:2003 standard. One of the operators, **xmlroot**, could be used to modify the version and standalone properties of an XML expression, but this was removed in SQL/XML:2006 so we don't cover it here.

The first two SQL/XML operations we'll look at, **xmlparse** and **xmlserialize**, convert back and forth between *structured* and *string* representations of XML. The **xmlparse** operation takes an SQL string expression and produces the equivalent XML structure. A parameter defines whether the string expression is expected to represent a document or content. The basic syntax has remained consistent since SQL:2003.

```
xmlparse ( { document | content } stringexpression
   [ { preserve | strip } whitespace ] )
```

The result of the operation could be assigned to a variable of type **xml** or stored in an **xml** column. The precise way in which whitespace is handled is by **xmlparse** is subject to detailed rules that are not considered here.

The **xmlserialize** operation performs the inverse of **xmlparse**, producing a string version of a given XML expression. The SQL standard does not mandate the precise format of the resulting string, but does require that a subsequent **xmlparse** of a string produced by **xmlserialize** should reproduce the original XML expression. In other words, given an XML document called myXmlData, the following construct:

```
xmlparse ( document ( xmlserialize ( document myXmlData as varchar ( max ) ) ) )
```

should produce a result equal to myXmlData. The original SQL:2003 syntax of **xmlserialize**

has been extended in SQL/XML:2006.

```
xmlserialize ( { document | content } xmlexpression as SQLdatatype
  [ version string ]
  [ including xmldeclaration | excluding xmldeclaration ] )
```

The SQL datatype for the result could be any SQL string type (**char**(n), **varchar**(n), …), but serialized XML tends to be lengthy, and so a character large object (**clob**) type is likely to be a common choice. Optionally, an XML declaration '< ?xml version="1.0" encoding="UTF-8"? >' can be prepended to the result: the default is **excluding xmldeclaration**. The XML version can also be separately specified: at present this will be limited to '1.0'.

The **xmlconcat** operation introduced in SQL:2003 has been retained in SQL:2006. It simply concatenates a list of XML expressions, producing a new XML expression.

```
xmlconcat ( xmlexpression [, …]  [ returning { content | sequence } ] )
```

The optional **returning** clause was added in SQL/XML:2006. If this is omitted, it is implementation defined whether returning content or sequence is implicit.

The SQL:2003 standard defined several "publishing" functions to generate XML expressions from SQL expressions. These have been extended in SQL/XML:2006, and new functions have been added. Here we cover only the SQL/XML:2006 definitions of the functions.

The **xmlelement** function is used to create an XML element from a given set of parameters. Recall that an XML element has the basic form:

```
<tagname optionalAttributes>  optionalElementContent </tagname>
```

So, for example, an element representing a customer might look something like:

```
<cust custid = "1234"> Widgets Inc </cust>
```

If required, the tagname can be further qualified by prefixing it with a namespace, in which case the namespace URI must also be specified. The **xmlelement** function constructs an individual XML element from the parameters provided and returns it as an **xml** value. The basic syntax for character strings is shown here. There is also a variant for binary strings that we won't consider.

```
xmlelement  ( name xmlelementname
  [ , xmlnamespaces (namespace-declaration-item [,…] ) ]
  [ , xmlattributes ( attributevalue [as attributename ] [,…] )
  [ , elementcontent [,…]
  [ option  { null on null | empty on null | absent on null | nil on null | nil on no content } ] ]
  [  returning { content | sequence } ] )
```

The embedded **xmlnamespaces** and **xmlattributes** functions are optionally used in the context of a given element to add any required namespace declarations or attributes to the element. The element content is defined by a list of expressions which is optional, so it's possible to create an empty element. Also optional is the **returning** clause which we've already seen.

Table 13.6 Options for handling null and nil.

Option	*Result when the function is applied to an SQL null*
null on null	A null SQL value results in a null XML value
empty on null	A null SQL value results in an empty XML element
absent on null	A null SQL value results in no element (i.e., an empty sequence)
nil on null	A null SQL value results in a XML element marked with xsi:nil = "true"
nil on no content	An element being created with no children is marked with xsi:nil = "true"

The content option requires a little more explanation. In SQL a missing optional value is explicitly marked as **null**. This is not a value recognized in XML, although **null** is legitimate for the SQL **xml** data type. Depending on the desired result, a missing value could be indicated in XML by simply omitting the relevant element or by including the element's begin/end tags with no contained value. If an explicit indicator equivalent to **null** is required, a further option is the "nil" mechanism specified in the XML Schema For Instances namespace. The namespace is typically introduced with a declaration such as:

```
xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
```

The xsi prefix is used by convention, but some other prefix could be substituted. Once this is defined, the **nil** attribute defined in XMLSchema-instance can be set to True to indicate that the element content is nil, as in:

```
<myElement xsi:nil = "true"></myElement>
```

An element attributed in this way may not have any element content, but may still carry other attributes. The SQL:2003 standard acknowledged the xsi:nil option, but did not prescribe how this should be specified in conjunction with operations such as **xmlelement**. SQL/XML:2006 now provides close control over these choices, as summarized in Table 13.6.

As a simple example of the use of **xmlelement**, consider a Customers table that is purely relational (it contains no XML). We could produce a result table containing a mixture of relational data and XML representing selected customer information in the following way:

```
select c.id,
   xmlelement( name "cust", xmlattributes (c.id as "custid"), custName) as Xcust
   from Customers as c where custid = 1234
```

which would produce a result such as:

id	*Xcust*
1234	<cust custid = "1234"> Widgets Inc </cust>

where the data contained in the *Xcust* column is of type **xml**. Calls to **xmlelement** can be nested, so that elements can be created within elements, producing the characteristic hierarchical structure of XML.

The **xmlforest** function is basically a convenient way of applying **xmlconcat** to a sequence of **xmlelement** results: it produces a sequence of elements from a list of expressions. The elements can be named explicitly or implicitly. As with **xmlelement**, calls to **xmlforest** can also be nested. The basic syntax is as follows:

```
xmlforest ( [ xmlnamespaces (namespace-declaration-item , ), ]
   elementvalue [ as elementname ] [,…]
[ option  { null on null | empty on null | absent on null | nil on null | nil on no content } ] ]
[ returning { content | sequence } ] )
```

The parameters are used in the same way as **xmlelement**. For example,

```
select o.orderId,
   xmlelement ( name "order",  xmlforest ( o.custName as "customer", o.date )  ) as Xorder
   from Orders as o
   where orderid > 9800
```

would produce a result such as:

orderId	*Xorder*
9876	<order> <customer> Widgets Inc </customer> <date> 2007-12-12 </date> </order>
9877	<order> <customer> Thingummy Co </customer> <date> 2007-08-15 </date> </order>

If preservation of the case in SQL names is case is important, they should be aliased as delimited identifiers. Regular identifiers are likely to be mapped to all uppercase in some implementations. For example, the *date* element in the example just given could appear as <DATE> … </DATE> instead of <date> … </date>.

To give an impression of the differences in syntax between different implementations, the result table just given could be produced in SQL Server by using a construct such as:

```
select o.orderId,
   (select o,custName 'customer', o.date 'date'
         from orders as o
         for xml path ('order'), type) as Xorder
from orders as o
where orderid > 9800
```

SQL Server uses various **for xml** … constructs to convert relational data to XML. The **for xml path** variant gives full control over how the XML structure is to be generated on a

column by column basis. The **type** parameter specifies that the result should be of type **xml**, as opposed to its string equivalent.

Returning to SQL/XML, the **xmlagg** function is an aggregate function, similar to SQL's **sum**, **avg**, etc. It returns a single value from a given bag of values, which must be the result of an **xml** type expression. The resulting **xml** values are concatenated to produce a single **xml** value. Optionally, an **order by** clause can be used to order the results before concatenating, using a similar syntax to SQL. Any **null** produced is dropped before concatenating. If all of the results of the expression are **null**, or the bag is empty, the result of the **xmlagg** function is the **null** value. The basic syntax is as follows:

```
xmlagg ( XML-value-expression [ order by sort-specification ]
[ returning { content | sequence } ] )
```

As an example of the **xmlagg** function, suppose we have an "Orders" table and want to group by customer, showing, for each customer, an element for each order placed by that customer. For simplicity, we'll just assume that both orders and customers are represented by a simple ID value. We could construct the result as follows:

```
select xmlelement
   ( name "Customer", xmlattributes ( o.custId as "customerId" ),
   xmlagg ( xmlelement ( name "order", o.orderId )
   order by o.orderId ) ) as XcustOrders
from Orders as o
group by o.custId
```

which would give a result resembling:

XcustOrders
<Customer customerId = 1234>   <order> AB123 </order>   <order> CD456 </order> </Customer>
<Customer customerId = 5678>   <order> BC345 </order> </Customer>
etc.

The **xmlpi** function generates a single XQuery processing instruction node, as follows:

```
xmlpi ( name target [ , string-expression ] [ returning { content | sequence } ] )
```

The target identifier (**name**) cannot be 'xml' (in any case combination) and the string expression cannot contain the substring '?>'. The **xmlcomment** and **xmltext** functions are also straightforward, each simply producing a node of the appropriate type. Here are some examples of the use of these functions

xmlpi (**name** "Warning", 'Turn off now')	produces	<?Warning Turn off now?>
xmlcomment ('This is my comment')	produces	<!-- This is my comment -->

xmltext ('Here is my text') produces Here is my text

The **xmldocument** function returns a document node (as defined by the XQuery data model) that has the given XML value as its content. This is useful for converting an arbitrary XQuery sequence into an XML document.

xmldocument (xmlvalue [**returning** { **content** | **sequence** }])

A separate casting function, **xmlcast**, is provided for casting to and from XML values. This works in the same way as the standard SQL cast function, except that at least one of the data types involved must be **xml**. It's also possible to cast between the subtypes of **xml**, subject to certain rules that we won't go into here. Note that casting backward and forward between XML values and strings is not the same as using the **xmlparse** and **xmlserialize** functions.

Like the **valid** predicate, the **xmlvalidate** function ensures that a given XML expression is valid according to the related schema definitions. As with the **valid** predicate, schemas can optionally be explicitly specified. However, unlike the **valid** predicate, the **xmlvalidate** function also persistently annotates the XML expression with type information. The syntax is:

xmlvalidate({ **document** | **content** | **sequence** } xmlexpression
[**according to xmlschema** { **uri** | registered-name }]
[valid-element-clause])

We saw earlier how the **xmlnamespaces** function could be used in the context of **xmlelement** and **xmlforest** functions. A more complete syntax for **xmlnamespaces** is:

xmlnamespaces ({ XML-namespace-URI **as** namespace-prefix |
default XML-namespace-URI | **no default** } [,…])

The **default** option applies the namespace definition to any nested elements, unless overridden by a lower level **no default**. Similarly, the **no default** option can be overridden buy a lower level **default**. The **default** and the **no default** options can be specified at most once in an **xmlnamespaces** argument list.

The SQL:2003 standard introduced an extensive set of mapping rules to define how SQL entities map from SQL to XML (and the reverse). The specification of the rules is quite extensive, and some aspects are implementation dependent, so we won't cover all of the details here. Broadly speaking, SQL data types are mapped to the closest equivalent type defined in XML Schema. XML Schema facets and annotations can be used to indicate distinctions in certain areas: for example, basic XML Schema does not distinguish between **character varying** and **character large object**, which are distinct types in SQL. Delimited identifiers in SQL are mapped as defined, preserving the upper/lower case of characters. Case is not guaranteed to be preserved for regular identifiers.

Some special treatment is given to characters that would result in illegal names in XML. The general technique is the replace the character by _xNNNN_ or _xNNNNNN_, where N is a hexadecimal digit and NNNN or NNNNNN is the Unicode representation of the character. There are two variants of the identifier mapping: *partially escaped* and *fully escaped*.

The key differences between these are in the treatment of noninitial colons and identifiers beginning with the letters "xml" (in any combination of uppercase or lowercase). The differences are summarized in the examples in Table 13.7.

Using XQuery

XQuery is a language for manipulating XML values. Roughly speaking, XQuery is partnered with XML data in the same way that SQL is partnered with relational data. Like SQL, XQuery is declarative: it expresses the data sources and the processing rules to be applied, but it does not specify the explicit procedure by which the results should be produced. One of the most important aspects introduced in SQL/XML:2006 was the ability to integrate XQuery into the established relational database framework—a key motivator in moving SQL/XML to the XQuery data model.

We'll return to SQL integration later, but first we'll look at some aspects of how XQuery works. It's important to understand some of the main features of XQuery in order to make sense of the SML/XML integration, but we won't attempt to cover every detail here. Some of the features we'll discuss are actually defined in the XPath standard, not the XQuery standard, but we'll gloss over that distinction and use "XQuery" as an umbrella term. The Chapter Notes give some pointers to the particulars if you need them. One point of caution: if you have a background in SQL, you will need to remember that, unlike SQL, XQuery is case sensitive, and so a little care is needed in specifying expressions. It's also important to keep in mind that we will be operating on XML values and not their serialized representations. So, for example, we're not trying to textually process "angle brackets", although these may be useful in denoting structures for human readability.

The crucial distinction between relational and XML data is that XML structure is inherently hierarchical. Interestingly, early hierarchical database systems were commercially successful—IBM's IMS being a prominent example—before they became overtaken by later developments. The rigidity of these earlier hierarchical systems proved to be uncompetitive against alternatives (especially the relational model) and so they became obsolete. XML does not set out to imitate these earlier systems, but there seems to be something compelling about the hierarchical notion that just won't go away.

Table 13.7 Mapping SQL identifiers to XML.

Mode	*SQL identifier*	*XML name*
Partially escaped	xmlthing XMLTHING :ab:cd	xmlthing XMLTHING _x003A_ab:cd
Fully escaped	xmlthing XMLTHING :ab:cd	_x0078_mlthing _x0058_mlthing _x003A_ab_x003A_cd

Figure 13.6 A node in a hierarchy.

For example, the directory structure of most operating systems is organized on the same basis. Even in a graphical interface we have become used to a system of "folders" that can contain a combination of other "folders" and leaf nodes called "files".

The file system analogy is particularly relevant here, because the XQuery expressions (actually, XPath, but we're not being picky) that are used to navigate the XML structures resemble the textual commands used to navigate directory structures in non-GUI environments. The basic concepts are shown in Figure 13.6.

We're going to be navigating from node Q, which provides the "context" for our path expression. Node Q might be the top of the hierarchy we're considering, in which case it can be taken to represent the whole of our XML document. Apart from that special case, we can always go up one level from Q to the node P, the *parent* of Q. If Q has *siblings* (nodes that also have the parent P) we can navigate sideways. If Q has *children*, then we can navigate down. Some combination of up/sideways/down movements can take us to a new node. Once we get there, the newly selected node can become the context for some other navigation.

Most of the time, we're going to be concerned with relationships between XML elements, so, for example, we might want to know about all elements that are "contained in" element Q (i.e., they are descendent nodes of the node Q). But we might also want to know about all elements that have some specific attribute, or all elements related to some particular namespace, and so on. All of these possible dimensions are known in XQuery as *axes*, and Xquery has a sophisticated notation for the precise definition of navigation along various combinations of axes. Unfortunately, the full notation is a little cumbersome. Fortunately, there are short-form notations that cover the main possibilities, and we'll concentrate on those.

To provide some concrete examples, we'll return to the *Movies* example from Chapter 1. There we saw how the content of a report could be captured as relational tables. Figure 13.7 shows similar information (with slightly different data from the example in Chapter 1) represented as an XML fragment. This is not an XML document because it lacks a root node. The XQuery `doc( )` function can be used to turn such fragments into documents.

```
<movie movieNr="1">
        <title>Cosmology</title>
        <released>2006</released>
        <director>Lee Lafferty</director>
</movie>
<movie movieNr="2">
        <title>Kung Fu Hustle</title>
        <released>2004</released>
        <director>Stephen Chow</director>
        <star gender="M">Stephen Chow</star>
</movie>
<movie movieNr="3">
        <title>The Secret Garden</title>
        <released>1987</released>
        <director>Alan Grint</director>
        <star gender="F">Gennie James</star>
        <star gender="M">Barret Oliver</star>
</movie>
<movie movieNr="4">
        <title>The Secret Garden</title>
        <released>1993</released>
        <director>Agnieszka Holland</director>
        <star gender="F">Kate Maberley</star>
        <star gender="M">Heydon Prowse</star>
</movie>
<movie movieNr="5">
        <title>The Da Vinci Code</title>
        <released>2006</released>
        <director>Ron Howard</director>
        <star gender="M">Tom Hanks</star>
        <star gender="M">Ian McKellen</star>
        <star gender="F">Audrey Tautou</star>
</movie>
<movie movieNr="6">
        <title>Cast Away</title>
        <released>2000</released>
        <director>Robert Zemeckis</director>
        <star gender="M">Tom Hanks</star>
</movie>
```

Figure 13.7 Movie information in XML.

Navigation is relative to a given node in an XQuery sequence. The basic navigational notation resembles that of a conventional file system. A *period* or *full stop* " . " means "the current node", " .. " means "the parent of the current node". A *slash* or *solidus* character " / " without further qualification means "go down one level"; " // " means go down any number of levels. Names are taken to refer to elements unless prefixed with "@", in which case they refer to attributes. At any step in the path a predicate can be introduced in square brackets. This will filter the resulting nodes: only those for which the predicate is "True" will be included. Table 13.8 shows some example expressions.

Table 13.8 Example path expressions.

Expression	*Result*
Find all directors movie/director	<director>Lee Lafferty</director> <director>Stephen Chow</director> <director>Alan Grint</director> <director>Agnieszka Holland</director> <director>Ron Howard</director> <director>Robert Zemeckis</director>
Find one star from each movie movie/star [1]	<star gender="M">Stephen Chow</star> <star gender="F">Gennie James</star> <star gender="F">Kate Maberley</star> <star gender="M">Tom Hanks</star> <star gender="M">Tom Hanks</star>
Find the titles of all movies with a female star movie/star [@gender = "F"] / .. / title	<title>The Secret Garden</title> <title>The Secret Garden</title> <title>The Da Vinci Code</title>
Find the release year of all movies starring Tom Hanks movie [star = "Tom Hanks"] / released	<released>2006</released> <released>2000</released>

Now that we have some idea of how to move around an XML hierarchy, we can consider how to query XML data. In simple cases the result of the path expression may be sufficient, but in general we will need more powerful expressions. XQuery is a *functional* language, so we must specify what we want to do in functional terms. Most software developers are familiar with the concept of a function. A function has a name, zero or more input parameters, and a body that computes some result to be returned. From a functional programming perspective, the arguments to a function can be constants, or the results produced by other functions (i.e., a parameter can be a call to another function). Now we have another hierarchy: functions that provide results to functions that provide … and so on. This *function* hierarchy may be established dynamically and may bear no direct relationship to the *data* hierarchy. The top level of the function hierarchy computes the result we are seeking: the lower level functions provide results to their parents to (ultimately) enable the top level computation. XQuery defines over a hundred built-in functions, but commercial systems may not yet implement all of these.

XQuery allows information to be combined from multiple sources and restructured to produce a new result. The most common expressions of this kind in XQuery are called FLWOR expressions, from the first letter of the clause types that may appear in the expression: **for**, **let**, **where**, **order by**, and **return**. FLWOR expressions are the XQuery equivalent of **select** statements in SQL.

The clauses in a FLWOR expression are defined in terms of *tuples*. The tuples are created by binding variables to values in **for** and/or **let** clauses, and so every FLWOR expression must have at least one **for** or **let** clause. The difference between the two is that **for** iterates over the items in an expression, producing a tuple for each, and **let** produces one tuple for the entire result of an expression. As a simple example, the following FLWOR

expression produces the same result as the last path example in Table 13.8. The dollar sign "$" indicates a variable in XQuery.

```
for $m in movie
   where $m/star = "Tom Hanks"
   return $m/released
```

As a slightly more complex example, let's assume that we want to invert the hierarchy of the source XML. For our Movies XML fragment, we could list the movies in which an actor starred under each actor, instead of listing the stars under each movie. An XQuery expression to do this might look like:

```
for $s in distinct-values ( movie/star )
order by $s descending
return
   <starring>
         <actor> { $s } </actor>
         <movies> { for $m in movie where $m/star = $s return $m/title } </movies>
   </starring>
```

Here we've used the **distinct-values**( ) function to make sure that each actor only appears once and sorted the result in descending order of actor name. The query contains *constructors*, indicated by braces around an inner expression, which are used to build element content. Evaluating the query produces the result:

```
<starring>
      <actor>Tom Hanks</actor>
      <movies>
            <title>The Da Vinci Code</title>
            <title>Cast Away</title>
      </movies>
</starring>
<starring>
      <actor>Stephen Chow</actor>
      <movies>
            <title>Kung Fu Hustle</title>
      </movies>
</starring>
…etc.
```

Of course, we're not limited to using a single XML document or fragment. Let's say we have some XML data containing reviews of the movies, which might follow a structure similar to the example shown in Figure 13.8.

The reviews are related to the movies by the unique movieNr reference. In standard SQL, if we wanted to show the movie titles along with the rating scores for each movie, we could use a join. We can do the same kind of thing in XQuery.

```
<review reviewNr="1">
        <movieNr>4</movieNr>
        <rating>4</rating>
        <comments>Classic story</comments>
</review>
<review reviewNr="2">
        <movieNr>5</movieNr>
        <rating>4</rating>
        <comments>Great movie!</comments>
</review>
<review reviewNr="3">
        <movieNr>5</movieNr>
        <rating>5</rating>
        <comments>Superb</comments>
</review>
<review reviewNr="4">
        <movieNr>5</movieNr>
        <rating>1</rating>
        <comments>Not my kind of movie</comments>
</review>
```

Figure 13.8 Movie reviews in XML.

Here's one way of doing the equivalent of an inner join between the movie data and the review data.

```
for $m in movie
where some $r in review satisfies( $m/@movieNr = $r/movieNr )
return
<reviewscore>
   { $m/title, $m/released }
   { for $r in review where $m/@movieNr = $r/movieNr return $r/rating }
</reviewscore>
```

The query includes the release year as well as the title to avoid ambiguities between movies with the same name. The "**where some** … **satisfies** (…) …" construct is an existential quantifier test, so that we only get results for movies that have had a review. XQuery also has a "**where every**… **satisfies** (…)" universal quantifier construct, which we don't need here. The query produces the following result:

```
<reviewscore>
        <title>The Secret Garden</title>
        <released>1993</released>
        <rating>4</rating>
</reviewscore>
<reviewscore>
        <title>The Da Vinci Code</title>
        <released>2006</released>
        <rating>4</rating>
        <rating>5</rating>
        <rating>1</rating>
</reviewscore>
```

To get the equivalent of an outer join, which would also include the movies with no reviews, we would simply need to omit the existential quantifier test in the second line of the query.

As a final XQuery example, we'll show the equivalents of the *aggregate* or *bag* functions in SQL, which include **sum** ( ), **count** ( ), **avg** ( ), **max** ( ), and **min** ( ). Let's suppose we want to produce a summary element for each movie, showing the total number of reviews and the average score as attributes. The following query would be one way of producing the required result.

```
for $m in movie
return
<reviewsummary total = "{ count ( review [ movieNr = $m/@movieNr ] ) }"
                    avg = "{ avg ( review [ movieNr = $m/@movieNr]/rating ) }" >
  { $m/title, $m/released }
</reviewsummary>
```

Here, we've used the XQuery functions **count** ( ) and **avg** ( ), which operate in a similar way to their SQL counterparts. For an empty list of elements (in this case, movies that have no reviews) the **count** ( ) function returns zero and the **avg** ( ) function returns an empty result. Running the query produces the following XML fragment.

```
<reviewsummary total="0" avg="">
    <title>Cosmology</title>
    <released>2006</released>
</reviewsummary>
<reviewsummary total="0" avg="">
    <title>Kung Fu Hustle</title>
    <released>2004</released>
</reviewsummary>
<reviewsummary total="0" avg="">
    <title>The Secret Garden</title>
    <released>1987</released>
</reviewsummary>
<reviewsummary total="1" avg="4">
    <title>The Secret Garden</title>
    <released>1993</released>
</reviewsummary>
<reviewsummary total="3" avg="3.33333333333333">
    <title>The Da Vinci Code</title>
    <released>2006</released>
</reviewsummary>
<reviewsummary total="0" avg="">
    <title>Cast Away</title>
    <released>2000</released>
</reviewsummary>
```

XQuery has many other features, and a more complete description would require a book in its own right. Some references to further information sources are included in the chapter notes.

Finally, we'll look at some mechanisms provided in SQL/XML that help integrate XQuery into the relational framework. The most important of these is the **xmlquery** function, which allows an XQuery expression to be evaluated from within an SQL context. SQL alone cannot operate on parts of an XML document: the whole document is treated as a single value. With the **xmlquery** function it's possible to query inside an XML document so that XML data can participate in SQL queries and SQL processing can be applied to returned XML values (for instance, using **order by** to order the results). The basic syntax of **xmlquery** is as follows. The arguments are used in a similar way to the equivalents we have already seen in other functions.

```
xmlquery ( XQuery-expression
[ passing [ { by ref | by value } ]  XML-query-argument  [,…] ]
[ returning { content | sequence } ]
[ { by ref | by value } ]
[ {  null on empty | empty on empty } ] )
```

Another function, **xmltable**, provides a useful way to integrate XQuery results back into SQL. Normally, an XQuery expression would return a sequence, but using **xmltable** we can execute an XQuery expression and return values as a table instead. The returned table can contain columns of any SQL data type, including XML, although values inserted into **xml** columns in **xmltable** must be well formed with a single root node. The basic syntax is:.

```
xmltable ( [ xmlnamespaces ( namespace-declaration-item [,…] ), ]
row-pattern [ passing [ { by ref | by value } ]  XMLquery-argument [,…] ]
columns { { colname for ordinality |
colname datatype [ by ref | by value ] path column-pattern} [,…] } )
```

The row-pattern defines an expression that produces a sequence of elements. Each element in this sequence will become a row in the resulting table. The **columns** clause lists the column definitions and the path expression used to find the column value within each element of the row-pattern. The **for ordinality** option creates a column with an automatic ordinal reference to the row, so the first row is 1, the second row 2, and so on. An example is shown here. This assumes that we have an **xml** value called movieXML; the contents of which are similar to the examples we used when discussing XQuery. If this **xmltable** function call is contained in a **select** statement, the value being passed could be selected from a suitable xml column in a table.

```
xmltable ('//movies'
passing movieXML
columns
   "Seqno" for ordinality,
   "MovieNr" int path '@movieNr',
   "Movie Name" varchar (50) path 'title',
   "Release Year" char(4) path 'released'
   "Director" varchar(50) path 'director' )
```

The resulting table would have the following form, and could be aliased and so on, in the same way as a table produced by other means. In this case, values in the automatically generated SeqNo column and the retrieved MovieNr column are coincidentally the same. Had the XML data been in a different order, the SeqNo column would remain unchanged but the MovieNr column (and the other retrieved columns) would reflect the XML element ordering.

SeqNo	*MovieNr*	*Movie Name*	*Release year*	*Director*
1	1	Cosmology	2006	Lee Lafferty
2	2	Kung Fu Hustle	2004	Stephen Chow
3	3	The Secret Garden	1987	Alan Grint
4	4	The Secret Garden	1993	Agnieszka Holland
5	5	The Da Vinci Code	2006	Ron Howard

Exercise 13.9

The following schema shows part of a database that holds information about repair technicians and the types of assemblies that they are competent to repair. Different technicians may be competent at repairing different assemblies.

1. Using the facilities provided in a modern database system, turn the contents of the repair database into three XML documents, i.e., one XML document per table.

2. Use XPath and XQuery expressions to produce the following results, using one or more of the Technician, Competence, and Assembly XML documents. Produce the results either as an XML document or as an XML fragment.

   (a) List all supervisors and under each supervisor all technicians that report to them. The Supervisor and Technician elements should have empNr as an attribute and the concatenation of 'firstName' || ' ' || 'lastName' as the element content.
   (b) List the assemblies in a hierarchic form (i.e., the top level assemblies will have subassemblies, which will have their own subassemblies, etc.). Show the assemblyCode as the content of each element.
   (c) List all the assemblies with a complexity code of 5 or higher that have a report. Order the list by complexityNr (descending) and assemblyCode (ascending).

(d) For each technician, list the types of assembly that they are competent to repair. Include each assembly's complexityNr as an attribute.

(e) List all technicians with a certificationRefs element under each technician. Use xsi:nil = "true" in cases where the technician has no certificationRef. For each certificationRef that does exist, show the related assemblyCode.

(f) List all assemblies that no technician is competent to repair.

3. Using the facilities provided in a modern database system, turn each XML result from the queries in Question 2 into a relational table.

13.10 Summary

Base table schemes may be defined using a *create-table statement*, whose main syntax is:

```
create table tablename (
  colname        data-type        [ not null ]
                                  [ default literal | null | user ]
                                  [ primary key | unique ]
                                  [ references tablename ]
                                  [ check (col-condition-on-same-row) ]
  [,...]
  [, primary key ( col-list ) ]
  [, unique ( col-list ) ]
  [, foreign key ( col-list ) references tablename [ ( unique-col-list ) ] [,...] ]
  [, check (table-condition-on-same-row) [,...] ] )
```

In addition to creating tables by definition, as just shown, tables can also be created **like** another table or **as** the result of a select query. Other SQL statements exist for tasks such as altering and dropping tables, and equivalent alter or drop statements are available for other types of database objects.

Views may be created using a *create-view statement*, with the syntax:

```
create view viewname [ ( col-list ) ] as
  select-query
            [ with check option ]
```

View definitions are used to translate queries over views into equivalent queries on base tables. Some views are nonupdatable (e.g., views involving joins).

Triggers are used to respond to specific events in the context of a given table or view. When the specified event is detected, the body of code in the trigger is executed. The data leading to the event can be examined by the trigger to determine the appropriate course of action. The basic syntax for creating a trigger is:

```
create trigger triggername
  { before | after } { insert | delete | update [ of col-list ] } on tablename
  [ referencing     { old | new } [row] [ as ] correlation-name |
                    { old | new } table [ as ] table-alias        [, ...] ]
  [ for each { row | statement } ]
```

```
[ when ( condition ) ]
triggered-SQL-statement
```

Routines are bodies of code that are explicitly *called*, in contrast to triggers, which are inherently event driven. User-defined functions return either a scalar or a table result. The basic syntax for creating a function is:

```
create function functionname ( [ parameter-list ] )
  returns { data-type | table col-name data-type [,...] }
  function-body
```

Stored procedures are able to carry out a greater range of activities than functions, such as updating tables, and are the standard way of encapsulating SQL code. Stored procedures are often used to "insulate" the database from user actions that could prove damaging. The basic syntax for creating a stored procedure is:

```
create procedure procedurename
  ( [ { in | out | inout } parmname parmtype [,...] ] )
      procedure-body
```

A database schema may contain other types of objects. *Sequences* provide for the automatic generation of unique identifiers. *Indexes* can be used to improve query performance. *Cursors* can be used to provide row-at-a-time processing.

Practical database systems include measures to deal with concurrency, typically based on the selective locking of resources to avoid interference between transactions. The preferred position between more isolation and more throughput can be specified at the statement or the connection level.

Database security revolves around two main concepts. Users are authenticated against specified credentials to confirm their identity. Privileges—the rights to carry out certain types of action—can be given to, or withheld from, specific individuals or groups.

Metadata—information about the database system itself—is stored in various tables for the system's internal use. This information can also be queried, although standard practice is to discourage direct access to these tables and instead use system views.

XML is becoming increasingly important, and most database vendors have added features to store XML data, to operate on XML data, and convert backward and forward between XML and relational table structures. XPath and Xquery allow XML data to be queried in a way that corresponds roughly to the way that the **select** statement is used in querying SQL tables. XQuery is a rich language with many features that would require a complete book to cover, and we have only been able to give an outline of its capabilities in the space available here.

This chapter completes our basic coverage of the major feature of modern relational database systems. Some other aspects of SQL are considered elsewhere in the book in the context of the topic under consideration. More advanced SQL aspects are included in the online Appendix C, and the chapter notes provide further detail. While knowledge of the relevant SQL standards provides an excellent basis, the easiest way to learn the language is by using it, which entails discovering how your chosen SQL dialect differs from the standard. Fortunately, most commercial DBMSs provide extensive online resources to assist you in this regard.

Chapter Notes

The SQL standard has been adopted by the International Standards organization (ISO) as ISO/IEC 9075. Copies can be ordered from ISO or via national standards organizations such as ANSI. Because of its size (several thousand pages in total), the standard is split into a number of parts. Part 2 (Foundations) and Part 14 (SQL/XML) are the most relevant to this chapter. Late drafts of SQL:2003 and SQL/XML:2006 are freely available at *www.wiscorp.com*. The standard continues to evolve, and a new version is expected to be approved in 2008.

Practical books related to SQL tend to be associated with a particular implementation; a notable exception is the series of books by Celko: see, for example Celko (2000). Examples of product-specific references include Chong et al (2008) – DB2; DuBois (2006) – MySQL; Greenwald et al (2007) – Oracle; and Vieira (2007) – SQL Server. Various specialist magazines, such as *SQL Server Magazine* (*www.sqlmag.com*) also provide coverage of detailed issues.

XML integration is relatively new in most relational databases. Because of this, and the variation between different implementations, the vendor's documentation is often the most useful resource. Most commercial systems provide on-line help that can be searched in various ways. The relevant standards are being progressed through the World Wide Web consortium (W3C). Full details, including the current versions of the standards themselves, can be found at *www.w3c.org*. The standards documents are not particularly easy to read, but a large amount of more readable tutorial information is available. XQuery in particular is still relatively unfamiliar to many database practitioners, but it seems likely to become progressively more important in utilizing heterogeneous data. Useful introductions can be found in Walmsley (2007) and Melton and Buxton (2006).

14 Schema Transformations

14.1 Introduction
14.2 Predicate Specialization and Generalization
14.3 Nesting, Coreferencing, and Flattening
14.4 Other Transformations
14.5 Conceptual Schema Optimization
14.6 Normalization
14.7 Denormalization and Low Level Optimization
14.8 Reengineering
14.9 Data Migration and Query Transformation
14.10 Summary

14.1 Schema Equivalence and Optimization

Previous chapters discussed how to model the structure of an application in terms of a conceptual schema, and then map it to a logical schema for implementation in a relational database system. Although much of the design and mapping can be automated, humans are required to perform CSDP step 1, since verbalizing the relevant facts about the real world involves human understanding. Given the informal nature of this initial step in modeling the UoD, it is not surprising that people often come up with different ways of describing the same reality.

Hence the same business domain may be modeled by more than one conceptual schema. Moreover, the same conceptual schema may be mapped in different ways to a logical schema, either by choosing a different type of logical model (e.g., object relational instead of pure relational) or by overriding the default mapping choices (e.g., choosing separate subtype tables or denormalizing). Finally the same logical schema may be implemented in more than one way in the physical schema (e.g., different index choices).

To help you choose from such a variety of different designs, this chapter examines various ways in which schemas may be transformed to provide alternatives and then presents guidelines for optimizing your choice. On the way, we'll cover some theory on schema transformations and normalization to ensure a proper understanding of the underlying concepts. We'll start at the conceptual level and then move down to the logical and physical levels. Examples will then illustrate how the ideas can also be used to reengineer existing applications that have proved unsatisfactory.

The remainder of this section provides a simple introduction to schema transformation and optimization, using an example. Earlier chapters already introduced some modeling choices. For example, we might model a ternary fact type in either flattened or nested form. Basically, two conceptual schemas are **equivalent** if and only if whatever UoD state or transition can be modeled in one can also be modeled in the other.

As a simple example, consider the medical report shown in Table 14.1. Here a check mark in the appropriate column indicates that a patient smokes or drinks. Since both these "vices" can impair health, doctors are often interested in this information. Try to schematize Table 14.1 for yourself before looking at the solutions provided.

Figure 14.1(a) shows one conceptual schema for this UoD, together with the sample population. Here two optional unaries are used for the smoker–drinker facts. In Table 14.1 the absence of a check mark might mean that the patient doesn't smoke/drink (closed-world approach) or simply that we don't know (open-world approach). You would need to check with the domain expert as to what is intended here. For our discussion, we'll assume that closed world semantics are intended.

Table 14.1 A report on vices of medical patients.

PatientNr	*Patient name*	*Smokes?*	*Drinks?*
1001	Adams, A		✓
1002	Bloggs, F	✓	✓
1003	Collins, T		

Figure 14.1 Two ways of modeling Table 14.1.

Another way to model this domain is to generalize the smoking and drinking predicates into a single binary, introducing the object type Vice {S = Smoking, D = Drinking} to maintain the distinction, as in Figure 14.1(b).

Intuitively, most people would consider the two schemas in Figure 14.1 to be equivalent. Formally, this intuition can be backed up by introducing Vice as a "virtual object type" to the schema of Figure 14.1(a), and by specifying how the predicates of each can be translated into the predicates of the other. For example, facts expressed in the first model may be expressed in terms of the second model using the translations

Patient smokes **iff** Patient indulges in Vice 'S'
Patient drinks **iff** Patient indulges in Vice 'D'

and facts in the second model may be expressed in terms of the first model using the following translation. Like the schemas themselves, these translations can be mapped into formulae of predicate logic. Formal logic may then be used to prove schema equivalence.

Patient indulges in Vice **iff** Patient smokes **and** Vice has ViceCode 'S'
or
Patient drinks **and** Vice has ViceCode 'D'

The equivalence just considered is an example of *contextual equivalence* since it adds definitional context to the original schemas. It also requires formal recognition of "virtual object types" (e.g., Vice exists implicitly in the first schema). This notion is called "*object relativity*", since it effectively allows that "objects lie in the eye of the beholder". This permits classical logic proof techniques to be applied to such cases.

The next few sections discuss some theorems that can help us decide whether or not two schemas are equivalent. These theorems can also help us to transform one schema into an alternative schema that is either equivalent, or at least acceptably close to being equivalent (sometimes we may wish to strengthen or weaken our schema a little by adding or deleting information). The act of reshaping a schema like this is said to be a **conceptual schema transformation**.

Figure 14.2 The schema equivalence example in UML.

Although the conceptual transformations are specified in ORM, they can be translated into other modeling notations. For example, the previous equivalence (minus definitional context) may be set out in UML, as shown in Figure 14.2, using "⇔" for "is equivalent to". Using ORM enables many more complex transformations to be captured graphically, without needing to supplement the diagram with textual constraints. But if you want to work in a non-ORM notation, you should be able to translate the transformation theorems into your preferred notation without much trouble.

Knowledge of schema transformations helps us to see what different design choices are possible. Moreover, if two independently developed schemas are to be either fully or partly integrated, we often need to resolve the differences in the ways that each schema models common UoD features. To do this, we need to know whether one representation can be transformed into the other, and if so how.

Another use of conceptual schema transformations is to reshape the original conceptual schema into one that maps directly to a more efficient implementation. This process is known as **conceptual schema optimization**. For example, the conceptual schema in Figure 14.1(a) maps to a single table. Taking a closed-world interpretation, we may use the table scheme shown in Figure 14.3(a). For discussion purposes, the population for the sample data is included. However, because the indulges fact type is *m:n*, the schema in Figure 14.1(b) maps to two tables, as shown in Figure 14.3(b). Again, the sample population is included.

Figure 14.3 Relational models based on the conceptual models in Figure 14.1

For many applications, the single-table solution is more efficient. It enables us to list all the information about the patients without requiring a table join. It also avoids any intertable accesses required to enforce the subset constraint (e.g., when adding a vice fact or deleting a patient). So the schema of Figure 14.1(b) might be optimized by transforming it to that of Figure 14.1(a) before passing it to the normal Rmap procedure.

However, if the set of vices we wish to record often changes, the two-table solution may be preferable since it caters for such changes without requiring the schema itself to change. Guidelines for performing such optimizations are given later in the chapter. If the optimized conceptual schema still fails to give an efficient map, some denormalization may then be required, as discussed in Section 14.7.

In reengineering, optimization is applied to an existing database. So after the schema itself has been transformed, we have the additional task of *data migration*, to populate the new schema with the information stored in the original database. This task can be lengthy and tedious without automated assistance.

Here is one basic way of transforming the populated two-table model in Figure 14.3(b) into the single-table model in Figure 14.3(a), assuming that an SQL database is used. First add an optional isSmoker column to the Patient table, using the following statement:

```
alter table Patient
 add column isSmoker char check (isSmoker in ('Y','N'))
```

At this stage, the isSmoker column is populated with nulls. The check constraint is not violated because it evaluates to unknown (instead of false). Now add the appropriate 'Y' or 'N' values to indicate who smokes:

```
update Patient
 set isSmoker = case
                  when patientNr in
                        (select patientNr from Indulges where vice = 'S')
                  then 'Y'
                  else 'N'
                end
```

Now make the isSmoker column mandatory:

```
alter table Patient
 add constraint isSmokerNonNull check (isSmoker is not null)
```

Similarly, add the isDrinker column to the Patient table, update its population, and finally drop the Indulges table. In practice, the most efficient way to migrate data may depend on the DBMS(s) being used. In contrast to the simple example just discussed, it is more typical to leave the original database unaltered, create a new schema and migrate data to it, and then drop the original database (see Section 14.9).

That completes our high level introduction to the basic concepts of schema equivalence and optimization. The following sections develop these concepts further, as well as providing a concise account of normalization theory.

14.2 Predicate Specialization and Generalization

The previous section illustrated how the same UoD structure may be described by different, but equivalent, schemas and discussed the use of transformations to reshape one schema into another. This section considers a class of schema transformations known as **predicate specialization**, as well as its inverse, **predicate generalization**.

If two or more predicates may be thought of as special cases of a more general predicate, then we may replace them by the more general predicate, as long as the original distinction can be preserved in some way. For example, if we transform the schema of Figure 14.1(a) into that of Figure 14.1(b), we *generalize* smoking and drinking into indulging in a vice, where vice has two specific cases. If we transform in the opposite direction, we *specialize* indulging in a vice into two predicates, one for each case.

Predicate specialization and generalization are similar notions to object type specialization and generalization, except that it is rare to specify any subtype connections between predicates (the predicates must be objectified for this to be legal).

A predicate may be specialized if a *value constraint or a frequency constraint* indicates that it has a finite number of cases. Examples with value constraints are more common, so we examine these first. The drinker–smoker example provides one illustration, where Vice has the value constraint {'S', 'D'}. As another example, recall the Olympic Games schema reproduced in Figure 14.4(a).

Because there are exactly three kinds of medal, the ternary may be specialized into three binaries, one for each medal kind, as in Figure 14.4(b). You may visualize the transformation from schema (a) into schema (b), thus when the object type MedalKind is *absorbed* into the ternary predicate, it divides it (or specializes it) into the three binaries. Hence this transformation is also known as *object type absorption*. The reverse transformation from (b) to (a) generalizes the three binaries into the ternary by extracting the object type MedalKind—this may be called *object type extraction*.

Notice that in the vices example, a binary is specialized into unaries. With the games example, a ternary is specialized into binaries. In general, when an *n*-valued object type is absorbed into a predicate, the *n* specialized predicates that result each have one less role than the original (since the object type has been absorbed). This general result is set out in Figure 14.5. Although the equivalence is set out diagrammatically to aid understanding, there is a formal mapping of the diagrams to predicate logic where the actual equivalence proofs were made (Halpin 1989b). In this sense, the results may be called theorems.

Figure 14.4 The ternary is specialized into three binaries by absorbing MedalKind.

where $m \geq 1$, and each S_i corresponds to R where $B = b_i$

Figure 14.5 PSG1: R may be specialized into S_1 .. S_n by absorbing B.

The schema equivalence in Figure 14.5 is called *Predicate Specialization/Generalization theorem 1* (PSG1). The predicates R and S, respectively, have $m + 1$ roles and m roles, where $m \geq 1$. The object types A_1 .. A_m and B are not necessarily distinct. The value type for B has n values b_1,.., b_n. If $m = 1$ we have conversion between a binary and n unaries; if $m = 2$, the conversion is between a ternary and n binaries; and so on. Transforming from left to right specializes the predicate R into n predicates S_1 .. S_n by absorbing the object type B. The reverse transformation from right to left generalizes the shorter predicates into the longer one by extracting B. As an exercise, draw the diagrams for the cases $m = 1$, 2, and 3.

The theorem PSG1 holds regardless of whatever additional constraints are added. However, *any constraint added to one of the schemas must be translated into an equivalent, additional constraint on the other schema*. For example, the uniqueness constraint in Figure 14.4(a) translates into the three shorter uniqueness constraints in Figure 14.4(b). This is an instance of the following corollary to PSG1 (using "UC" for "uniqueness constraint" and reading "spans" as "exactly spans").

If a UC in R spans a combination of B's role and other roles, a UC spans the specialization of these other roles in S_1 .. S_n; and conversely.

Figure 14.6 illustrates the most common case, where R is a ternary with a UC spanning B's role and one other. However the result applies for longer predicates too. The games example of Figure 14.4 is an instance of the equivalence in Figure 14.6, where B = MedalKind, $n = 3$, and B's role is included in a compound uniqueness constraint. What happens, however, if B's role in the general predicate R is not included in a uniqueness constraint? Since R is elementary, its other roles must be spanned by a uniqueness constraint; this constraint is transformed into a mutual *exclusion* constraint over the specialized predicates.

Each S_i corresponds to R where $B = b_i$

Figure 14.6 The UC on the left is equivalent to the UCs on the right.

Figure 14.7 The UC on the left is equivalent to the exclusion constraint on the right.

Exclusive unaries provide the simplest case of this. For example, suppose that workers in a company may hold at most one position (manager, clerk, or secretary). This may be rephrased in terms of exclusive unaries as shown in Figure 14.7, assuming appropriate translations between the predicates. For implementation purposes, the binary version is usually preferred (e.g., its relational schema simplifies both updates to a worker's position, and schema updates to the list of allowable positions). The larger the number of unaries, the worse the unary solution becomes (why?).

Another example, this time with exclusive binaries, is shown in Figure 14.8. For simplicity, reference schemes are omitted. Here the task codes 'A' and 'R' denote authoring and reviewing. Note the two alternative ways of saying that a person cannot act both as an author and as a reviewer of the *same* book.

Figure 14.8 The UC on the left is equivalent to the exclusion constraint on the right.

These two examples illustrate unary ($m = 1$) and binary ($m = 2$) cases of the following second corollary to theorem PSG1. This result applies to longer predicates as well (see Figure 14.9). Here the exclusion constraint means that no row of values may appear in more than one of the S_i fact tables.

If a UC spans all roles of R except for B's role, then S_1 .. S_n are mutually exclusive; and conversely.

where $m \geq 1$, and each S_i corresponds to R where $B = b_i$

Figure 14.9 The UC on the left is equivalent to the exclusion constraint on the right.

Figure 14.10 A textual constraint in schema (a) appears as a graphical constraint in (b).

In rare cases, when a schema transformation is performed, a graphical constraint in one schema may have no corresponding graphical constraint in the other. In this case, the constraint should be expressed as a *textual constraint* in the other version. For example, consider the two schemas in Figure 14.10. The codes 'A' and 'S' indicate "assistant" and "supervisor", respectively. Reference schemes are omitted for simplicity. A person might supervise one project and assist on another.

Assuming appropriate translations between the predicates, the uniqueness constraint in schema (a) translates to the exclusion constraint in schema (b), and the top uniqueness constraint in (b) is implied. However, the lower uniqueness constraint in (b) declares that each project has at most one supervisor. This cannot be expressed as a graphical constraint on the ternary since it is a *restricted uniqueness constraint* (as discussed in Section 7.4). To ensure equivalence, this is added to the left-hand schema as the textual constraint **Each** Project uses **at most one** Employee in Position 'S'.

If Project plays other functional roles, the binary approach of the right-hand schema would normally be preferred since its relational version simplifies the enforcement of this supervision constraint. If Project has no other functional roles, the ternary approach might be preferred since it maps to a single table.

Since schema transformation theorems are typically applied to subschemas within some global schema, care must be taken to preserve any *mandatory role constraints* (implicit or explicit). For example, in Figure 14.4, if Country plays no other role, then its role in schema (a) and the disjunction of its roles in (b) are implicitly mandatory. If Country does play another role in the global schema, and at least one medal result must be recorded for it, then these simple and disjunctive mandatory role (inclusive-or) constraints must be explicitly shown (see Figure 14.11).

Figure 14.11 A mandatory constraint in (a) maps to an inclusive-or constraint in (b).

Figure 14.12 A frequency constraint in (a) maps to an equality constraint in (b).

This illustrates our third corollary to theorem PSG1:

If *Ai*'s role (or role disjunction) in *R* is mandatory, then the disjunction of its specialized roles is mandatory, and conversely ($1 \leq i \leq m$).

The inclusion of "(or role disjunction)" covers the rare case when A_i plays more than one role in *R* (recall that $A_1 .. A_m$ are not necessarily distinct). Note also how our convention for implicit mandatory role constraints enables theorems such as PSG1 to be specified without any assumption about other roles played in the global schema.

Now consider Figure 14.12(a). Here Country's medal winning role is optional (this implies Country plays another role, depicted here as an unnamed role) but it has a *frequency constraint* of 3. So if any medal results are recorded for a country, all three medal results (gold, silver, and bronze) are required. To express this in Figure 14.12(b) we add an *equality constraint* between the medal winning roles played by Country. This is an instance of the following fourth corollary to theorem PSG:

If *R* is a ternary with a UC spanning just *B*'s role and one other role, then adding a frequency constraint of *n* to this other role is equivalent to adding an equality constraint over the specialized versions of that role.

Now consider Figure 14.13(a), where again Country plays another role in the global schema, but this time its medal winning role is mandatory. The mandatory, frequency, uniqueness, and value constraints together ensure that each country must have its medal tally recorded for each kind of medal. Hence Figure 14.13(b) has three mandatory roles. This example illustrates the following fifth corollary to theorem PSG1:

If *R* is a ternary with a UC spanning just *B*'s role and one other role, then adding a mandatory role constraint and frequency constraint of *n* (the number of possible values for *B*) to this other role is equivalent to making each specialized version of that role mandatory.

Figure 14.13 The impact of adding mandatory role and frequency constraints.

Figure 14.14 Fifth corollary to theorem PSG1.

This corollary is illustrated in Figure 14.14. It is implied by the previous two corollaries, since an equality constraint across a set of disjunctively mandatory roles implies that each of these roles is mandatory.

Other Kinds of Predicate Specialization/Generalization

As an introduction to a second equivalence theorem, consider Figure 14.15. The two schemas provide alternative models for a fragment of a car rally application. Each car in the rally has two drivers (a main driver and a backup driver), and each person drives exactly one car. The schema in Figure 14.15(a) is transformed into the schema in Figure 14.15(b) by absorbing the object type DriverStatus into the drives predicate, specializing this into the main driver and backup driver predicates. The reverse transformation generalizes the specific driver predicates into the general one by extracting the object type DriverStatus. Since this object type appears in a different fact type, this equivalence does not fit the pattern of PSG1.

Note how the constraints are transformed. The external uniqueness constraint in (a) says that each car has at most one main driver and at most one backup driver. This is captured in Figure 14.15(b) by the uniqueness constraints on the roles of Car. The uniqueness constraint on the drives predicate in Figure 14.15(a) corresponds in Figure 14.15(b) to the uniqueness constraints on the roles of Driver. The uniqueness constraint on the driver status predicate in Figure 14.15(a) is captured by the exclusion constraint in Figure 14.15(b). The mandatory and frequency constraints on Car's role in Figure 14.15(a) require the two mandatory role constraints on Car in Figure 14.15(b). Finally, the mandatory role constraints on Driver in Figure 14.15(a) are catered for in Figure 14.15(b) by the inclusive-or constraint (shown explicitly here as part of the exclusive-or constraint).

Figure 14.15 The drives predicate is specialized by absorbing DriverStatus.

Each S_1 corresponds to R where T is restricted to $B = b_1$

Corollary 1: If A's roles are mandatory in the LHS, the disjunction of A's roles in the RHS is mandatory, and conversely.

Corollary 2: If an external UC spans the roles of B and C in the LHS, then a UC applies to each of C's roles in the RHS, and conversely.

Corollary 3: If C's role in the LHS is mandatory, then the disjunction of C's roles in the RHS is mandatory, and conversely.

Corollary 4: An equality constraint over C's roles in the RHS is equivalent to a frequency constraint of $\geq n$ on C's role in the LHS; this constraint is strengthened to n if a UC exists on each of C's roles in the RHS.

Figure 14.16 PSG2: R may be specialized into $S_1..S_n$ by absorbing B.

This example illustrates our *second predicate specialization/generalization theorem* (PSG2*)* as well as four of its corollaries (see Figure 14.16). The terms "LHS" and "RHS" abbreviate "left-hand schema" and "right-hand schema". In our example, *A, B* and *C* correspond to Driver, DriverStatus, and Car; the equality constraint is implied by the two mandatory role constraints on Driver.

Theorem PSG2 can be derived from earlier results. For example, adding an exclusion constraint across A's roles in the RHS of Figure 14.6 makes A's role unique in the LHS, causing this compound fact type to split into two binaries to agree with the LHS of Figure 14.16. Section 14.4 considers two more general versions of this theorem.

Sometimes we may wish to transform a schema into another that is not quite equivalent. For example, suppose that in our car rally application we limit each car to at most two drivers, but do not classify the drivers in any meaningful way (e.g., as main or backup drivers). Let us also remove the constraint that drivers may drive only one car. This situation is schematized in Figure 14.17.

Although no DriverStatus object type is present, the frequency constraint in Figure 14.17 says that each car has at most two drivers. This enables us to introduce an artificial distinction to specialize the predicate into two cases, as shown in Figure 14.18.

Figure 14.17 Can the predicate be specialized?

Figure 14.18 Two ways of strengthening the schema in Figure 14.17**.**

Since this distinction is not present in the original schema, the alternatives shown in Figure 14.18 are *not equivalent* to the original. They are, in fact, *stronger*—each implies the original schema, but is not implied by it.

Notice the use of *hyphens* in the predicates of Figure 14.18 to bind adjectival phrases to their object terms. This improves verbalization of the fact types and constraints. The four predicates are Car has Driver-A, Car has Driver-B, Car has first-Driver, and Car has second-Driver. When the NORMA tool generates constraint verbalization, it binds the adjective to the object noun. For example, the upper uniqueness constraint in Figure 14.18(b) is verbalized as "**Each** Car has **at most one** first Driver". Without the hyphen, the verbalization defaults to the awkward "**Each** Car has first **at most one** Driver".

The schema in Figure 14.18(b) is actually stronger than the schema in Figure 14.18(a). If the Car role in Figure 14.17 is mandatory, then the Car roles in Figure 14.18(a) are disjunctively mandatory, and the top Car role in Figure 14.18(b) is mandatory (which then implies the subset constraint).

In a business domain where other facts are stored about cars but not about drivers, one of these alternatives may well be chosen to avoid a separate table being generated for car–driver facts when the schema is mapped (the specialized predicates are functional rather than *m:n*). In practice, the schema in Figure 14.18(b) would normally be chosen. The UML version of this choice is shown in Figure 14.19(b), using a note to declare the subset and exclusion constraints. Figure 14.19(a) shows the UML version of Figure 14.17.

Transforming from the original schema in Figure 14.17 to one of those in Figure 14.18 *strengthens* the schema by adding information. Transforming in the opposite direction *weakens* the schema by losing information. Any such transformations that add or lose information should be the result of conscious decisions that are acceptable to the client (for which the business domain is being modeled).

Figure 14.19 UML versions of the schemas in Figure 14.17 and Figure 14.18(b).

Corollary 1: If an equality constraint applies over *A*'s roles in the LHS, then the frequency constraint in the RHS is strengthened to *n*, and conversely.

Corollary 2: Adding a UC to *B*'s role in the RHS is equivalent in the LHS to adding UCs to *B*'s roles (making the S_i 1:1) and strengthening the exclusion constraint to an exclusion constraint over *B*'s roles.

Figure 14.20 PSG3: The left-hand schema implies the right-hand schema.

This example illustrates our *third specialization/generalization theorem* (PSG3), as shown in Figure 14.20. Note the use of "⇨" for "implies", since this result is a schema implication rather than an equivalence. In practice, the transformation is usually performed from right to left (strengthening the schema), with a subset constraint added from the first role of S_2 to that of S_1 if $n = 2$. Some corollaries for this result are also specified.

Well that covers the most important results concerning predicate specialization and generalization. Note that the theorems require that the appropriate translations hold between general and special predicates. Humans are needed to provide natural readings for the new predicate(s) and to confirm that the translation holds. This is not something that can be decided automatically by the system.

Exercise 14.2

1. Schematize the following table using (a) unaries and (b) a binary.

Male staff	*Female staff*
Halpin, T	Jones, E
Morgan, T	Smith, A

2. A company committee has to decide whether to increase its budget on staff training. The following table indicates the current views of the committee members on this issue. Schematize this using (a) unaries and (b) a binary.

For	*Against*	*Undecided*
Alan	Betty	Chris
David	Eve	Fred
Gearty		

3. The following table is an extract from an output report indicating quarterly sales figures for software products marketed by a particular company.
   (a) Schematize this using a ternary.
   (b) Transform your solution into an equivalent one using binaries.

Software	*Quarter*	*Sales ($)*
DataModeler	1	200 000
	2	500 000
	3	500 000
	4	700 000
WordLight	1	90 000
	2	150 000
	3	155 000
	4	200 000

4. An embassy maintains details about how well its employees speak foreign languages. The following table is an extract from this system.
   (a) Schematize this using binaries.
   (b) Transform this into a ternary.

Language	*Expert*	*Novice*
Arabic		Smith, J
Dutch	Bruza, P Proper, HA	
French	Rolland, C	Bruza, P

5. University debating teams are to be selected, with four students in each team, with one student from each year level (1..4). No student may be in more than one team. Students are identified by their student number, but their name is also recorded. Debating teams are identified by codes.
   (a) Schematize this UoD using YearLevel as an object type.
   (b) Transform your schema by absorbing this object type.

6. Employee records are kept that show the employee number, name, and up to two phone numbers for each employee.
   (a) Schematize this in the natural way.
   (b) Map this to a relational schema.
   (c) It is now required to map all the information into a single relational table. Transform the conceptual schema to enable this to happen.
   (d) Rmap your revised conceptual schema.

   Each phone is now to be classified as a work phone or a home phone. At most one work phone and at most one home phone may be recorded for an employee.
   (e) Modify your solution to (a) accordingly.
   (f) Rmap this.
   (g) Modify your solution to (c) accordingly.
   (h) Rmap this.

7. The following conceptual schemas are meant to describe the same UoD, but each fails to capture some constraint in the other. Add a textual constraint to schema (1) and a graphical constraint to schema (2) to obtain equivalence. The codes ‘chr’, ‘sec’, and ‘ord’ abbreviate “chairperson”, “secretary”, and “ordinary member”.

8. Consider the following academic UoD. Each subject is identified by its code but also has a unique title. Each subject has at most three assignments (possibly none), numbered 1, 2, and 3. Each subject has a second assignment only if it has a first assignment, and it has a third assignment only if it has a first and a second assignment. Each assignment has exactly one due date. Although assignments for different subjects may be due on the same date, no subject has more than one assignment due on the same date.

   Although unlikely, it is possible that within a subject the chronological order of due dates differs from the numerical order of the assignments (e.g., because of software problems the due date for CS400 assignment 1 might be postponed until after the due date for CS400 assignment 2).

   (a) Model this by adding constraints (graphic, and textual if needed) to the schema:

Reference schemes:	Subject(.code)
	Assignment (has AssignmentNr(), is for Subject)
Fact types:	Subject has SubjectTitle()
	Assignment is due on Date(mdy)

   (b) Map this to a relational schema, including all constraints.
   (c) Specialize the is-due-on predicate in (a) by absorbing AssignmentNr into it.
   (d) Map this to a relational schema, including all constraints.
   (e) Suppose that within each subject the chronological order of assignment due dates must match their numeric order. How does this affect answers to (c) and (d)?

14.3 Nesting, Coreferencing, and Flattening

As humans, we have the freedom to think of the universe in different ways. One choice in modeling is whether to represent some feature in terms of an object type, and if so, how. The previous section discussed how to absorb or extract object types by predicate specialization and generalization. An object type with a composite identification scheme may be portrayed explicitly either as a nested object type or as a coreferenced object type. If we don’t want to think of this feature in terms of an object type, we can model it with predicates using a flattened approach. This section we consider transformations among nested, coreferenced, and flattened approaches.

Table 14.2 A relational table of student results.

Result:

studentNr	*subject*	*rating*
1001	CS100	4
1002	CS100	4
1002	CS114	5

Let's start with a familiar example. The report in Table 14.2 may be modeled in three ways, as shown in Figure 14.21. The schema in Figure 14.21(a) takes the flattened approach, using a ternary fact type. In the schema in Figure 14.21(b), Enrollment is a coreferenced object type, where its primary identification scheme combines two reference types. In the schema in Figure 14.21(c), Enrollment is modeled as a nested object type or objectified association. As discussed in Chapter 9, UML uses association classes for nesting\ and has a limited form of coreference.

In this example, the nested or coreferenced object type plays just one fact role, and this role is mandatory. In such a situation, the relational mapping is the same whether we nest, coreference, or flatten. In this case, each maps to the table scheme *Result*(studentNr, subjectcode, rating). So we may choose whichever of the three approaches appeals most to us. Although this choice is partly subjective, there are a few guidelines that may be helpful.

First let's consider choosing between nesting and co-referencing. Both of these approaches always have the same relational mapping, no matter what the situation. Choose what seems most "natural". If in thinking about the real world, you first "see" a relationship and later want to talk about it, then nesting is probably the best choice. On this basis, more people would probably nest Enrollment rather than coreference it.

However, it often seems unnatural to think of a visible object in the real world as an objectified relationship. For example, consider Table 14.3, which provides details about the length of various annual reports by departments within a company.

Figure 14.21 Modeling Table 14.2 by (a) flattening, (b) coreferencing, and (c) nesting.

Table 14.3 Length of departmental annual reports.

Year	*Department*	*Report length (pp.)*
1993	Research	40
1993	Sales	25
1993	Support	40
1994	Research	67
…	…	…

Because you can hold a report (or at least a copy of a report) in your hand, it is natural to think of it first as an object and later ask yourself how to identify it. In this case we use both the year and the department to coreference it, as in Figure 14.22(a). The nested approach in Figure 14.22(b) treats a report as a relationship between Year and Department. The authors find this unnatural, although some modelers still prefer to do it this way. Since both map the same way, choose whichever suits your taste.

If at least one of the object types used for the composite identification is a value type, then coreferencing is often more natural than nesting. Recall the following co-referenced example: Subject (is offered by Department, has SubjectTitle). We could model this instead by objectifying the association Department offers a subject with SubjectTitle, but this seems unnatural. As another example, consider the composite reference scheme: Account (is at Branch, has AccountNr).

For efficiency, we sometimes introduce a new, simple identifier to replace a composite one, e.g., Subject(.code). In this case, the former reference types become elementary fact types. For example, in Figure 14.23(a) a report number is introduced to identify annual reports. We could use a plain serial number (e.g., 1, 2, …) or a coded number for humans to derive the semantics (e.g., "RES-2008", "SAL-2008",…). Though unnatural, a simple identifier may be introduced for Report even though it remains objectified, as shown in Figure 14.23(b).

Figure 14.23(a) includes the relationship types: Report is for Year; Report is by Department. These were reference types in Figure 14.22(a), but are now elementary fact types. For example, two fact instances might be Report 'RES-2008' is for Year 2008; Report 'RES-2008' is by Department 'Research'. If the semantic basis of such introduced identifiers is unstable, then plain serial numbers might be preferred (e.g., suppose departments are often split, merged, or renamed). However, if the semantic basis is stable, such "information-bearing" identifiers can lead to efficiency gains by reducing the need to perform relational joins when information about the object is spread over many tables.

Figure 14.22 Since a report is a physical object, is (a) more natural than (b)?

Figure 14.23 Annual departmental reports are now identified by a report number.

For example, suppose one table is used to store the data in Table 14.3 as well as an extra column for reportNr, and a separate table is used to store the *m:n* fact type Employee(.nr) authored Report(.nr). To find out who authored what reports, most users would be content with a listing of this second table, if meaningful report identifiers such as "RES-2008" are used, since this would be enough for them to deduce the year and department. However, if plain serial numbers are used, most users would want the year and department listed too, requiring a join to the first table. As another example, consider the use of subject codes (e.g., "CS115") in listing results.

If a new identifier is introduced, the new schema is no longer strictly equivalent to the original one. For example, the schemas in Figure 14.23 are stronger than the schemas in Figure 14.22. Any schema strengthening or weakening should result from a conscious decision of the modeler. However, shifting between a coreferenced and a nested approach (as in Figure 14.22) may be treated as an equivalence transformation. The general theorem, called **N/CR** (*nest/coreference*), is set out in Figure 14.24. Here the objectified predicate R has n roles, where n is at least 2. The object types $A_1..A_n$ are not necessarily distinct. Although not shown in the figure, we assume the nested and coreferenced object types have at least one role attached.

Where $n > 1$, and $Ra_1 \ldots a_n$ iff there is a b such that bR_1a_1 & .. & bR_na_n.

Note: If B is independent, show it as B ! in the coreferenced version.

Figure 14.24 N/CR: The nested version is equivalent to the coreferenced version.

Recall that an object type is independent if the disjunction of its fact roles is optional. If independent, the object type must be explicitly marked so by appending "!" to its name. For example, look back at Figure 14.21(b) and Figure 14.21(c), and suppose that enrollments are recorded before any results are known. The mandatory role constraint must now be removed from the rating association, and the Enrollment object type renamed "Enrollment !". As shown shortly, a more drastic change is needed to the schema in Figure 14.21(a) to maintain a flattened approach for this situation.

As long as the nested or coreferenced object type plays only one fact role, and this role is mandatory, we may transform to a single, flattened fact type. Recall the example in Figure 14.21. In such a case, the flattened approach is usually recommended since it gives a simpler diagram, is easier to verbalize, and it sometimes avoids arbitrary decisions about which part of the predicate to objectify.

When a ternary or longer predicate has just one uniqueness constraint, and this spans all but one of its roles, the subpredicate spanned by this UC provides the best choice for objectifying. This was our choice in Figure 14.21. In principle, we could pick any subpredicate to objectify. For example, suppose in Figure 14.25(a) that we objectify the subpredicate comprising the first and third roles, instead of the first two roles. This yields the awkward schema shown in Figure 14.25(b).

The external UC is needed to capture the UC in the flattened version (i.e., each student obtains at most one rating for each subject). Notice also that the UC on the outer predicate is compound. Although it maps to the same relational schema as the others, Figure 14.25(b) is more difficult for a human to understand.

If a ternary or longer predicate has overlapping UCs, then any objectification is best based on one of these. Each other UC in the flattened version is then captured by an external UC. See Figures 4.40 and 4.41 for an earlier example.

In the case of overlapping UCs, there is more than one "natural" choice of a subpredicate to objectify. This is also true if a ternary or longer predicate has a single UC spanning all its roles, for example: Lecturer visits Country in Year. Here we could objectify any of three role pairs: roles 1 and 2; roles 1 and 3; or roles 2 and 3. Unless there are additional facts about the subpredicates, the flattened fact type provides the simpler approach.

Figure 14.26 shows the ternary version of the **N/F** (*nest/flatten*) equivalence theorem. Here *A*, *B*, and *C* need not be distinct. A spanning UC over the roles in the nested object type *S* is assumed. Typically a UC spans the *A* and *B* roles in *R*; this is equivalent to a UC over the first role of *T*. For equivalence, the predicates must be formally related by the condition shown in the where-clause.

Figure 14.25 A poor choice for nesting (the UC spans other roles).

where *Rabc* iff (*a,b*)*Tc*

Corollaries: A UC exactly spans the *A, B* roles of *R* iff a simple UC spans the first role of *T*.
A binary UC over the *A, C* (or *B, C*) roles of *R* is equivalent to an external UC over the *A, C* (or *B, C*) roles, respectively, in the nested version.

Figure 14.26 N/F: The nested version is equivalent to the flattened version.

In Figure 14.26, if a UC spans all the roles of *R*, then a UC spans both roles of *T*. We do not bother stating this corollary in the figure, since each predicate has an implied UC spanning its full length. Recall that for generality, schemas used to depict theorems may omit constraints that are not relevant to the transformation.

Our equivalence theorems state formal connections between predicates in the different versions, but do not specify how predicate names used in one schema might help choose the different predicate names used in the other. A CASE tool could generate suggested names, but humans can often provide more natural ones (cf. generation of table names in relational mapping). In choosing identifiers, we should ensure that their background meaning agrees with the formal connection required by the transformation rule. This still leaves many possibilities. In Figure 14.25(a) for example, instead of "... for ... scored ..." we might use "... enrolled in ... obtaining ...".

Note that since nesting can always be replaced by co-referencing, we could specify **CR/F** (*coreference/flatten*) equivalence theorems analogous to any N/F theorem. For example, consider transforming between the two schemas in Figure 14.21. To save space however, we limit our unflattening discussion to nesting.

The ternary nest/flatten equivalence may be generalized to flattened fact types of any arity above two. Note that nesting always introduces an extra role (to be played by the objectified association). If we objectify *m* roles of an *n*-ary predicate, the outer predicate will have $n - m + 1$ roles. This is shown in Figure 14.27(a), where other object types and role connectors are omitted for simplicity.

Recall that each UC of an elementary predicate must span at least all but one of its roles. If a UC exactly spans all but one role of a predicate, and we objectify on this UC, the outer predicate in the nested version has two roles, with a simple UC on its first role (Figure 14.27(b)).

Figure 14.27 (a) Nesting ternaries and beyond; (b) nesting on a UC.

Figure 14.28 Binarizing a quaternary by nesting.

With long predicates, nesting may be applied more than once, as shown in Figure 14.28, where a quaternary is binarized in two stages. For simplicity, other object types and role links are omitted. We saw earlier how unaries may be converted into binaries. Nesting or coreferencing may be used to convert *n*-ary predicates into binaries. So, in principle, any domain may be modeled with binaries only. However, non-binary predicates often enable an application to be modeled in a more natural and convenient way.

In our examples so far, the objectified association played just a single, mandatory role. In this case, flattening is generally preferable. But what if this role is optional? For example, suppose we modify our earlier Table 14.2 by allowing nulls in the rating column (see Table 14.4). In this population, no ratings appear for the subject CS100. Perhaps the final exam for this subject is still to be held, or marking for it has not finished. In this UoD, it is possible to record the fact that a person enrolls in a subject before knowing what rating the student gets for that subject.

Since nulls are not allowed at the conceptual level, we cannot model this as a single ternary. We may, however, use two fact types—one for enrollments and one for ratings. Figure 14.29 shows this flattened approach, together with the sample population.

Table 14.4 Enrollments may be recorded before ratings are finalized.

Result:

studentNr	*subject*	*rating*
1001	CS100	?
1002	CS100	?
1002	CS114	5

Figure 14.29 Modeling Table 14.4 with flat fact types.

Figure 14.30 Modeling Table 14.4 with a nested approach.

The *pair-subset constraint* in Figure 14.29 indicates that students can score ratings only in subjects in which they enrolled. If this figure is part of a model with other roles for Student and Subject, the roles in the binary fact type may be optional; if the figure is not, they should be marked mandatory. This UoD may also be modeled using a nested or coreferenced object type. The nested version is shown in Figure 14.30, including the sample population. The role played by the objectified association is now *optional*. The objectified predicate corresponds to the binary predicate in Figure 14.29.

If no other role is played by Enrollment in the global schema, it is independent, as shown by the appended "!". This is not unusual with nesting. Any independent object type, whether simple, nested, or coreferenced, should be explicitly declared independent by appending "!" to its name.

When two or more flat fact types are involved, the equivalence between nested and flattened versions is called **N/Fm** (*nest/ flatten into many fact types*). The subset pattern illustrated by our example is only one of many cases for this theorem, but since it is the most common case we set it out in Figure 14.31.

We generally favor the nested version in this case, since it is more compact and it Rmaps directly to just one table. In contrast, if the flattened version is passed to our standard Rmap procedure, two tables result. Nevertheless, the flattened approach is quite natural and it verbalizes easily. If you prefer the flattened approach for these reasons, it would be nice if your CASE tool displayed it this way, but gave you the option of having it mapped it to a single table in the same way as the nested version. However, such flexibility is typically not provided by current CASE tools. So the nested approach is generally preferable for this case.

Figure 14.31 Another nest/flatten equivalence.

Table 14.5 Student assignment and exam marks.

Subject:	CS102		CS115	
StudentNr	*Assignt*	*Exam*	*Assignt*	*Exam*
1001	17	65	15	58
1002	20	79	20	67
1003	15	60	12	55

In the flattened version in Figure 14.31, the *S* predicate is compatible with the subpredicate comprising the first two roles of *R* (they are played by the same object types *A* and *B*). It is this compatibility that enables the nesting to occur. For this subset case, any (*a, b*) pair in the population of either must belong to the population of *S*. So we objectify *S* and add an optional connection to *C* to handle the *R* facts.

Recall that in our CSDP, whenever different predicates have compatible role sequences (of two or more roles), we should ask ourselves to what extent the populations of these [sub-] predicates must overlap. If a set-comparison (subset, equality or exclusion) constraint exists between them, it must be declared.

The kind of nesting transformation that may now occur depends partly on what kind of set-comparison constraint exists (if any) and on whether other roles occur in the full predicates. Let's look at some more examples before summarizing the overall procedure.

Consider the output report shown in Table 14.5. This may be modeled with two flat fact types, as shown in Figure 14.32(a). Notice the *equality constraint* between the two role-pairs. An assignment mark is recorded for a student in a subject if and only if an exam mark is recorded for that student in that subject.

The compatible subpredicates might be read as did assignment in and did exam in. Since the populations of these must be equal, we may objectify their conjunction did assignment and exam in. The nested version in Figure 14.32(b) uses the wording was assessed in. The specific marks are catered for by attaching two specific predicates as shown. Because of the equality constraint, these are both mandatory.

When passed to the standard Rmap procedure, the schema in Figure 14.32(a) maps to two tables, whereas the nested version in Figure 14.32(b) maps to a single table. For this reason, as well as compactness, the nested version is generally preferred whenever two or more functional roles are attached to the objectified predicate.

Figure 14.32 ORM schemas for Table 14.5: (a) flattened version and (b) nested version.

where *Rabc* iff (*a*,*b*)*Uc*, and *Sabd* iff (*a*,*b*)*Vd*

where *T* = *R[a, b]* ∪ *S[a, b]*

Corollaries: A binary UC over *R[a,b]* is equivalent to a UC on the first role of *U*; and a binary UC over *S[a,b]* is equivalent to a UC on the first role of *V*.
A pair-subset constraint from *S[a,b]* to *R[a,b]* makes *U* mandatory.
A pair-equality constraint between *R[a,b]* and *S[a,b]* makes *U* and *V* mandatory.
A pair-exclusion constraint between *R[a,b]* and *S[a,b]* adds an exclusion constraint between the attached roles in the nested version.

Figure 14.33 A more general nest/flatten equivalence.

A more general N/Fm equivalence result is shown in Figure 14.33. This may be generalized further to n predicates ($n \geq 2$) and compatible subpredicates of any arity above 1. The object types *A*..*D* need not be distinct. As long as [sub-]predicates are compatible, nesting may always be performed by objectifying their *disjunction*.

Since the predicate *T* in Figure 14.33 is a logical disjunction, it may be verbalized using the word "or" to connect verbalizations of the compatible [sub-]predicates. In terms of populations, the objectified relation *T* in Figure 14.33 is formed by taking the *union* of the compatible [sub-]relations. In relational notation: $T = R[a,b] \cup S[a,b]$.

Suppose we remove the equality constraint in Figure 14.32(a). If no other set-comparison constraint exists, we can nest this by replacing the two mandatory role constraints in Figure 14.32(b) by an inclusive-or constraint: **Each** assessment included **an** assignment **or an** exam mark **or both**. The uniqueness constraints in the flat version correspond to the simple UCs in the nested version (see first corollary in Figure 14.33).

In Figure 14.33, adding a pair-subset constraint from the first two roles of *S* to the first two roles of *R* is equivalent to adding a subset constraint from the first role of *V* to the first role of *U*. In the context of the disjunctive mandatory constraint, this means that *U* becomes mandatory (see second corollary). In terms of populations, it also means that the *T* relation becomes *R[a,b]*, since if one set is a subset of a second then their union is just the second set. Compare this with the result in Figure 14.31.

A subset constraint in both directions is an equality constraint. So adding an equality constraint between *R[a,b]* and *S[a,b]* in Figure 14.33 makes both *U* and *V* mandatory in the nested version. Figure 14.32 provides an example.

If an exclusion constraint exists between *R[a,b]* and *S[a,b]* in Figure 14.33, and we decide to nest, we must add an exclusion constraint between the attached roles in the nested version. For example, suppose that for any given subject, a student can be awarded either an actual grade or a notional grade, but not both (a notional grade might be awarded on the basis of performance in a similar subject from another university). As an exercise, schematize this in both flat and nested versions.

Table 14.6 Student enrollments and passes (if known).

Student	*Subjects enrolled in*	*Subjects passed*
3001	CS100, CS114, MP105	CS100, CS114
3002	CS114, MP105	CS114
3003	CS100, CS114	?

A relation is said to be *partial* if it is a projection of a longer relation; otherwise it is said to be *whole*. Sometimes we run into a case where a whole relation must be a subset of another. Consider Table 14.6 for instance. Here "?" is an ordinary null. In this UoD, we want to know what subjects students have passed, but are not interested in their actual ratings. Figure 14.34(a) schematizes this using a flattened approach.

We can nest this by objectifying enrollment, and then using a unary to indicate which enrollments resulted in a pass, as in Figure 14.34(b). By default, the flat version maps to two relational tables, while the nested version maps to one. So the nested version is usually preferred in such a case. You can decide for yourself whether to take an open- or closed-world approach when mapping the unary.

The **overlap algorithm** in Figure 14.35 summarizes the main cases of the N/Fm equivalence. This shows how to nest when two or more fact types include compatible role sequences, each with at least two roles. For simplicity, the diagram shows just two predicates with compatible role pairs (shaded). The predicates may have additional roles.

Let *P* and *Q* be the compatible [sub-]predicates. The nesting action depends on the amount of overlap allowed between the populations of *P* and *Q*. In any given case, this overlap condition is specified by replacing the pair connection marked "?" by a pair-subset (either direction), pair-equality or pair-exclusion constraint, or by no constraint (proper overlap is possible).

In the figure, a subset constraint upward from *P* to *Q* is denoted "↑", a downward subset constraint is shown as "↓", and an equality constraint appears as "↕". Recall that a predicate is partial if it is embedded in a longer predicate.

The five cases are based on which set-comparison constraint is explicitly *displayed*. An equality constraint must be specified if it exists. Hence in this context, display of a subset constraint is taken to imply that an equality constraint does not exist. In any situation, exactly one of the five cases listed will apply.

As one example of the "↑" case, consider the transformation from Figure 14.29 to Figure 14.30. Here *Q* is the whole enrollment predicate, while *P* is the partial predicate comprising the first two roles of the fact type: Student for Subject scored Rating.

Figure 14.34 ORM schemas for Table 14.6: (a) (flat version) and (b) nested version.

Overlap algorithm:

case ? constraint **of**

↑: -- *P*'s population must be a subset of *Q*'s, but need not equal *Q*'s
objectify *Q*;
if *Q* is partial **then** attach rest of its predicate via mandatory extra role;
if *P* is partial **then** attach rest of its predicate via optional extra role
else attach optional unary "is-*P*"

↓: reverse of previous case

↕ : -- *P*'s population must equal *Q*'s
if *P* or *Q* is partial
then objectify the single predicate "*P* and *Q*";
attach rest of their predicates via mandatory extra role(s)
else replace both by the single predicate "*P* and *Q*"

⊗: -- *P*'s population is mutually exclusive with *Q*'s
objectify "*P* or *Q*";
if *P* is partial **then** attach rest of its predicate via extra role
else attach as unary "is-*P*";
similarly for *Q*;
mark the attached roles as exclusive and disjunctively mandatory

: -- no constraint; proper overlap is possible
as for ⊗ but omit exclusion constraint

end

Figure 14.35 Overlap algorithm for nesting when compatible subpredicates exist.

As an example of the "↑" case where *Q* is partial, replace the enrollment predicate in Figure 14.29 by the ternary: Student enrolled in Subject on Date. The nested version in Figure 14.30 must now be modified by adding the mandatory binary: Enrollment occurred on Date. As an example when both *P* and *Q* are whole, consider the transformation of Figure 14.34(a) to Figure 14.34(b).

As an example of the "↕" case, consider the reshaping of Figure 14.32(a) to Figure 14.32(b). Here both *P* and *Q* are partial, and their conjunction is reworded as "was assessed in". As an example of the "↕" case where neither *P* nor *Q* is partial, remove the mark information from Figure 14.32(a) to give two binaries: Student did assignment in Subject; Student did exam in Subject. The equality constraint enables these to be replaced by a single binary: Student did assignment and exam in Subject (or Student was assessed in Subject). Derivation rules may be specified for the former binaries if desired.

As an example of the "⊗" case where both *P* and *Q* are partial, recall this exclusive disjunction instance of Figure 14.33: Student for Subject obtained actual Grade; Student for Subject obtained notional Grade. This may be modeled by objectifying Student was assessed in Subject and attaching the mandatory, exclusive disjunction: resulted in actual Grade; resulted in notional Grade.

As an example where both *P* and *Q* are whole, consider the exclusive disjunction: Student passed Subject; Student failed Subject. Suppose we objectify the disjunction of these predicates (i.e., Student passed or failed Subject) as: Student was assessed in Subject. This Assessment object type may now be classified by attaching an exclusive disjunction of the following unaries: is a pass; is a fail. As an exercise, draw these examples.

While the mutual exclusion case may be transformed by nesting in this way, this is rarely the best modeling alternative. Sometimes the flat version is preferable, and sometimes a predicate generalization transformation is more appropriate. For instance, the pass/fail example just cited may be modeled instead using the ternary: Student in Subject obtained Result {'pass','fail'}. Also, the nested solution to the actual/notional grade example may be transformed using PSG2 to replace the exclusive disjunction by: has Grade; is of GradeType {'actual', 'notional'}.

In the final case of the overlap algorithm, no set-comparison constraint applies between *P* and *Q*, so their populations may properly overlap. For example, suppose that for any given student and subject we might have a predicted grade, an actual grade, or both. This may be modeled by two flat fact types: Student for Subject has predicted Grade; Student for Subject has actual Grade.

We may nest this by objectifying Student is assessed for Subject and attaching the mandatory, inclusive disjunction: has predicted Grade; has actual Grade. As with the previous mutual exclusion case, other modeling alternatives may be preferable.

If *P* and *Q* are keys (i.e., each is exactly spanned by a UC) and the standard Rmap procedure is used, applying the overlap algorithm to flattened cases before mapping will reduce the number of relational tables. In this situation, especially for the subset and equality cases, nesting (or the equivalent coreferenced solution) is generally preferred.

Exercise 14.3

1. Consider the fact type: City in Year has Population. Population figures are collated only once each year. Populations may go up, down, or remain the same.
   (a) Model this using a flattened approach.
   (b) Now use a nested approach.
   (c) Now use a coreferenced approach.
   (d) Which solution do you prefer?
   (e) Use a nested approach that objectifies the association between City and Population. State whether this is better or worse than your solution to (b).

2. Consider the fact type: Flower blooms in City in Month. Note that flowers may bloom for more than 1 month in the same city.
   (a) Model this as a flat fact type.
   (b) Show three alternative nested solutions.

3. Schematize the following table using (a) flat, (b) nested, and (c) coreferenced approaches. Which do you find the most natural?

Software title	*ReleaseNr*	*Size (Mb)*
DataModeler	1	3
DataModeler	2	5
DataModeler	3	5
WordLight	1	5
…	…	…

4. Suppose employees are identified by their surname and initials, and each works for exactly one Department. In principle, we could model this using the ternary fact type: Surname and Initials belong to an employee who works for Department.
   (a) Draw the flat fact type. Is this natural?
   (b) Nest this instead. Is this natural?
   (c) Now use coreferencing. Is this natural?
   (d) We now decide to primarily identify employees by an employee number, but still require that surname and initials provide an alternative identifier. Model this.

5. The following conceptual schema refers to the recent Olympic Games.
   (a) Rmap this.
   (b) Transform the conceptual schema by nesting.
   (c) Rmap your solution to (b).
   (d) Which conceptual schema do you prefer?

6. The following is an extract from a yearly report giving test results of reaction time (in milliseconds) and resting heart rate (in beats per minute) for members of a health club. As the table indicates, the club may gain or lose members during the year.
   (a) Schematize this using two, flat ternaries.
   (b) Transform this by nesting.
   (c) Which schema do you prefer?

Month	*Member*	*Reaction time*	*Heart rate*
Jan	Jones, E	250	80
	Matthews, S	320	120
	Robinson, S	300	100
Feb	Jones E	250	75
	Matthews, S	300	100
Mar	Anderson, P	250	80
	Matthews, S	280	85
…	…	…	…

7. The following table indicates the rooms and times for lectures in various subjects. Schematize this using (a) flat, and (b) coreferenced approaches.

Subject	*Time*	*Room*
CS213	Mon 3 p.m.	B19
CS213	Wed 9 a.m.	A01
CS213	Wed 10 a.m.	A01
EN100	Mon 3 p.m.	F23
EN100	Tue 3 p.m.	G24
…	…	…

8. The report shown indicates student performance in subjects in a given semester. Once a student fails a subject in the semester, the subject cannot be passed by the student in that semester. As students pass or fail subjects these results are recorded. It is possible that a student might have neither passed nor failed a subject taken (e.g., the exam is yet to be held).
   (a) Schematize this UoD using three binaries.
   (b) Transform this by nesting, with two attached unaries.
   (c) Transform the two unaries into a binary.
   (d) For this UoD is it possible that the null for Adams AB might be updated to an actual value? What about the null for Casey J?

Student	*Subjects taken*	*Subjects passed*	*Subjects failed*
Adams AB	CS100, CS114, MP104	CS100, CS114	?
Brown SS	CS114, MP102	CS114	?
Casey J	CS100, CS114	?	CS100, CS114

9. The following table is an extract from an output report concerning the finals of a recent judo competition. For each weight division, the four clubs that made it to the finals are recorded, together with the results for first and second places. For a given weight division a club can obtain at most one place. No ties are possible.
   (a) An information modeler schematizes this in terms of three fact types: Club is finalist in Event; Club wins Event; Club is second in Event. Set out this conceptual schema.
   (b) Transform the winner and second-place predicates into a single ternary.
   (c) Transform this schema into an equivalent nested version.
   (d) Assuming complete information, show an alternative nested solution, by assigning the places "3A" and "3B" to the clubs that didn't get first or second in the event.
   (e) Transform this to a flattened fact type.
   (f) Which solution do you prefer?

Event	*Finalists*	*Winner*	*Runner-up*
Lightweight	Budokan, Judokai, Kodokan, Zendokan	Kodokan	Judokai
Middleweight	Budokan, Judokai, Kodokan, Zendokan	Kodokan	Zendokan
Heavyweight	Budokan, Kanodojo, Kodokan, Mifunekan	Mifunekan	Kodokan

10. The following examples are extracts of output reports from an information system about media channels. Each channel (TV or radio) has a unique, identifying call sign. Some radio channels broadcast using frequency modulation (FM) whereas others use amplitude modulation (AM). All TV channels are rated in the range 1..7 in three categories on two surveys. Channels are either commercial (com) or owned by the government (govt). All commercial radio channels have their audience composition assessed (see the sample pie charts). The mark "–" means "inapplicable because of other data".

    Schematize this, including uniqueness, mandatory role, value, subtype, and frequency constraints. Provide meaningful names and definitions for each subtype. If a fact type is derived, omit it from the diagram but provide a derivation rule. Do not nest. Minimize the number of fact types in your schema (if necessary, use transformations).

TV channels:

		Ownership details		
Call sign	*Ownership*	*Company*	*% share*	*Head office*
ATQ8	com	MediaCo	100	Brisbane
CTQ3	com	MediaCo	50	Brisbane
		TVbaron	50	Sydney
TVQ3	govt	–	–	–

Radio channels:

		Ownership details			*Modul-ation*	*Music played*
Call sign	*Ownership*	*Company*	*% share*	*Head office*		
4BZ	com	MediaCo	100	Brisbane	FM	rock country
RB3	govt	–	–	–	AM	–
STR5	com	MediaCo OzRadio	30 70	Brisbane Cairns	AM	country
4AA	govt	–	–	–	FM	–

TV survey ratings:

		Category			
Channel	*Survey*	*News*	*Drama*	*Sport*	*Totals*
ATQ8	A	5	3	3	
	B	4	4	3	22
CTQ3	A	5	3	4	
	B	5	4	5	26
TVQ3	A	3	4	4	
	B	2	4	5	22

Audience composition of commercial radio channels:

14.4 Other Transformations

The previous two sections covered the most useful conceptual schema transformations. In this section, several other transformations of lesser importance are considered briefly.

Recall PSG2, our second predicate specialization/generalization theorem (Figure 14.16). This may be generalized further by removing constraints. First note that removing the UC on *A*'s role in *R* corresponds to removing the UC's on *A*'s roles in each of the S_i predicates (see Figure 14.36). In the absence of additional internal UCs, the *R* and S_i binaries will be many:many. As an example, let *A* = Car, *B* = Status {'company', 'private'}, *C* = Employee, *T* = has, *R* = is used by, S_1 = is provided for, and S_2 = is privately used by. Allow the same employee to use many company cars and many private cars, and vice versa.

Each S_i corresponds to *R* where *T* is restricted to $B = b_i$.

Figure 14.36 Another PSG2 case: *R* may be specialized into $S_1..S_n$ by absorbing *B*.

Now suppose the equality constraint is weakened to an upward subset constraint. The *T* predicate must be retained since there may be instances of *A* that do not play *R*, and hence *T* will not in general be derivable from the S_i. Moreover, subtypes must now be introduced for the specialized predicates (Figure 14.37). The exclusion constraint over the subtype roles is omitted since it is implied by the subtype definitions.

For example, consider the car UoD just discussed, where some cars might not yet be used. In this case the subtypes are CompanyCar and PrivateCar. Cases that require one of these more general forms of PSG2 usually invoke some of the corollaries stated for the original version of the theorem (Figure 14.16).

Each A_i is an *A* that plays *T* with $B = b_i$.

Figure 14.37 Another PSG2 case: *R* is specialized into $S_1..S_n$, but *B* is not absorbed.

Figure 14.38 An example of value concatenation/separation.

For example, suppose each employee uses at most one company car and at most one private car. An external UC is now required between the Status and Employee roles in the left-hand version, and a simple UC must be applied to each role of Employee in the right-hand version. As an exercise, draw both schema versions for this UoD and prove that a frequency constraint of ≤2 is implied on the role played by Employee in the left-hand version.

If an object type in an *m:n* binary has two values, the fact type can be converted to a functional one by converting the object type to one that allows concatenated values. A simple example is shown in Figure 14.38. The transformation from left to right is called *value concatenation*, and the inverse transform is *value separation*. In this example, a physician may be trained in medicine, acupuncture, or both. Allowing the both-case to be recorded as a concatenated value (med-acu) is a "quick and dirty" way of effectively creating a set object type (in this case, DisciplineSet).

Appropriate translation rules are needed to license the transformation (e.g., Physician is trained in Discipline 'med' **iff** Physician is trained in DisciplineSet **in** {'med', 'med-acu'}). Moreover, the open/closed-world approach adopted should be the same for both. For example, if the left-hand version has a closed-world interpretation, then the code 'med' in the right-hand version means 'medicine only'.

Although sometimes used as a quick fix, this kind of transformation rapidly becomes unwieldy as the number of atomic values increases, since a list of *n* atomic values leads to $2^n - 1$ concatenated values (e.g., if we added herbalism and physiotherapy to give four atomic disciplines, we now have 15 concatenated disciplines). In such cases, if functional predicates are required, it is better to replace the binary with *n* unaries, one for each value, before mapping. Alternatively, if the target system supports set-valued fields (e.g., a nested relational or an object-oriented database), a mapping to a set-valued structure is possible.

In the presence of a pair-subset constraint, a binary may be *contracted* to a unary; conversely, the unary may be *expanded* to a binary (see Figure 14.39). Note that the UC on the first role of *R* is implied by the other constraints (as an exercise, prove this).

Figure 14.39 Binary–unary contraction and unary–binary expansion.

As an example of the left-hand schema, consider the fact types (*S* listed first): Politician is a member of Party; Politician is a minister in Party. Since one can be a minister only in one's own party, a pair-subset constraint runs from the minister to the member fact type. In the right-hand version, the minister fact type is contracted to the unary: Politician is a minister. Here a minister's party can be derived using the rule: Politician is a minister in Party **iff** Politician is a member of Party **and** is a minister.

In this case, contraction is usually preferred to expansion. However, if the binary to be contracted is 1:1, its second uniqueness constraint may be captured as a textual constraint in the unary version. For example, suppose the following fact types appear in the position of *S* and *R* in Figure 14.39: Politician is a member of / includes Party; Politician leads Party. Since each party has at most one leader, there is an extra UC on the lower binary, making it 1:1. Now suppose we contract this binary to the unary: Politician is a leader. The extra UC in the binary version appears in the unary version as a textual constraint: **Each** Party includes **at most one** Politician **who** is a leader. At the time of writing, ORM is being extended to capture such textual constraints graphically using a "unique-where-true" constraint (see the chapter notes).

Previous sections have shown how object types can be absorbed in order to specialize predicates and how nested and coreferenced object types may be removed by flattening. Apart from these cases, modelers sometimes eliminate an entity type from a fact type, phrasing the association in terms of the object type(s) formerly used to reference the entity type (usually however, it is more convenient not to do this).

Figure 14.40 provides an example with a simple reference scheme. Transforming from the left-hand to the right-hand version *eliminates* the entity type Degree. This weakens the schema, since the entity type and its reference scheme are lost. The translation defines the right-hand predicate in terms of the left. This is an implication rather than an equivalence. The reverse transformation from right to left *introduces* an entity type. In almost all cases, the left-hand version is preferred because of its richer formal semantics.

This transformation is sometimes performed with a coreferenced object type to which another object type is functionally related. Figure 14.41 provides an example. Here a room is identified by combining its local room number and building. Note the equality constraint; this is implied if the office roles of Employee are mandatory.

Figure 14.41(a) is generally preferable to Figure 14.41(b) for two reasons. First, it facilitates schema changes that add other facts about rooms (e.g., Room has Size(m^2:)). Second, it is natural to think in terms of rooms, and the composite identification scheme is clearly displayed. In general, don't eliminate an entity type unless you feel that no further details will be recorded for it and that no clarity is lost by its removal.

Figure 14.40 A simple case of entity type elimination/introduction.

Figure 14.41 A functional, coreferenced case; (a) is generally preferable.

If you do decide to eliminate a compositely identified entity type, ensure that it is the target of only *functional* predicates. For example, Figure 14.41(a) includes the fact type Employee works in Room. Suppose we now make this association *m:n* to cater for some unusual case where an employee may have more than one office. For example, if an employee has to move his/her office from one building to another, he/she may be assigned an office in each building to facilitate working efficiently in both places until the move is completed. It would now be wrong to eliminate Room as in Figure 14.41(b), since the two associations are now *m:n*, losing the information about which room number goes with which building.

That covers the main conceptual schema transformations of practical relevance in information modeling. Other transformations exist, but we have no space to discuss them here. Some transformations that deal with compound (nonelementary) fact types are discussed later in the context of denormalization and nonrelational mappings.

Exercise 14.4

1. (a) Each employee uses one or two phone numbers, and each phone number is used by an employee. Each phone number is classified as a work phone or home phone. No employee can have two phone numbers with the same classification. Schematize this using the following entity types Employee, PhoneNr, and PhoneType.
   (b) Provide an alternative schema by absorbing PhoneType.
   (c) Now suppose that some phones might not be used by any employee. Modify your answer to (a) accordingly.
   (d) Provide an alternative schema by specializing the uses fact type.

2. (a) Consider the fact types: Person is of Gender {'m', 'f'}; Person is a parent of Person. Each person's gender is recorded. Parents of the same child must differ in gender. Some people might not be recorded as a parent or child. Schematize this.
   (b) Set out an alternative schema by specializing the parenthood fact type.

3. (a) Each person has at least one PersonType {'lecturer', 'student'}. It is possible that a lecturer is also a student. Assume that all persons are identified by a personNr. Schematize this in terms of a many:many fact type.
   (b) Transform this to a schema comprising one functional fact type.
   (c) Provide an alternative schema using unaries.
   (d) People may now be classified as any or all of the following: lecturer, student, driver. How would you best model this?

4. (a) Consider the fact types: Employee works for Department; Employee heads Department. Each employee works for exactly one department and heads at most one department (the department he/she works for). Each department has workers and at most one head.
   (b) Schematize this using the two fact types given.
   (c) Set out an alternative schema that contracts the heads fact type to a unary. Which schema is preferable? Discuss.

5. The following table indicates where certain objects are placed in three-dimensional space (*x*, *y*, and *z* are the Cartesian coordinates). Only one object can occupy the same position at any given time.
   (a) Schematize this in terms of three binaries.
   (b) Set out an alternative schema using Position as an entity type.
   (c) Set out an alternative schema using AxisType {'x', 'y', 'z'} as an object type. Why is this alternative schema inferior?

Object	*x*	*y*	*z*
A	3	1	0
B	3	1	2
C	0	1	2

14.5 Conceptual Schema Optimization

The previous three sections discussed how to transform conceptual schemas to equivalent, or at least acceptable, alternatives. *Conceptual schema optimization* involves transforming a conceptual schema into an alternative conceptual schema that maps to a more efficient (ideally, the most efficient) implementation. Section 14.1 gave a simple example of this. Guidelines for performing such optimizations are now discussed.

Four **main factors** to consider when optimizing a conceptual schema are the *target system*, the *query pattern*, the *update pattern*, and *clarity*. The target system is the DBMS or other platform used for implementation. Here we assume it is a centralized, relational system. The query pattern includes the kinds of question the system is expected to answer, as well as statistical data about the expected frequency and priority of these questions. The update pattern includes the kinds, frequencies, and priorities of the expected insertions, deletions, and modifications to the database tables.

Response times for queries are usually more vital than for updates, so as a first criterion, we might try to minimize the response times of the "focused queries" (i.e., those queries with high priority or frequency). Clarity here refers to the ease with which the modeler can fully grasp the semantics conveyed by the schema.

Before the optimization procedure is executed, the global conceptual schema should be completed and validated. Any mapping choices (e.g., for 1:1 cases or subtypes) should also be declared. The procedure is then run on the whole schema. The complete optimization procedure for ORM schemas is somewhat complex, so only an overview of its main components is given here.

Although the default procedure could be executed without human intervention, better results may often be obtained by allowing modelers to override a suggested transformation when their additional insights reveal other factors that make the transformation counterproductive.

Several optimization strategies were introduced earlier in the chapter. Overall, the procedure comprises two main stages:

- Transform to reduce the number of mapped tables (steps 1 and 2)
- Transform to simplify individual tables (steps 3–5)

Since slow queries tend to involve joins or subqueries, a default strategy is to try to reduce the number of focused queries that involve joins or subqueries. Steps 1 and 2 of the optimization procedure do this for the main case (two or more tables involved) by transforming to reduce the number of tables involved in these queries. Since updates that require checking constraints between tables also tend to be expensive, this strategy usually improves update performance as well.

Recall that a composite key in a predicate amounts to a UC spanning two or more roles. Since the mapping algorithm maps predicates with composite keys to separate tables, a basic strategy is to reduce the number of (relevant) compositely keyed fact types in the conceptual schema. There are two main situations with the potential for achieving this: compatible keys that may be unified and nonfunctional roles that may be replaced by functional roles. Let's consider these in turn.

Step 1 aims to unify compatible, composite keys (see Figure 14.42). As preparation, step 1.1 includes two moves. As an example of its second move, consider a schema with the following fact types: Company in State has staff of Gender {'m', 'f' } in Quantity; Company in State has Budget. Compatible keys based on Company–State are formed by absorbing Gender to specialize the quaternary into two ternaries: Company in State has male staff in Quantity; Company in State has female staff in Quantity.

Steps 1.2 and 1.3 ensure that each pattern involving n compatible, composite keys is replaced by a single predicate ($n \geq 2$). Several examples were discussed in previous sections. Whereas the original pattern maps to n tables, the nested version maps to just one table.

As discussed earlier, if nesting leads to a loss of clarity in the conceptual schema, the modeler may prefer not to nest. Ideally, the modeler may then choose whether to have nesting performed automatically as an invisible, pre-processing stage to Rmap. Recall also that any nested object type may be recast as a coreferenced object type if this is felt to be more natural.

Step 2 examines *object types with both a functional and a nonfunctional role*, and typically attempts to replace the latter by functional roles that can be grouped into the same table as the former (see Figure 14.43).

Most of step 2.1 amounts to predicate specialization using enumerated object types or frequency constraints. Several examples were discussed earlier. The predicate generalization case with the restricted UC is rare; for an example, recall Figure 14.10.

The first case of step 2.2 for pattern (a) uses PSG3 with its second corollary (see Figure 14.20). The second case for pattern (a) is extremely rare and was ignored in earlier transformations.

1.1 Flatten any objectified predicate that plays just a single, mandatory role

if another compatible, composite key can be formed by absorbing a low cardinality object type into a predicate with a key of arity ≥ 3
then do so

1.2 **if** *n* nonfunctional, whole predicates form compatible, composite, exclusive keys and are incompatible with all other (sub-)predicates
then generalize them to a single longer predicate by extracting an object type of cardinality *n*

1.3 Apply the overlap algorithm to any pattern of compatible, composite keys (see Figure 14.35; sometimes the ⊗ and no-constraint cases are best left as is.)

Figure 14.42 Optimization step 1: unifying compatible, composite keys.

As an example, consider the fact type: Official holds Position {'president', 'secretary', 'treasurer'}. We may transform to three unaries: Official is president; Official is secretary; Official is treasurer. Assuming that each position is held by at most one official, we also need to add a role cardinality constraint declaring # ≤ 1 on each unary.

The final stage of step 2.2 (pattern (b)) uses PSG2 and its second corollary. As an example, consider the main and backup driver UoD of Figure 14.15 and add the functional fact type: Car is of CarModel. Specializing the drives predicate enables all the information to be mapped to a single Car table instead of a Driver and a Car table. As an exercise, draw the schemas and perform the mappings. Step 2.2 completes the optimization strategy to reduce the number of mapped tables.

2.1 For each case where an object type *A* has simple and binary keys attached, where the other role of the binary key is played by *B* (see here):

if *R* is a binary
then if *B* has values $b_1, \ldots, b_n$ (and *n* is small)
  then specialize *R* to *n* unaries **or** replace *B* by *B'* {b_1, b_2, both} (if $n = 2$)
    making the key(s) simple
  else if *A*'s role in *R* has a frequency constraint *n* or $\leq n$ (and *n* is small)
    then specialize *R* into *n* exclusive binaries simply keyed on *A*
    else if *A* has just one functional role
      and the predicates are compatible, pair-exclusive binaries
      then generalize both to a single ternary with a restricted UC
else -- *R* is a ternary
  if *B* has values $b_1, .., b_n$ (and *n* is small)
  then absorb *B*, specializing *R* into *n* binaries simply keyed on *A*.

2.2 For each case where an object type *A* has a functional role attached, as well as a 1:*n* binary predicate *R* connected to object type *B* (see here):

if *B* has no other functional roles (see (a))
then if *A*'s role in *R* has a frequency constraint of *n*, or $\leq n$ (and *n* is small)
  then specialize *R* into *n* 1:1 binaries with *B*'s roles mutually exclusive
  else if *B* has values $b_1, .., b_n$ (and *n* is small)
    then consider specializing *R* into *n* unaries
else if *B* has exactly one more functional predicate, linked to *C* {$c_1, .., c_n$}
  and an equality constraint spans *B*'s functional roles
  and an external UC spans *B*'s co-roles (see (b))
  then specialize *R* into *n* 1:1 binaries with *B*'s roles exclusive, by absorbing *C*
    (this is PSG2 with corollary 2: see Figure 14.16)

Figure 14.43 Optimization step 2.

Before going on to step 3, let's consolidate the first two steps by considering an example for which substantial optimization is possible. Figure 14.44 depicts a conceptual schema for postgraduate coursework in a computer science department. This UoD was first introduced in Exercise 4.2. The pair-subset constraint reflects the fact that after students enroll in a subject they choose a topic for it. Each topic has a lecturer in charge, and at most two (other) colecturers.

Figure 14.44 This sub-optimal schema maps to six tables.

As an exercise, map the schema in Figure 14.44 to a relational schema and then check your solution with the one in Figure 14.45.

If you did this correctly, you should have obtained six tables. Notice the intertable constraints. The "⊙" superimposed on "⊗" in the relational schema denotes a partition of PGtopic.topicCode (i.e., PGtopic.topicCode is the disjoint union of ActualPrerequsite.topicCode and PrefPrerequisite.topicCode). The ordinary "⊗" indicates that the topic–lecturer pairs in Colecturing and PGtopic are exclusive. The other constraints are straightforward.

Suppose a focused query for this domain is the following: list all topics (code and title) together with their main lecturers, colecturers, actual prerequisites, and preferred prerequisites.

Figure 14.45 The relational schema mapped from Figure 14.44.

Figure 14.46 The conceptual schema obtained by optimizing Figure 14.44**.**

Let another focused query be the following: list students, the subjects in which they are enrolled, and the topics chosen by them (if any) for those subjects. With the relational schema of Figure 14.45, the first query involves a join of four tables, and the second query involves a composite outer join of two tables. So these queries will be slow to run.

Applying optimization steps 1 and 2 to the conceptual schema of Figure 14.44 results in three main changes, as shown in Figure 14.46. Step 1.2 generalizes the two prerequisite binaries to a ternary by extracting the object type PrereqKind {'actual', 'preferred'}. Step 1.3 uses the overlap algorithm to nest the enrollment and topic fact types. Step 2.1 specializes the colecturer predicate into two exclusive binaries simply keyed on Topic. If we also demand that a topic has colecturer2 it must have a colecturer1, then this may be added as a subset constraint.

As an exercise, Rmap this then check your solution with that shown in Figure 14.47. The optimized relational schema has just three tables: one for the ternary, one for the nesting, and one for the five functional binaries of Topic. The "≠" symbol means that on the same row any nonnull values for headLecturer, colecturer1, and colecturer2 must differ. This relational schema leads to much faster execution of the focused queries, as well as simpler constraint enforcement on updates.

We now consider the final steps (3, 4, 5) in the optimization procedure. Unlike the previous steps, these do not reduce the number of tables. *Step 3* is set out in Figure 14.48. This aims to reduce self-joins and is called the *table width guideline* (TWG) since it indicates when a wider table (more columns) may be preferable.

Figure 14.47 The relational schema mapped from Figure 14.46**.**

3 For the ternary pattern given here, if focused queries on *R* require comparisons between different tuples, and the values $b_1,..., b_n$ are few and fairly stable, then absorb *B*, specializing *R* into *n* functional binaries.

Figure 14.48 Optimization step 3: the table width guideline.

The schema equivalence illustrated was considered earlier. If the object type *A* plays a functional role, the transformation to binaries would have already occurred in the last phase of step 2.1. Even if *A* has no functional role, the specialization into binaries may still be worthwhile if it avoids self-joins in focused queries.

As an example of step 3, suppose we wish to see how full-time university fees varied for various degrees over the triennium 2001–2003. Let's model this with the ternary Degree in Year had FulltimeFee with a uniqueness constraint spanning the first two roles, a frequency constraint of 3 on the first role, and a value constraint of {2001..2003} on Year's role. Suppose we map this to the following relational schema:

3 {2001..2003}
FulltimeFee (degree, feeYear, fee)

A fragment of a sample population for this table is shown in Figure 14.49. Let a sample focused query be: How much did the BS full-time fee increase over the triennium? Answer this query yourself by inspecting the table.

In arriving at the answer ($500), you compared values from two different rows from the same table. To perform this query in a relational database requires joining the table to itself (a self-join). An SQL version of the query is included in Figure 14.49.

FulltimeFee:

degree	*feeYear*	*fee*
BS	2001	5000
BS	2002	5000
BS	2003	5500
MBA	2001	7000
MBA	2002	7300
MBA	2003	7700

How much did the BS fee increase over the 3 years?

SQL code for query:

```
select New.fee – Old.fee
from   FulltimeFee as Old cross join FulltimeFee as New
where  Old.feeyear = 2001 and Old.degree = 'BS' and
       New.feeYear = 2003 and New.degree = 'BS'
```

Figure 14.49 This query requires a self-join, since it compares different rows.

Such joins can be avoided by first transforming the schema via TWG. Applying a variant of PSG1 where the value constraint applies to a *role* instead of an object type, the ternary specializes to the three conceptual binaries: Degree had 2001- FulltimeFee; Degree had 2002- FulltimeFee; Degree had 2003- FulltimeFee.

Here the roles of Degree are functional and linked by an equality constraint. If, in the global schema, fee information is mandatory for Degree, these roles are mandatory (which implies the equality constraint). If Degree has no other functional roles, these three conceptual binaries map to a single relational table:

FulltimeFee (degree, fee2001, fee2002, fee2003)

If Degree had other functional roles, the optimization would have been performed at step 2.1, with the extra functional fact types mapping to the same table.

Figure 14.50 illustrates how the wider table enables the query to be performed simply by comparing values on the same row.

This example was trivial and clear-cut, but life is not always so simple. If the fee year range is large, the number of extra columns generated by the transformation may be too high. For example, suppose that we must record fees for the period 1951..2000. Specializing the ternary would generate 50 binaries and lead to a relational table with 51 columns: *FulltimeFee*(degree, fee1951, .., fee2000).

Such a large number of binaries would clutter the conceptual schema (although a flexible CASE tool could be set to display the ternary with the optimization hidden in the mapping). The very wide relational table would also be awkward to view (lots of horizontal scrolling) and to print. For these reasons, once the *cardinality* of the value constraint exceeds a reasonable number (e.g., 5) the ternary fact type might well be preferred.

The *stability* of the value constraint is also important. In many cases, such a constraint is known to be stable (e.g., {'m', 'f'} for gendercode). Sometimes, however, the value constraint changes with time. For example, suppose fees are to be recorded for all years from 2001 to the current year, or for just the most recent triennium. The binary approach requires changes to the fact types and relational columns each year. In this case the ternary approach might be preferred since no structural change is needed (the year changes are made only to rows of data, not the table structure itself).

FulltimeFee:

degree	*fee2001*	*fee2002*	*fee2003*
BS	2000	2000	2300
MBA	3000	3300	3700

How much did the BS fee increase over the 3 years?

SQL code for query:

```
select fee2003 – fee2001
from   FulltimeFee
where  degree = 'BS'
```

Figure 14.50 The transformed schema leads to a more efficient query.

4. Generalize each group of n exclusive unaries ($n \geq 2$) to a functional binary by extracting an object type with n values.

5. **if** A plays exclusive, functional roles in n 1:m predicates with C,
   and C has no other functional role
   then replace these n predicates by two binaries, by extracting B $\{b_1,.., b_n\}$ as per the LHS of PSG2.

Figure 14.51 Optimization steps 4 and 5: further aspects of exclusive roles.

The wording of optimization step 3 reflects this trade-off between query efficiency and the ease of table display and schema evolution. Although default choices can be built into an automated optimizer, interaction with human modelers is advisable to ensure the best trade-off.

The final steps (4 and 5) of the optimization procedure deal with exclusive roles (see Figure 14.51).

Step 4 is straightforward. Recall that different unaries attached to the same object type map to different columns of the same table. If these unaries are exclusive, we may replace them all by a functional binary that maps to just one column. Moreover, the transformation of the exclusion constraint to a uniqueness constraint enables it to be simply enforced by the primary key constraint of the table. For an example of this predicate generalization process, review Figure 14.7 where three exclusive unaries (Employee is a manager; Employee is a clerk; Employee is a secretary) are replaced by the functional binary Employee has Rank {'M', 'C', 'S'}.

Step 5 helps us decide when to favor predicate generalization in applying theorem PSG2. This is illustrated, along with several other optimizations, within our next example. Consider the conceptual schema shown in Figure 14.52. The interpredicate uniqueness constraint asserts that a team may have only one player of each sex. This conceptual schema maps to seven tables. As an exercise, you might like to check this for yourself and then optimize the conceptual schema before reading on.

The overlap algorithm (step 1.3) applied to the bottom ternaries of Figure 14.52 generates the intermediate, nested subschema shown in Figure 14.53. This is then transformed (step 5) to the nested pattern included in the optimized conceptual schema (Figure 14.54). This modification to the nesting does not change the number of tables or columns, but it does simplify the constraint pattern that has to be enforced.

Figure 14.52 This unoptimized conceptual schema maps to seven tables.

At the final stage of step 2.1, the subtype PlayingCountry is seen in Figure 14.52 to have a functional role attached (via its supertype) and a role within a composite UC; the two-valued UniformItem is absorbed to specialize the ternary into two functional binaries. Note the constraint pattern arising from the application of corollaries.

At the first stage of step 2.2, the frequency constraint on the supplies predicate is used to specialize it into two functional binaries. The second half of step 2.2 is used to absorb the object type Gender, specializing the team membership predicate into binaries in which Team now has functional roles. Hence these may be mapped to the same table as Team's other functional fact type. Steps 3 and 4 are not invoked by this example. Step 5 transformed the original nesting to another, as discussed earlier.

This completes the basic optimization procedure. Generation of all the constraints in the optimized version is determined rigorously by the underlying transformation theorems and their corollaries. In the absence of such formal grounding it would be easy to lose some constraints in transforming such a schema.

Figure 14.53 An intermediate transform of the bottom left ternaries of Figure 14.52.

Figure 14.54 This optimized conceptual schema maps to three tables.

Although UML and commercial versions of ER are useful for summarizing schemas, they typically lack the required detail (e.g., disjunctive mandatory or pair-exclusion constraints) to illustrate many complex transformations, unless supplemented by textual constraints. The optimized conceptual schema maps to just three tables, as shown in Figure 14.55, thus reducing the number of tables by four.

Exclusion between column sets is shown by "⊗". The weaker constraint of exclusion between values on the same row is shown by "≠". Pairing attributes in the same square brackets entails that any qualification applies to each and that if one is null so is the other (this constraint is implied for the color attributes by their qualification).

The schema optimization procedure discussed here does not include decisions about how subtypes and symmetric 1:1 fact types should be mapped, since these do not change the conceptual schema itself. As discussed in Chapter 11 however, these mapping choices can have a significant impact on the efficiency of the relational schema obtained.

Figure 14.55 The relational schema mapped from Figure 14.54.

Recall that the optimization procedure assumes that the target system is a relational DBMS, at least in the sense that each entry in a table column is either an atomic value or a null. If multivalued fields are permitted (as in SQL:2003), other designs may be used to avoid table joins, leading to further possible optimizations (see Section 14.7).

Exercise 14.5

1. Consider a conceptual schema in which the only fact type associated with Department (code) is: Department at Level {UG, PG} has students in Quantity. Here "UG" and "PG" abbreviate "undergraduate" and "postgraduate", respectively. The role played by Department has a frequency constraint of 2. Let a focused query be: What is the ratio of postgraduate to undergraduate enrollments for the department of computer science?
   (a) Draw the conceptual fact type, and Rmap it.
   (b) Optimize the conceptual (sub)schema, and Rmap it.
   (c) Although primarily identified by its code (e.g., "CS"), each department also has a unique name (e.g., "Computer Science") which is now mandatorily recorded. The enrollment figures are now optional, but if any figures are recorded, both UG and PG figures are required. Draw the new conceptual schema and Rmap it.
   (d) Optimize the new conceptual schema, and Rmap it.

2. The following conceptual schema deals with applicants for positions as astronauts. Applicants are given ability tests (C = Cognitive, A = Affective, P = Psychomotor) and their performance on various tasks is also measured.
   (a) Rmap this.
   (b) Optimize the conceptual schema.
   (c) Rmap your optimized conceptual schema.

3. Consider the following conceptual schema. Assume that the Project-Programmer-Class subschema may be treated simply in snapshot fashion (i.e., only current data for these fact types are recorded).
   (a) Rmap this.
   (b) Optimize the conceptual schema.
   (c) Rmap your answer to (b).

(d) As indicated in the original conceptual schema, each project must normally have a junior and senior programmer. However, suppose now that while, or after, working on a project, a programmer may be promoted or even demoted in class (junior, senior). Discuss briefly how you would deal with this situation from a practical business standpoint and what changes, if any, you would suggest for the original and optimized conceptual schemas.

14.6 Normalization

The process of mapping from a conceptual schema expressed in atomic fact types to a relational schema involving table types is one of **deconceptualization**. This approach facilitates sound design primarily because it emphasizes working from examples, using natural language, and thinking in terms of the real-world objects being modeled.

The advantages of using *elementary facts* are clear: working in simple units helps get each one correct; constraints are easier to express and check; nulls are avoided; redundancy control is facilitated; the schema is easier to modify (one fact type at a time); and the same conceptual schema may be mapped to different logical data models (so decisions about grouping fact types together may be delayed until mapping).

Nowadays, many database designers do a conceptual design first, whether it be in ORM, ER, or UML notation, and then apply a mapping procedure to it. In the past, in place of conceptual modeling, many relational database designers used a technique known as **normalization**. Some still use this, even if only to refine the logical design mapped from the conceptual one. If an ORM conceptual schema is correct, then the table design resulting from Rmap is already fully normalized, so ORM modelers don't actually need to learn about normalization to get their designs correct. However, a brief look at normalization is worthwhile, partly to consolidate aspects of good design and partly to facilitate communication with those who use the normalization approach.

There are two techniques for normalization: *synthesis* and *decomposition* (sometimes called "analysis"). Each operates only at the relational level and begins with the complete set of relational attributes, as well as a set of dependencies on these, such as functional dependencies (FDs). Recall that if *X* and *Y* are simple or composite attributes (sets of one or more column headers), we say that *X* functionally determines *Y* (written $X \rightarrow Y$) if and only if for each value of *X* there is at most one value of *Y*.

In any realistic business domain it is too easy to get an incorrect model if we begin by simply writing down a list of attributes and dependencies, since the advantages and safeguards of conceptual modeling are removed (e.g., validation by verbalization and population). Moreover, the set of dependencies considered in normalization is usually very limited, so that many UoD constraints are simply ignored.

The output of both synthesis and decomposition techniques is a set of table schemes each of which is guaranteed to be in a particular "normal form". We will have a closer look at normal forms presently. The synthesis algorithm basically groups attributes into tables by finding a minimum, reduced, annular cover for the original dependencies. Redundant FDs (e.g., transitive FDs) are removed, redundant source attributes (those on the left of a "→") are removed, and FDs with the same source attribute are placed in the same table. With respect to its specified class of dependencies and attributes, the synthesis algorithm guarantees a design with the minimum number of tables, where each table is in elementary key normal form (see later for a definition of this concept).

For example, given the FDs {empNr → gender, empNr → birthdate, empNr → job, job → salary, empNr → salary}, synthesis generates two tables: *R1*(<u>empNr</u>, gender, birthdate, job); *R2*(job, salary). While this example is trivial, for more complex cases the execution of the synthesis algorithm is arduous and is best relegated to a computer. It should be apparent that the derived fact type checks in the CSDP, and the basic fact type grouping in Rmap, have a close correspondence to the synthesis method.

Although synthesis has often been taught in academia it is rarely used by practitioners, partly because of its highly technical nature, but mainly because it is almost useless in practice. The main problem with synthesis is that it assumes for its input a correct global set of attributes and dependencies. In practice, however, such an input is rarely available, and determining such an input is often harder than determining the correct design by other means. For such reasons, we essentially ignore the synthesis technique from this point on.

Another problem with normalization (by synthesis or decomposition) is that it can't change the set of attributes. Hence, apart from ignoring transformations that involve other kinds of constraint, normalization misses semantic optimization opportunities that might arise by allowing transformations to change attributes.

For example, the FD set {department → budget, (department, gender) → nrStaff} synthesizes to two table schemes *R1*(<u>department</u>, budget) and *R2*(<u>department, gender</u>, nrStaff). But normalization is unable to transform the FD set into {department → budget, department → nrMaleStaff, department → nrFemaleStaff}, which would lead to the single table design *R*(<u>department</u>, budget, nrMaleStaff, nrFemaleStaff). Compare this with the ORM schema optimization approach just discussed.

Considerations such as these indicate that use of normalization alone is inadequate as a design method. Nevertheless the use of normalization as a check on our conceptual modeling may at times be helpful. On the positive side, by limiting their scope to a small but important class of constraints, and ignoring semantic transformations, normalization theorists have been able to establish several interesting results.

Unlike synthesis, the decomposition approach to normalization is an iterative one, progressively splitting badly designed tables into smaller ones until finally they are free of certain update anomalies, and it provides no guarantee of minimality of number of tables. However, the decomposition approach is more well known, and some modelers find its principles useful for correcting poor relational designs. The rest of this section focuses on the decomposition approach, with some comparisons to ORM. To assist the reader who may read this section out of sequence, some basic concepts treated earlier are reviewed briefly.

A relational schema obtained by Rmapping it from a correct ORM conceptual schema is already fully normalized. Suppose, however, that some error was made in either the conceptualization or the mapping, or that the relational schema was designed directly without using a conceptual schema. It is now possible that some of the table designs might be unsafe because some conceptual constraint is not enforced. If a conceptual constraint may be violated when a table is updated by inserting, deleting, or modifying a row, the table design is said to contain an *update anomaly*.

Constraints on conceptual reference types are the responsibility of humans. Only people can ensure that preferred reference schemes actually do provide a correct 1:1-into map from real-world entity sets to sets of data values (or data tuples). Special care is required when an entity type plays just one fact role, this role is not functional, and an instance of this entity type changes its preferred identifier.

For example, suppose City occurs only in the *m:n* fact type: RussianTour(.nr) visits City (.name). In the early 1990s, the city then known as "Leningrad" reverted to its original name "Saint Petersburg". As an exercise, discuss some problems arising if only some instances of "Leningrad" are renamed.

Assuming preferred reference schemes are already enforced, the information system itself should normally enforce all the fact type constraints. So all constraints on (and between) conceptual fact types should be captured in the relational schema. The normalization procedure ensures that most of these constraints are so captured. In particular, it aims to remove any chance of redundancy (repetition of an elementary fact). If a fact were duplicated in the database, when the fact changed it would be necessary to update every instance of it, otherwise the database would become inconsistent.

Starting with a possibly bad table design, the normalization procedure applies rules to successively refine it into higher "normal forms" until it is fully normalized. Let's confine our attention to the normal forms that focus on eliminating problems with a table by splitting it into smaller ones. There are many such normal forms. In increasing order of acceptability, the main ones are first normal form (1NF), second normal form (2NF), third normal form (3NF), elementary key normal form (EKNF), Boyce/Codd normal form (BCNF), fourth normal form (4NF), fifth normal form (5NF), and domain key normal form (DKNF). We'll discuss each of these in turn.

Figure 14.56 Classification of tables based on their level of normalization.

A table that is not in first normal form is said to be "unnormalized" or in *Non-First Normal Form* (NFNF = NF^2). A table in a higher normal form is also in its lower normal forms (e.g., a 3NF table is also in 2NF and 1NF). Hence tables may be classified as shown in Figure 14.56. A table's *highest normal form* (hNF) is its deepest normalization level.

The first three normal forms were originally proposed by E. F. ("Ted") Codd, who founded the relational model of data. The other forms were introduced later to cater for additional cases. These improvements were due to the work of several researchers, including E. F. Codd, R. Boyce, R. Fagin, A. Aho, C. Beeri, and J. Ullman. Other normal forms have also been proposed (e.g., horizontal normal form and nested normal form) but are not discussed here.

A table is normalized, or in **first normal form** (**1NF**), if and only if its attributes are single valued and fixed. In other words, a 1NF table has a fixed number of columns, and each entry in a row-column position is a simple value (possibly null). Almost every table we've seen so far in this book has been in 1NF.

A table may violate 1NF (and hence be an NF^2 table) either by having multivalued attributes (e.g., entries that are collection such as sets, arrays, or even relations) or by allowing different rows to have different numbers of attributes.

As an example of the first kind of violation, consider Table 14.7. While the first two columns in this Student table store only atomic values, the Car column stores sets of values, and the Result column holds sets of (course, rating) pairs.

To model this in a relational database we need to flatten the structure out into three tables: *Student*(studentNr, gender); *Drives*(studentNr, car); and *Scores*(studentNr, course, rating). Hence, 1NF relations are often called *flat relations*.

Table 14.7 A nested-relational structure.

Student:

studentNr	*gender*	*Car*	*Result*	
		regNr	*course*	*rating*
1001	F	ABC123 PKJ155	CS100 CS114 PD102	6 7 6
1002	M		CS100 MA104	6 5

However, it is possible to model the structure directly by using a relation that is not in first normal form. For example, we might use the following *nested relation* scheme:

Student(studentNr, gender, *Car*(regNr), *Result*(course, rating))

Here an entry in the third column is itself a relation (in this case, a set of car registration numbers). Similarly, each entry in the Result column is a relation (with course and rating attributes). Since the logical data model underlying this approach allows relations to be nested inside relations, it is called the *nested relational model*.

Since the nested table is not in 1NF, it is not a relational table (i.e., it does not conform to the relational model of data). Although pure nested relational systems have yet to make any significant commercial impact, many of the major relational DBMSs have evolved into object-relational systems with support for NF^2 tables, as discussed in the next section. While tables not in 1NF might be in some other exotic "normal form", it is common practice to refer to them as being "unnormalized".

Set-valued fields such as Car in Table 14.7 are sometimes called "repeating attributes", and fields such as Result that hold sets of grouped values are sometimes called "repeating groups". This term may be used to include the notion of repeating attribute by allowing degenerate groups with one value. Using this older, if somewhat misleading, terminology we may say that 1NF tables have no repeating groups.

Another example of an unnormalized table is the "variant record type" used in languages such as Pascal and Modula. Here different "rows" in the same record type may contain different fields. For example, the Figure record type in Figure 14.57 is designed to store details about colored geometric figures. A sample data population is included. The record type includes common fields for figureId, color, and shape (circle, rhombus, triangle).

Figure:

figureId	*color*	*shape*			
		circle	*radius*		
		rhombus	*side*	*angle*	
		triangle	*side1*	*side2*	*side3*
F1	red	circle	15		
F2	red	rhombus	10	30	
F3	blue	triangle	40	30	50
...					

Figure 14.57 A variant record type, with a sample population.

The record type has different remaining fields depending on the shape (radius for circle; side and angle for rhombus; three sides for triangle). This provides one way to implement a restricted notion of subtypes, without using subclasses as is common in modern object-oriented programming languages.

Recall that a *key* of a table is a set of one or more attributes spanned by an explicit uniqueness constraint. Here "explicit" excludes any UCs that are implied by shorter UCs inside them; so keys are spanned by "minimal" UCs. For a given relation, an attribute is a *key attribute* if and only if it belongs to some key (primary or secondary) of the relation. A *nonkey* attribute is neither a key nor a part of a composite key.

For example, consider the relation scheme *City*(cityNr, cityname, statecode, population, [mayor]). Here cityNr is the primary key and (cityname, statecocde) is an alternate key. So cityNr, cityname, and statecode are key attributes. Although unique, the mayor attribute is not a key because it is optional. Since they don't belong to a key, population and mayor are nonkey attributes.

Although definitions of normal form and related dependencies are often given in terms of relations (fixed sets of tuples), in practice we should define them in term of *relation schemes*, since we want the table structures to exhibit the desired quality for each possible population (as seen later, this requires revised definitions for some of the higher normal forms). With this in mind, we define an FD over a relation scheme as follows. Given attributes X and Y of a relation scheme, X functionally determines Y ($X \rightarrow Y$) if and only if, *given any possible population* of the table scheme, for each value of X there is only one value for Y. Attributes X and Y may be composite. To decide whether a population is possible for a business domain requires the understanding of a domain expert (this is an informal semantic issue, not a syntactic issue).

A table scheme is in **second normal form** (**2NF**) if and only if it is in 1NF and every nonkey attribute is (functionally) dependent on the whole of a key (not just a part of it). If a table is normalized it is in at least 1NF. For example, the table in Figure 14.58(a) is in 2NF since gender and birthdate are both functions of student number. Here the FDs from studentNr are depicted as arrows.

The table in Figure 14.58(b) is in 1NF but not 2NF, since height, a nonkey attribute, is functionally dependent on athletename, which is just part of the composite key. Notice the redundancy: the fact that Jones E has a height of 180 cm is recorded twice.

To avoid redundancy, we split the original Athlete table into two smaller ones, as in Figure 14.59. This figure shows both conceptual and relational schemas for the example (assuming that the information is mandatory for athletes). If you verbalized Figure 14.58(b) correctly in terms of elementary facts, you would obtain this conceptual schema directly, and the relational schema would follow automatically using Rmap.

(a) *Student* (studentNr, gender, birthdate)

1001	M	1984-02-15
1002	F	1987-12-25
1003	M	1984-02-15
1004	F	?

(b) *Athlete* (athleteName, sport, height)

Jones E	hockey	180
Jones E	tennis	180
Smith T	tennis	170

Figure 14.58 Table (a) is in 2NF but table (b) is not.

Figure 14.59 Conceptual schema and correct relational schema for Figure 14.58(b).

You would have to be a real novice to verbalize Figure 14.58(b) in terms of a ternary, but suppose you did. The checks provided in the CSDP, especially steps 4 and 5, prompt you to discover that the functional fact type Athlete has Height. On seeing this, you would know to split the ternary into two binaries.

If you were careless in applying the CSDP, you could end up with the ternary in your relational schema. At this stage you could look for an FD coming from only part of a key, and use the 2NF normalization rule to at last correct your error. Even at this level, you should think in terms of the functional fact type behind the FD.

Recall that if fact types are elementary, all FDs are implied by UCs. This property is preserved by Rmap, since it groups together only functional fact types with a common key. So if we find an FD in a relational schema that is not implied by a UC, we must have gone wrong earlier. Since UCs imply FDs, every nonkey attribute is functionally dependent on any key of the relation. Note that to determine the hNF of a relation we need to be told what the relevant dependencies are (e.g., keys and FDs) or have a significant population from which these may be deduced.

Although normalization to 2NF overcomes the redundancy problem in the original Athlete table, this reduces the efficiency of those queries that now have to access both the new tables. For example, if we want the name and height of all the hockey players, the 2NF design means that two tables must be searched and their athleteName fields matched. This kind of efficiency loss, which is common to each normalization refinement (each involves splitting), is usually more than offset by the higher degree of data integrity resulting from the elimination of redundancy.

For example, if we need to change the height of Jones E to 182 cm, and record this change on only the first row of the original Athlete table, we now have two different values for the height. With more serious examples (e.g., defense, medical, business) such inconsistencies could prove disastrous.

As discussed in the next section, we may sometimes denormalize by introducing controlled redundancy to speed up queries. Although the redundancy is then safe, control of the redundancy is more expensive to enforce than in fully normalized tables, where redundancy is eliminated simply by enforcing key constraints. In short, normalization tends to make updates more efficient (by making constraints easier to enforce) while slowing down queries that now require additional table joins.

Given attributes X and Y (possibly composite), if $X \rightarrow Y$, then an update of X entails a possible update of Y. Within a relation, a set of attributes is *mutually independent* if and only if none of the attributes is (functionally) dependent on any of the others. In this case the attributes may be updated independently of one another.

(a)

Student (studentNr, gender, birthdate)

1001	M	1984-02-15
1002	F	1987-12-25
1003	M	1984-02-15
1004	F	?

(b)

Student (studentNr, birthcountry, population)

1001	US	301139947
1002	AU	20434176
1003	US	301139947
1004	CA	33390141

Figure 14.60 Table (a) is in 3NF; table (b) is in 2NF but not 3NF.

A table scheme is in **third normal form** (**3NF**) if and only if it is in 2NF and its *nonkey attributes are mutually independent*. Hence, in a 3NF table no FD can be transitively implied by two FDs, one of which is an FD between two nonkey attributes. For example, the table in Figure 14.60(a) is in 3NF since there is no dependency between gender and birthdate.

However the table in Figure 14.60(b) is not in 3NF, since population depends on birthcountry, and these are both nonkey attributes. Note that population is transitively dependent on the key (studentNr determines birthcountry, and birthcountry determines population). This table exhibits redundancy: the fact that the United States has a population of 301,139,947 is shown twice. To avoid this redundancy, the fact type underlying the FD between the nonkey attributes is split off to another table, as in Figure 14.61, which shows both ORM and relational schemas for the example.

Now consider the table in Figure 14.62, which is a modified version of an example used in Section 5.5 to illustrate the logical derivation check in the CSDP. If this CSDP step has been carried out properly, FDs from one nonkey attribute to another cannot arise in any tables obtained from Rmap. In the original Section 5.5 example, we split this table into two, one for the fact type Lecturer works for Department, and one for the fact type Department is located in Building, deriving the fact type Lecturer works in Building from the rule Lecturer works in Building **iff** Lecturer works for **a** Department **that** is located in Building.

However, if the "only if" part of the "iff" in this biconditional is not satisfied, then the two-table design loses information. For example, suppose that Wang and Zalewski do not work for any department (e.g., they might be visiting lecturers who work in some general office). This situation is represented by the last two table rows in Figure 14.62, using "?" to indicate a null. If we split the table in two as described earlier, it would be impossible to derive the facts about where Wang and Zelewski work.

The original work on normalization assumed that nulls do not occur, but this assumption is often false. If we agree with Codd (1990, p. 201) that nulls should be ignored when applying rules about FDs, then the FDs shown in Figure 14.62 still exist; but it is wrong to simply decompose this table into two, so this normalization step can be unsafe. Clearly, the safest way to deal with such issues is at the conceptual level.

Figure 14.61 Conceptual schema and correct relational schema for Figure 14.60(b).

Figure 14.62 Are facts underlying the FD surname → building transitively derivable?

Codd's original definitions of 2NF and 3NF used "key" in the sense of primary key, so attributes of alternate keys were regarded as nonkey attributes. However, nowadays an attribute is termed a key attribute if it belongs to *some* key (primary or alternate), so a nonkey attribute belongs to *no* key.

Hence, in one sense the definition of 3NF does allow transitive dependencies if the intermediate attribute is a key. Consider the table in Figure 14.63. It has two keys. We could argue that the FD countrycode → population is transitively implied by the FDs countryCode → countryName and countryName → population. Alternatively, we could argue that countryName → population is implied by countryName → countryCode and countryCode → population. However, the relation is still in 3NF as there is no FD between nonkey attributes. The ORM schema is shown at the left of the figure.

Let X and Y be attributes (possibly composite) of the same table. A functional dependency $X \rightarrow Y$ is *trivial* if and only if X contains Y. For example, the table scheme *Student*(studentNr, gender) has six trivial FDs: studentNr → studentNr, (studentNr, gender) → studentNr, etc. A functional dependency $X \rightarrow Y$ is *full* (rather than partial) if and only if there is no FD from just part of X to Y.

A full, nontrivial FD is said to be an *elementary FD*. Consider, for example, the table in Figure 14.64 about populations of U.S. cities (July 2006 figures). This has three elementary FDs that start at the primary key cityNr, and two elementary FDs that start at the alternate key (cityname, stateCode). Some nonelementary FDs in that table are cityNr → cityNr and (cityName, stateCode) → cityName. We typically avoid displaying nonelementary FDs because they are obviously implied by the elementary FDs.

A key is an *elementary key* if there is an elementary FD from it to some attribute in the table. An attribute is an elementary key attribute just in case it belongs to some elementary key. For example, the city table in Figure 14.64 has two elementary keys cityNr and (cityName, stateCode). We may now define the next strongest normal form, which was proposed as an improvement on 3NF (Zaniolo 1982).

Figure 14.63 The FD countryCode → population is a harmless transitive dependency.

Figure 14.64 Five elementary FDs: three from cityNr and two from (cityName, stateCode).

A table scheme is in **elementary key normal form** (**EKNF**) if and only if all its elementary FDs begin at whole keys or end at elementary key attributes. In other words, for every full, nontrivial FD of the form $X \rightarrow Y$, either X is a key or Y is (part of) an elementary key. For example, the City table in Figure 14.64 is in EKNF. If the primary key of a table includes all its columns, the table is automatically in EKNF since it has no elementary FDs; for example: *Enrollment*(studentNr, subjectCode).

Now consider the OfficialLanguage table in Figure 14.65(a), which records the languages officially used in various countries. Countries may be identified by their codes or by their names. This table has two composite keys: (countryCode, language); (countryName, language). The primary key is indicated by a double underline; since its attributes are not adjacent, a dotted line connector is added to show that this is a single, composite key rather than two simple key constraints. The secondary key is indicated by a single underline.

Since the OfficialLanguage table has no nonkey attributes, it is automatically in 3NF. However, it has obvious problems (e.g., the fact that the country with code 'CA' is named 'Canada' is duplicated). One way of spotting this bad design is to note that the table is not in EKNF. There is an FD from countryCode to countryName, and another FD from countryName to countryCode (depicted by arrows above the table). These elementary FDs come from parts of keys rather than whole keys, and these FDs do not end at elementary key attributes, since neither of the composite keys is elementary. So the table is not in EKNF.

Figure 14.65 (a) Table in 3NF but not EKNF; (b) Rmap provides a correct schema .

The redundancy problem can be avoided by splitting the table into two EKNF tables as shown in Figure 14.65(b): *Country*(countryCode, countryName); *LanguageUse*(countryCode, language). The two FDs missed in the original table scheme are now captured by key constraints in the Country table, and the redundancy problem has been eliminated. The same design is obtained by Rmapping the ORM schema shown. If the language fact type is not mandatory for Country, replace the equality constraint in the relational schema by a simple subset constraint from LanguageUse.countryCode to Country.countryCode.

The next normal form is named after Raymond Boyce and Edgar Codd, who proposed it as an improvement on the old 3NF. An equivalent definition was given earlier by Heath (1971). A table scheme is in **Boyce–Codd Normal Form** (**BCNF**) if and only if *all its elementary FDs begin at whole keys* (i.e., given any full, nontrivial FD $X \rightarrow Y$, it must be that X is a key). For example, the relation scheme *Student* (studentNr, gender, birthdate) is in EKNF because all its elementary FDs begin at the studentNr key. The only time that a relation can be in 3NF (or EKNF) but not in BCNF is when it has at least two candidate keys that overlap (Vincent and Srinivasan 1994).

Now consider the table in Figure 14.66(a). This is like the previous table, except that the percentage of the country's population that actually uses the language is recorded (the figures here for Canada are fictitious). The keys are now elementary, since they functionally determine the percentUsage attribute. Since the FDs between countryCode and countryName now end at elementary key attributes, the table is in EKNF. But the table has the same redundancy as the previous table.

One way to spot this problem is to note that it is not in BCNF, since the FDs shown above the table do not start at whole keys. For instance, the functional dependency countryCode → countryName exists, but CountryCode is only part of the (countryCode, language) key.

The redundancy problem can be avoided by splitting the table into two BCNF tables, as shown in Figure 14.66(b). This design is automatically obtained by Rmapping the ORM schema shown. As before, if the language use fact type is optional, change the equality constraint to a subset constraint.

Figure 14.66 (a) Table in EKNF but not BCNF; (b) Rmap provides a correct schema.

The principle underlying the refinement to second and third normal forms has been nicely summarized by Bill Kent (1983) as follows: "A nonkey field must provide a fact about the key, the whole key, and nothing but the key". This should be refined by replacing "the key" by "a key", and treated as a description of BCNF rather than 3NF.

In the previous two examples, a table with overlapping keys had to be split. Sometimes overlapping keys are permitted. Recall the example from Chapter 4 where students are assigned unique positions in subjects, with no ties. This leads to the BCNF table scheme

Enrollment(studentNr, subjectCode, position)

The previous normal forms considered only functional dependencies. The next normal form is due to Ronald Fagin (1977) and considers *multivalued dependencies* (MVDs). For a given relation with attributes *X*, *Y*, and *Z* (possibly composite), *X multidetermines Y* (written as $X \twoheadrightarrow Y$) if and only if the *set* of *Y* values is a function of *X* only (independent of the value of *Z*). In this case, *Y* is said to be multivalued dependent, or multidependent, on *X*; in such a case *Z* will also be multidependent on *X*. A functional dependency is a special case of an MVD, namely when the set of dependent values is a unit set. An MVD $X \twoheadrightarrow Y$ is trivial if and only if *X* includes *Y*, or *X* and *Y* together include all the attributes in the relation.

For example, the Athlete table considered earlier in Figure 14.58(b) is repeated in Figure 14.67(a), this time showing the nontrivial MVD athleteName → sport. As the ORM schema in Figure 14.67(b) indicates, MVDs usually arise from *m:n* associations. Although this poor design would be detected earlier because of its embedded FD (it is not even in 2NF), there are relations with MVDs that are in BCNF yet still have redundancy problems. To address such cases, we now consider the next normal form.

A relation is in **fourth normal form** (**4NF**) if and only if it is in BCNF and all its nontrivial dependencies are functional (single-valued) dependencies. So a 4NF relation cannot have any nontrivial MVDs that are not FDs. Basically, each nonfunctional MVD requires a separate table for itself.

Consider the Lecturer table in Figure 14.68(a). It is in BCNF since it is "all-key". The table is not in 4NF since language and sport are each multidependent (but not functionally dependent) on surname. The MVDs are depicted as double arrows above the table. Assuming that the ternary means Lecturer plays Sport and speaks Language, the table exhibits redundancy. The facts that Halpin plays judo and karatedo and speaks English and Japanese are duplicated. To avoid this problem, the table is split into two 4NF tables (as obtained by Rmapping the ORM schema) as shown in Figure 14.68(c).

Figure 14.67 The MVD athleteName → sport in (a) is due to the *m:n* association in (b).

Figure 14.68 (a) Table in BCNF but not 4NF; (b) debatable MVDs; and (c) Rmap result.

Now let's delete the fourth row of this Lecturer table to obtain the smaller Lecturer table in Figure 14.68(b). Notice the question mark next to the MVDs. If we strictly apply Fagin's definition of MVD, the relation (here a set of four tuples) does not exhibit these MVDs. According to Fagin (1977), the MVD $X \twoheadrightarrow Y$ holds for the relation $R(X, Y, Z)$ if and only if whenever (x, y_1, z_1) and (x, y_2, z_2) are tuples of R, then so are (x, y_1, z_2) and (x, y_2, z_1). By this definition, if the MVD surname ↠ sport did apply, then the presence of the tuples ('Halpin', 'karatedo', 'English') and ('Halpin', 'judo', 'Japanese') requires the presence of the tuples ('Halpin', 'judo', 'English') and ('Halpin', 'karatedo', 'Japanese'). But the second of these tuples is absent (we just deleted it). Similarly, the MVD surname ↠ language does not apply because of this missing tuple. However even without these MVDs, the table shows fact redundancy (the facts that Halpin plays judo and speaks English still appear twice).

Clearly, what is important in practice is that the relation *scheme* be well designed, regardless of whether the current population of that scheme satisfies some required dependency. If somebody produced the table in Figure 14.68(b) the table would have no nontrivial MVDs according to Fagin's definition, but the design is still faulty if the ternary means Lecturer plays Sport and speaks Language, since it allows fact redundancy.

For such reasons, we adopt the notion of what we call *semantic 4NF*, which relates MVDs to the *meaning* of the underlying fact types. If the ternary in Figure 14.68(b) means Lecturer plays Sport and speaks Language, then there is a semantic MVD surname ↠ sport because, in terms of the underlying fact type Lecturer plays Sport, the set of sports is determined by the lecturer alone (without regard to language). Similarly there is a semantic MVD surname ↠ language.

Suppose instead the ternary meant Lecturer plays Sport when it is conducted in Language. For example, Halpin might play judo only when it is conducted in English. In this case, the ternary is elementary so cannot be split without information loss. As this example shows, it is not in general possible to determine whether a table is well designed simply by inspecting its population(s). You also need to know what the table means.

ORM's Rmap procedure forbids grouping two or more fact types with composite UCs (e.g., *m:n* associations) into the same table scheme, regardless of what the current populations might be. ORM's notion of an alethic, static constraint refers to *any possible population*, and in principle requires an understanding of the business domain by the domain expert who asserts that such a constraint must apply. While sample populations can reveal the absence of a constraint, and provide hints as to the presence of constraints, they cannot guarantee that such constraints really do apply—such guarantees are the responsibility of the domain experts.

It can be shown that BCNF is equivalent to no redundancy when there are FDs, and 4NF is equivalent to no redundancy when there are FDs and MVDs (Vincent and Srinivasan 1993). Moreover, if a relation scheme is in BCNF and some key is simple, then it is also in 4NF (Date and Fagin 1992).

The next normal form is also due to Fagin (1979) and is based on *join dependencies* (JDs). A relation has a join dependency if it can be reconstructed without information loss by taking a join of some of its projections. If one of these projections is the table itself, this is a trivial join dependency.

A table is in **fifth normal form** (**5NF**) if and only if, for each nontrivial join dependency, each projection includes a key of the original table. A table is in **project-join normal form** (**PJNF**) just in case each JD is the result of the key constraints. Any 3NF relation scheme whose keys are all simple is also in PJNF (Date and Fagin 1992). The forms 5NF and PJNF are often treated as equivalent, but some subtle differences can be distinguished to show that PJNF is a stronger notion (Orlowska and Zhang 1992).

As an example of a key-based join dependency, *Employee*(empNr, birthdate, gender) is equivalent to the join of the projections: *Emp1*(empNr, birthdate); *Emp2*(empNr, gender). The original table as well as the two smaller ones are all in 5NF. Since a relational schema is in a given normal form if its table schemes are all in that form, a relational schema that includes just the Emp1 and Emp2 tables is a 5NF schema. This illustrates that 5NF of itself does not guarantee that a schema is minimal with respect to the number of tables.

Nontrivial join dependencies that are not key-based are rare, so 4NF tables are almost always in 5NF as well. The theory underlying the test for 5NF was included in the projection-join check at CSDP step 4, so we confine ourselves here to a brief discussion. A classic example used to discuss the notion is portrayed in the SellsFor table in Figure 14.69(a). A small sample population is provided to show that the only uniqueness constraint is the weakest possible (verify this for yourself).

Conceptually, this ternary fact type may be expressed as: Agent sells Cartype for Company. The fact types behind the three binary projections may be expressed as: Agent sells Cartype; Agent represents Company; Company makes Cartype. Any attempt to split this ternary into *two* binaries will result in information loss. For example, from the facts that Smith is a representative for Foord, and Foord makes a four-wheel drive (4WD), it does not follow that Smith sells a 4WD.

However, the population of the SellsFor table does equal the join of the three binary projections (confirm this for yourself). So we could split this particular table population into three binary table populations without information loss.

(a)

SellsFor (agent, cartype, company)

Jones E	sedan	Foord
Jones E	4WD	Foord
Jones E	sedan	Yotsubishi
Smith J	sedan	Foord

(b)

*Agent sells Cartype for Company **iff**
Agent sells Cartype **and**
Agent represents Company **and**
Company makes Cartype.

(c)

Sells (agent, cartype)

Represents (agent, company)

Makes (company, cartype)

Figure 14.69 (a) Is this a 5NF Table? (b) Possible ORM schema that maps to schema (c).

But since database populations are updated continually, if we are to split the table scheme into three binary table schemes, we need to know that this join dependency applies to all possible, complete populations. In other words, is the sample population significant in this regard? If it is, the derivation rule shown in Figure 14.69(b) applies.

The only way to check that this is an actual business rule is to ask the domain expert. Let's suppose that it's *not* a rule. For example, it is now acceptable to delete just the first row of the SellsFor table in Figure 14.69(b). Check for yourself that this deletion should be rejected if the rule did apply. In this case, the ternary fact type is elementary and should be left as it is. If the derivation rule does not apply, the SellsFor table *scheme* is already in 5NF.

Now suppose the derivation rule *does* apply, as shown in Figure 14.69(b). This rule needs to be enforced, but the key constraint in the SellsFor ternary fails to do this (e.g., it allows the first row to be deleted). To avoid this problem, the 4NF ternary should be split into three 5NF binaries, as shown in Figure 14.69(c). If some of the roles in the ORM schema are mandatory, some inter-table constraints need to be added. If desired, the original ternary may be defined as a view derived from the join of the new tables.

It has been argued (e.g., by Kent) that when relation schemes such as SellsFor in Figure 14.69(a) are elementary, there is a kind of "unavoidable redundancy". For example, the "fact" that agent 'Jones E' sells sedans is repeated on rows 1 and 3. However, such "redundancy" is harmless because it is derived rather than stored. Although the tuple ('Jones E', 'sedan') is stored twice, the fact that agent 'Jones E' sells sedans can only be obtained via the derivation rule Agent sells Cartype **iff** Agent sells Cartype for Company.

The seven normal forms discussed so far are strictly ordered. Each higher normal form satisfies the lower forms. Recall that "⊂" denotes "is a proper subset of". Let "5NF-rels" denote the set of all possible relations in fifth normal form, and so on. The strict ordering may now be set out thus (in decreasing order of normality):

$$\text{5NF-rels} \subset \text{4NF-rels} \subset \text{BCNF-rels} \subset \text{EKNF-rels} \subset \text{3NF-rels} \subset \text{2NF-rels} \subset \text{1NF-rels}$$

Essentially, normalization through to 5NF simplifies table updates by ensuring that three kinds of dependency (FDs, MVDs, and JDs) will automatically be enforced by key constraints (i.e., uniqueness constraints within each table). An FD is a special case of an MVD, which in turn is a special case of a JD. As noted by Fagin, BCNF ensures that all FDs are implied by key constraints, 4NF ensures that all MVDs are implied by key constraints, and 5NF ensures that all JDs are implied by key constraints.

The last normal form we consider is **domain key normal form** (**DKNF**), which was introduced by Fagin (1981). A table scheme is said to be in DKNF if and only if all its constraints are expressible as domain dependencies or key dependencies. A domain dependency specifies a set of possible values for a given attribute. This basically declares a data type (e.g., varchar(20)) or a value constraint (e.g., {'M', 'F'}) for the attribute. A key dependency is a uniqueness constraint spanning one or more attributes in the table scheme. If all domains are infinite, any DKNF table scheme is also in 5NF.

For a relational database schema to be in DKNF, each of its table schemes should be in DKNF, and there should be either no intertable constraints or only "simple" intertable constraints that are easily enforced, such as inclusion dependencies (subset constraints).

DKNF designs are easy to understand, and their constraints can normally be efficiently enforced by a DBMS. Unfortunately, however, life is not so simple. Many other kinds of constraint often arise in practical applications, and need to be enforced even though their inclusion in the relational schema violates DKNF.

As a simple example, Figure 14.70 shows a conceptual schema and its relational mapping. The table scheme Person is not in DKNF because it contains two constraints other than domain and key dependencies. First, the subset constraint from manager to personNr is an intratable inclusion dependency. Second, the "personNr ≠ manager" constraint enforces the irreflexive nature of the reporting association. The overall schema is not in DKNF because its Person scheme is not in DKNF, and because of the intertable exclusion constraint (no person wrote and reviewed the same book).

Although the relational schema in Figure 14.70 is not in DKNF, each of its table schemes is in 5NF. The additional constraints are simply not included in the definition of 5NF so do not violate it. Any valid ORM schema, where each asserted fact type is atomic, will result in a 5NF relational schema if mapped according to our Rmap procedure.

Figure 14.70 The relational schema is not in DKNF.

Table 14.8 An unusual and awkward example.

Lecturer	*Course*	*Student*
Halpin	CS113	Brown A
Rose	CS102	Brown A
Nijssen	CS113	Smith J
Rose	CS102	Smith J
Halpin	CS113	Wang J
Bloggs	CS226	?

As this example shows, ORM actually goes beyond 5NF by catering for other kinds of constraint. Although this rich support for expressing and mapping constraints may lead to schemas that violate DKNF, that's actually an advantage rather than a weakness, since such violations are needed to capture the relevant business rules.

It is very common to encounter cases where normalization to 5NF or DKNF fails to fully capture the relevant constraints. A modified version of an awkward but classic example is set out in Table 14.8. In this UoD, lecturers are assigned exactly one subject, and this is the only course they may teach.

The null indicates that Bloggs does not teach the assigned course (e.g., because too few enrolled for it). For each course they take, students have exactly one lecturer. The same course may have many assigned lecturers. Because of its strange constraint pattern, this example is not easy to verbalize in elementary facts on a first attempt. You are invited to try this before reading on.

An initial conceptual schema based on one verbalization is depicted in Figure 14.71(a). If this is the global schema, the mandatory role constraints are implied by the pair-subset constraint. More importantly, the uniqueness constraint on the binary when combined with the pair-subset constraint has other implications. Can you spot these?

The pair-subset constraint indicates that lecturers may teach only their assigned courses, but the uniqueness constraint on the binary indicates that each lecturer is assigned at most one course. Hence lecturers may teach at most one course. So there is an FD from the first role of the ternary to the second role, as shown in Figure 14.71(b). Since the ternary has an implied, spanning UC, the implied FD implies the UC across the first and third roles. Moreover, since there is a third role in the ternary, the fact type now cannot be elementary. This is an example of the general rule set out in Figure 14.72. Two corollaries are included, one of which we will make use of in our example. You may wish to prove these corollaries as an exercise.

Figure 14.71 Is the ternary fact type elementary?

Corollaries:
(1) A UC across roles 4 and 5 maps to an external UC between roles 2 and 5.
(2) A UC across roles 3 and 4 implies a simple UC on role 3 alone.

Figure 14.72 A basic splittability check.

If a fact type is elementary, all its FDs are the result of its UCs. If another FD exists, then semantically underlying this FD is another fact type, which is the target of a pair-subset constraint from the original fact type. If this is expressed at the conceptual level, we know the original fact type is not elementary and should remove the relevant role (as in Figure 14.72).

The ternary in Figure 14.72 matches this pattern, but has an additional UC across its last two roles (for each subject, a student has only one lecturer). To preserve this constraint after removing the role, an external UC must be added between the course and student roles in the new schema (see Figure 14.73).

The general requirement for this extra move is specified as the first corollary in Figure 14.72. When the corrected conceptual schema is Rmapped, we obtain two table schemes with two intertable constraints (Figure 14.73).

Suppose that instead of dealing with Table 14.8 conceptually, we develop a relational schema for it using the normal form theory discussed earlier. We might begin with a single ternary: *Teaches*(lecturer, course, [student]). Suppose that someone now tells us all the constraints or cleverly constructs a significant population from which the constraints can be deduced. We note that there are two composite uniqueness constraints over (lecturer, student) and (course, student). Since student is an optional attribute, neither of these can be used as a primary key, so we know something is wrong (although the normal form theory discussed earlier ignored nulls). Indeed we know the table is not in BCNF since we have the FD lecturer → course, and lecturer is not a key.

At this stage we split the table into two: *Assigned*(lecturer, course); *Instructs*(lecturer, student). These are the same two tables we arrived at by deconceptualization in Figure 14.73. However, the normal form theory discussed in this section tells us nothing about the two intertable constraints. We have to figure these out for ourselves.

Figure 14.73 The corrected conceptual schema and its relational map.

Further normal forms could be invented to cater for further constraints, but surely it is better to focus on cleaning up design faults at the conceptual level where humans can utilize their semantic understanding of the UoD more easily.

As discussed in the next section, it is sometimes necessary to denormalize a relational schema in order to achieve acceptable performance. This is a possible option for the current example. Note that the intertable constraints are expensive to enforce, unlike key constraints. For example, enforcing the external uniqueness constraint effectively requires a table join each time a row is inserted into either table. Moreover, queries to list who teaches what to whom need a table join.

Another design is obtained by mapping the original ORM schema in Figure 14.71(a) to give the relational schema shown in Figure 14.74(a). The pair-subset constraint implies the ternary's other key constraint on (lecturer, student), which may thus be omitted. Although the ternary is denormalized, it is safe since its problematic FD is enforced by the pair-subset constraint and the key constraint in the binary. We may now list who teaches what to whom, without a table join. If this is a focused query for the application, this design might be preferred.

To modify the example somewhat, suppose lecturers must teach the course assigned to them (so the null in Table 14.8 cannot occur). Figure 14.71(a) is changed to make both roles of Lecturer mandatory, with the subset constraint becoming an equality constraint, and Figure 14.73 is changed to make both roles of Lecturer mandatory, and the relational subset constraint becomes an equality constraint.

If the normalized, two-table design has poor performance, we might denormalize the design to a single table that is their natural join, as shown in Figure 14.74(b). With the subset constraint of Figure 14.71(a) replaced by an equality constraint, assignment facts can be retrieved by projecting on the first two roles of the ternary, but the uniqueness constraint on the assignment binary still needs to be catered for. In the denormalized ternary, this constraint cannot be enforced as a uniqueness constraint, so a separate constraint is required to enforce the FD lecturer → course. As long as this additional constraint is enforced (e.g., as a user-defined constraint in SQL), the single-table design is safe. As before, the alternate key constraint is implied and hence omitted.

Advances in relational languages are making it easier to specify constraints such as FDs that are not key based. However the overhead of enforcing such constraints needs to factored in when choosing the final relational design. In general, we should first try a fully normalized design, using conceptual optimization where relevant to improve it. If performance is still unacceptable, controlled denormalization is an option.

Figure 14.74 Some denormalization options.

Exercise 14.6

1. The conceptual schema for a given UoD is as shown. Some table schemes designed by novices for parts of this UoD are listed here. Indicate briefly, in terms of normalization theory, why each of these table schemes is not fully normalized.
   (a) *Parenthood* (parentName, gender, *Child*(childName, gender))
   (b) *Teaches* (person, subjectTutored, [subjectLectured])
   (c) *ParentOf* (parent, child, genderOfParent, genderOfChild)
   (d) *Tutor* (tutorName, subject, [child])
   (e) Use Rmap to generate a correct relational schema.

2. (a) The conceptual schema of Question 1 is incomplete with respect to real-world constraints. For example, parents of the same child must differ in gender, but this constraint is not captured. Replace the parent predicate with mother and father predicates, to develop a more complete schema that includes this constraint (Hint: include subtyping).
   (b) Rmap your solution to (a).

3. Suppose that while developing a conceptual schema you specify the following fact type: Manager supervises Trainee on Project. Assume that no manager is a trainee, and that all objects are simply identified by name. The following constraint applies: no manager may supervise the same trainee on more than one project.
   (a) Draw this as a ternary fact type, including the constraint.
   (b) You now check whether there are any other functional fact types of interest between these object types, and discover that each manager supervises at most one project. Add this fact type to your schema and then add any set-comparison constraint that applies.
   (c) Is the ternary elementary? If not, explain why not, and correct the schema.
   (d) The following business rule is now added: each trainee on a given project has only one supervisor for that project. Add this constraint to your schema.
   (e) Rmap your answer to (d).
   (f) Because of poor performance, this relational schema is regarded as unsatisfactory. Using controlled denormalization, suggest an alternative relational schema.

14.7 Denormalization and Low Level Optimization

Applying the conceptual optimization procedure followed by the Rmap procedure ensures a redundancy-free relational schema that should be reasonably efficient. To test the performance of this logical schema, it should be implemented as an internal schema on the target DBMS, using a realistic population. At this stage, *indexes* should normally be created for all primary and foreign keys, as well as any other columns that play a significant role in focused queries.

For large tables, indexes can dramatically improve performance since they can normally be loaded, in part or whole, into main memory and their entries are sorted into structures that permit fast retrieval (e.g., binary trees). At times, indexes may slow down updates since they must be updated when relevant changes are made to the base tables, but even with updates, indexes can speed up many kinds of constraint checking (e.g., ensuring that keys are unique). There are many kinds of indexes (e.g., clustered, nonclustered), and indexing schemes often differ among vendors.

To speed up or facilitate certain complex queries, *working tables* (either predeclared or ad hoc) may be used (e.g., to hold intermediate results, perhaps obtained by joining several tables, which might be reused several times before the base values are updated). Predeclared working tables are populated at the start of a session to hold intermediate results that may be treated as stable for the session; these temporary tables are dropped at the end of the session. Ad hoc working tables are similar, but are created during a session and may be dropped before the end of the session as soon as they are no longer needed. Another popular option is to use *materialized views*—these are views that are populated automatically with data according to their definition, and updated automatically as needed when their base tables are updated. Support for materialized views varies with the DBMS; in SQL Server they are called indexed views.

Different DBMSs offer other means of tuning the internal schema (e.g., clustering the storage of related data). In addition, since no query optimizer is perfect, the form of the actual query itself can impact (sometimes seriously) on the performance. With some older optimizers, merely changing the order in which tables are listed in a join condition can cause significant changes in the response time. In most cases you can use the performance tuning assistance provided by the DBMS to compare execution plans and select the best alternative.

If the data has been prechecked for integrity, consider deactivating constraint checking when performing bulk updates (e.g., loading a data warehouse from operational databases). You might also delay expensive constraint checking until after performing a bulk update. Although most database applications can be developed completely in 4GLs, consider writing some modules in a 3GL (e.g., C#) to speed up critical aspects or to achieve the desired control (e.g., checking acyclic constraints is usually faster in a 3GL than using recursive union in SQL). For a few specific applications, a relational database system might not be capable of giving the desired performance, and a different kind of DBMS might be needed (e.g., an object database).

Since realistic applications typically require data updates by users to be performed via an external interface (e.g., screen forms) rather than directly on the base tables, it needs to be decided how much of the constraint checking will be done at the table level and how much at the form level. With client–server networks we need to decide how much work is done at the client end and how much at the server end. Constraints are often implemented both at the user interface and at the DBMS. Implementing constraints on screen forms can detect data entry errors as early as possible. Implementing constraints on the DBMS ensures that they are checked regardless of which user interface is used. With noncentralized database systems, great care is needed in deciding how to replicate data to reduce communication overheads. A proper discussion of these issues is beyond the scope of this book.

If the performance of the system is still unacceptable after the aforementioned internal schema optimization has been carried out, it may be necessary to *denormalize* the base relational schema. Typically this entails introducing *controlled redundancy* to reduce the number of table joins required for focused queries. This slows down updates (redundant facts must be kept consistent) and makes the system harder to work with, but it can dramatically reduce response times for queries. Any denormalization decisions should be carefully documented, and appropriate subset or equality constraints must be enforced to control the redundancy, unless the denormalized version is used only for queries, not for updates (e.g., a data warehouse).

As a simple example, consider the following relational subschema, which is extracted from our software retailer example.

Suppose we frequently need to list the line items, including item titles, for specified invoices. With the schema given earlier, this requires a join of the two tables. If the tables are very large, accessing the second table simply to get the item title may slow things down too much. To avoid this, we might denormalize by inserting an extra copy of title in the LineItem table. This means the fact type Item has Title is now mapped to both tables. To control this *redundancy between tables*, we add a pair-subset constraint between the tables as shown:

The new LineItem table is denormalized to 1NF, since it includes the functional dependency itemCode → itemTitle, and itemCode is only part of its primary key. Although the constraint pattern is more complex, the query may now be performed by accessing just the LineItem table. The asterisked uniqueness constraint and FD are both implied by the other constraints (why?), so need not be specified. The primary and alternate key constraints are inexpensive, but the pair-subset constraint may be expensive. It is not a foreign key constraint, but in the SQL standard it may be declared as the following assertion:

```
create assertion LineItem_itemCode_itemTitle_pair in_Item
 check (not exists
        (select * from LineItem
         where itemCode not in
              (select itemCode from Item
               where itemTitle = LineItem.itemTitle)))
```

In most DBMSs, assertions are not supported, so this constraint would normally be coded with triggers. The subset constraint needs to be checked whenever items are deleted from or renamed (code or title) in the Item table, or inserted or renamed in the LineItem table. Since items are rarely deleted or renamed, the main overhead for the subset constraint involves validating insertions to the LineItem table. For some applications the additional update overhead may be a price well worth paying to improve the query performance. Alternatively, explicitly declare the implied constraint **unique**(itemCode, itemTitle) on the Item table, leading to the following schema:

Although this adds some unneeded overhead (the extra constraint is implied by either of the candidate key constraints), it enables us to declare the inter-table subset constraint simply as a foreign key constraint (since its target is declared unique).

The above example denormalized by storing item title facts twice in different tables. Another denormalization technique is to embed facts that functionally depend on a nonkey attribute, hence violating 3NF. For example, the conceptual schema in Figure 14.75(a) by default maps to the normalized relational schema in Figure 14.75(b).

The FD from roomNr to phoneNr is efficiently enforced by the primary key constraint in the Room table, but a query to list an employee's phone number requires a join. To speed up this query we could denormalize to 2NF by embedding the phone facts in the Employee table, as shown in Figure 14.75(c).

Since many employees may share the same room, this denormalized schema is open to *redundancy within a table*. For example, if employees 1 and 2 share room 200 which has phoneNr 4251234567, then the fact that room 200 has this phone number is stored twice. To control this redundancy, we need to enforce the FD roomNr → phoneNr. Since this is no longer enforced by a key constraint, we need to specify it separately. In the SQL standard, the following assertion may be used:

```
create assertion eachRoomHasAtMostOnePhoneNr
 check( not exists
  (select roomNr from Employee
   group by roomNr
   having count(distinct phoneNr) > 1))
```

Figure 14.75 (a) A conceptual schema and its (b) normalized and (c) denormalized mapping.

If assertions are not supported (which is likely), the embedded FD can be enforced by insert and update triggers on the Employee table.

As shown from these examples, denormalization can speed up queries but may slow down updates. So there is a trade-off in performance. In general, don't denormalize unless it is the only way to obtain the desired performance. If you do decide to denormalize, and your database will be used for updates, then make sure you control the denormalization by coding the relevant constraints.

Real-world applications sometimes use very poor database designs. In particular it is not uncommon for novice modelers to introduce uncontrolled redundancy (e.g., to ignore the pair-subset constraint and embedded FD in the above denormalized schemas). The next section discusses a reengineering example that illustrates, among other things, how such poor designs can be improved by applying conceptual modeling techniques.

If you are using an object-relational or object-oriented DBMS, you have the option of denormalizing to NF^2 using complex types. For example, SQL:1999 allows array types and row types to be used as columns that may themselves be nested as components of array or row structures used for outer columns. SQL:2003 added support for multisets. Recall the Voter table example in an earlier chapter. Some DBMSs already support these collection types. This gives you a far greater range of mapping choices, which need to be used judiciously.

As a simple example, Figure 14.76(a) models details about books in two normalized tables, using a position column to explicitly store each author's position in the book's author list. Up to 10 authors are allowed for a book, and some or all of these might not be recorded.

By using an array-valued column for the author list, the denormalized table scheme in Figure 14.76(b) allows all the details to be captured in one table. Although this solution might seem to be preferable, extra care is needed to enforce constraints. For example, the entries in the array must be unique. The best choice depends on the query/update pattern for the application. For example, the normalized solution makes it easier to list all the books of a given author.

Considerable care is required in the use of collection types within table design, especially in constraint enforcement. We'll have a few more words to say about this in the next chapter. Given the current lack of a uniform approach to such object-relational extensions by commercial DBMSs, we'll say no more about object-relational DBMSs here.

Figure 14.76 Modeling author lists with (a) normalized tables or (b) an NF^2 table.

In addition to denormalization, a technique known as *fragmentation* is commonly used to improve performance on very large tables. For example, a PhoneBook table for all of the United Sates would contain hundreds of millions of rows. To speed things up, we can partition this table into 50 smaller tables, one for each state (if needed, we could partition further into smaller regions). Each of these smaller tables has the same structure, with the name of the table providing the state information. If this process sounds familiar, it should be. It's simply an example of predicate specialization, as discussed early in the chapter. At any rate, if you want to query or update phone details about one state only, you can do this on the table for that state. This gives much better performance than trying to work with a table for the whole country.

If you need to work with multiple states at a time, you can create a view for it based on a union of the smaller tables. Some DBMSs enable good performance with such views. For example, if you include a column for the statecode in each table, and a check constraint to ensure the right statecode, SQL Server can use this information in the union view to consider only the base tables for the states mentioned in your query. For a good discussion of this technique, based on partitioning trades by month, see Seidman (2000).

Physical optimization techniques vary considerably among DBMSs. For example, Oracle's *materialized view* support differs somewhat from Microsoft SQL Server's *indexed views* support. Different systems support different kinds of *indexing schemes* (e.g., clustered, hash table), and Microsoft SQL Server even provides a tuning wizard to make the best index selection based on a significant sample of transactions. Detailed suggestions on physical optimization should be available in the online documentation for your specific DBMS.

Exercise 14.7

1. The following relational schema is used to store details about students and their results. A focused query is as follows: List the studentNr and name of all students who enrolled in CS114. As there are over 50,000 students, and on average each enrolled in over 20 subjects, the tables are large. Denormalize the schema to improve the performance of the query, and discuss any possible disadvantages of this action.

2. This question is based on an example from a colleague, Dr. Bob Colomb. A simpler version of the problem was discussed by Zaniolo in motivating EKNF and BCNF. As this problem involves some advanced ORM concepts, it should be considered a challenge. The application deals with a directory for making international, direct dial telephone calls. Suppose the following table scheme is used to store the data. The first key denotes the city (e.g., Australia, Queensland, Brisbane) and the second key gives the dial codes to phone that city (e.g., 61, 7).

PlaceCodes (countryName, stateName, cityName, countryCode, areaCode)

From a normalization point of view, the table has problems because there are some FDs that are not key based. In particular, countryName → countryCode. It is not the case, however, that countryCode → countryName (e.g., both Canada and the United States have the country code 1, since here "country code" really means "region code"). Note also that three values are needed to identify a city (e.g., different countries could have states and cities with the same simple names).

(a) Schematize this UoD, including derivation. Note that external uniqueness constraints may be declared across any join path. If any subset constraints apply (possibly from a join), specify them. Ensure that the fact types are all elementary.
(b) Rmap your solution.
(c) Discuss whether a denormalized relational schema may be more appropriate.

14.8 Reengineering

While conceptual modeling provides the best approach to developing new applications, it can also be used to remodel existing systems to better meet the requirements. There are three main reasons for replacing or revising (perhaps drastically) an existing information system.

First, the current system might provide an *incorrect* model of the UoD, either by being *inaccurate* (wrong information) or by being *incomplete* (missing information). Wrong or missing fact types, constraints, and derivation rules might arise because of bad modeling in the first place. For example, the schema might assert that planets orbit moons, allow people to have more than one birth date, or derive ages from a simple formula that ignores the extra day in leap years. Such errors can also arise because the model has not kept pace with changes in the UoD. For example, a system for recording academic results based on a seven-point numeric rating scale becomes outdated if the rating scheme is replaced by a five-point letter grade.

Second, the existing system might be *inefficient*. It might be too slow (e.g., too many table joins needed in queries), too large (e.g., too many nulls gobbling up space), or difficult to evolve (e.g., outdated rules hardwired into program code rather than modeled as data in tables). Perhaps the hardware platform itself is outmoded and needs to be replaced, or the external interface is awkward for users.

Finally, the existing system might be *unclear*. This also makes it hard to maintain or evolve. Many legacy systems in place today were designed at the logical, or even physical, level and have no conceptual schema formulated for them. They may have been developed completely in 3GLs, with no underlying design method, have little or poor documentation, and be riddled with misleading or cryptic identifiers.

Often a combination of the three reasons leads to an information system that needs to be *reengineered*, or restructured, in order for its owner (e.g., a company) to remain competitive in today's marketplace. The company might even reengineer the very way it does business. Although the need for restructuring has sometimes been used as an excuse for retrenching, a creative response to this need can lead to major enhancements in the quality and efficiency of the business and its supporting information systems. Here we consider only the problem of reengineering a database system. This process may be divided into four stages, as shown in Figure 14.77.

Figure 14.77 Four steps in reengineering a database.

The first stage, *conceptualization*, or *reverse engineering*, involves developing an initial conceptual schema of the application from the original database. This can be done simply by applying the CSDP to the database, treating it as a set of output reports. Most of the verbalization stage is usually straightforward since tables provide the easiest kind of output report to interpret. Usually, at least an incomplete logical schema for the database is also available, and constraints in this schema (e.g., primary and foreign key constraints) can be used to help derive some of the constraints in a fairly automatic fashion.

If the database is not relational, some of the intertable connections may need to be verbalized as fact types. Often, additional constraints that have not been specified will be identified with the assistance of the domain expert.

The second stage, *transformation*, involves modifying the original conceptual schema to cater for changes in the UoD not modeled in the original database, and/or optimizing the conceptual schema as discussed in the previous sections.

The third stage, *mapping*, or *forward engineering*, takes us down to the logical schema. If a relational database system is to be used, we may use the Rmap procedure for this. The fourth stage, *data migration*, copies the facts stored in the original database to the new database. This involves conversion as well as copying since some of the fact types involved will now be grouped into different table structures (recall the patient data example in Section 14.1). If the final two stages update the original database (rather than simply generating a new database) the whole reengineering process is sometimes called round-trip engineering or simply "round-tripping".

Much of the reengineering process can be automated, especially if a very detailed database schema is provided at stage 1. However, human interaction is also required to ensure a complete and elegant result. While our discussion is limited to the logical and conceptual levels, reengineering an application involves work at the external level (e.g., revising the user interface) and physical level (e.g., retuning) as well.

The four basic reengineering stages are now illustrated with a worked example. As the main input to the process, Figure 14.78 shows a relational schema as well as two rows of data for each table (shown as tuples on the right to save space).

Relational schema (some constraints missing):		**Sample data** (not significant)
Committee	{Prog,Org} (cteeCode, chairPerson, [budget]) ≤ 2	('Prog', 'Adams, Prof. A.B.',?) ('Org', 'Bloggs, Dr F.',10000)
Person	(personName, affiliation, [email])	('Adams, Prof. A.B.','MIT',?) ('Pi, Dr Q.T.', 'UQ','pi@uq.au')
Rated	{1..10} (referee, paperNr, rating)	('Adams, Prof. A.B.',43,8) ('Pi, Dr Q.T.',5,4)
Referees	(personName, paperNr)	('Adams, Prof. A.B.',43) ('Knot, Prof. I.M.',43)
Authored	(personName, paperNr, paperTitle)	('Sea, Ms A.B.',5,'EER models') ('Pi, Dr Q.T.',43,'ORM dialects')
Presents	(author, paperNr)	('Pi, Dr Q.T.', 43) ('Knot, Dr I.M.',61)
Paper	{undec,accept,reject} (paperNr, paperTitle, status)	(5,'EER models','undec') (43,'ORM dialects','accept')
AcceptedPaper	(paperNr, [pageLength])	(43,14) (75,?)
AcPaperDiags	{tbl, fig} (paperNr, diagramKind, qty) 2	(43,'tbl',3) (43,'fig',5)
PaperSlot	{A,B} (slotNr, stream, hr, [bldgNr, roomNr], [paperNr])	(1,'A',9,69,'110',43) (2,'B',9,50,'110',?)
Room	{lab,lec,office} (bldgNr, roomNr, roomType, area)	(69,'110','lec',100) (50,'110','lab',120)
LabOrLecRm	(bldgNr, roomNr, [nrPCs], [nrSeats])	(69,'110',?,350) (50,'110',40,?)

Figure 14.78 The original database schema and some sample data.

Suppose this design is used to record details about a 1-day computer conference, but its performance is poor mainly because of the number of table joins required for common queries.

Moreover, application developers find parts of the schema awkward or even unsafe. For example, the budget column in the Committee table applies to the committee as a whole, not to a particular chairperson of that committee (each committee has at most two chairpersons). It is suspected that this is not the only unnormalized feature of the tables.

Because of your expertise in conceptual modeling, you are hired to provide a conceptual model of the application, so that a clear and complete picture of the business domain is available. Moreover, you are asked to provide a new relational design that is both normalized and efficient, if this is possible.

In practice, reengineering usually requires dialog with the domain expert to clarify the UoD, especially to identify missing constraints. To help you try the reverse engineering phase yourself, uniqueness, value, exclusion, and frequency constraints are depicted. However, other constraints are omitted (e.g., subset constraints, and non-implied functional dependencies due to denormalization). Some missing constraints are "obvious", and others may be identified from the brief description that follows.

As well, the sample rows of data enable you to apply CSDP step 1. These rows do not provide a significant, or even a legal, population (e.g., if the database were composed of only these data, various subset constraints between the tables would be violated).

Abbreviated codes have the following meanings: Prog = Programming; Org = Organizing; undec = undecided; tbl = table; fig = figure; lec = lecture; lab = laboratory. Authors submit papers that are then sent to referees to be rated. The method for determining which papers are accepted is not modeled here. Various statistics (number of pages, tables, and figures) may be kept about accepted papers, partly to help with publication of the conference proceedings. If statistics are kept for a paper, all three numbers must be recorded.

An accepted paper may be presented by one or more of its authors at the conference. Which authors present the paper might be unknown until some time after the paper is accepted. The conference program includes slots to which accepted papers are eventually allocated. Each slot is of 1 hour duration. Time is measured in military hours (e.g., 14 = 2 p.m.). To allow more papers on the same day, two streams (A and B) may be run in parallel (so two paper slots may have the same time).

A room directory is used to help assign slots to rooms; room area is in square meters. Only lecture rooms or laboratories may be used for paper presentations. The number of seats is recorded just for lecture rooms, and the number of PCs is recorded just for laboratories.

Before peeking at the solution provided, try to provide a conceptual schema for this application. Include all constraints (including those missing from the original relational schema). In performing this conceptualization, it will become apparent that some of the tables have redundancy problems. Which tables are these?

By now you should have completed your own reverse engineering. Compare your solution with the one provided in Figure 14.79. Give yourself a pat on the back if you got all the constraints. It should be clear that the table grouping differs from what Rmap would generate if applied to the conceptual schema. To begin with, the two Committee fact types are wrongly grouped into the same table, even though one of them is nonfunctional.

As a result, the original Committee table is open to redundancy problems with facts of the type Committee has budget of MoneyAmt. Similarly, the Authored table exhibits redundancy problems by wrongly including facts of the type Paper has PaperTitle. So the original designer denormalized the relational schema in an uncontrolled way.

Although the ORM schema looks complicated, the conceptualization actually clarified the UoD. For example, the semantics of the PaperSlot table are easier to understand on the ORM schema. Now that the detailed semantics are displayed graphically, we are in a better position to explore optimization possibilities in a controlled manner.

As the second phase of the reengineering, use the conceptual optimization procedure discussed in the previous section to transform the conceptual schema into one that one that Rmaps to a more efficient relational schema.

Compare your solution with that provided in Figure 14.80. The specialization of the chairperson and diagram predicates into functional binaries is straightforward (although care is needed to preserve the constraints). In nesting the referee and author predicates, we could instead have united these exclusive but compatible predicates. Although this alternative move would lead to one less table, nesting these exclusive predicates would detract from the readability and lead to awkward constraint patterns.

Figure 14.79 The conceptual schema resulting from reverse engineering.

Indeed, even the nesting in Figure 14.80 leads to awkward verbalization. The original flattening into four predicates seems more natural than the two nestings. Ideally a CASE tool would allow the modeler to choose the flattened display for clarity, with the option of nesting in a hidden, preprocessing stage to the standard mapping.

The next stage is to map the optimized conceptual schema to a new relational schema. As an exercise, try this yourself before looking at the solution in Figure 14.81. Notice that the final relational schema contains just 7 tables (compared with the 12 original tables), is fully normalized, and captures all the original constraints, as well as those that were missing from the original.

Finally the data migration is performed. The main aspect of this conversion involves defining the new relation types in terms of the original ones. Once this is done, the data migration can be performed automatically. Figure 14.81 displays one row of converted data for each of the new tables.

Figure 14.80 The optimized conceptual schema.

[1] **exists only if** paperNr **in** (Paper.paperNr **where** status = 'accept')
[2] **exists only if** status = 'accept'
[3] **exists iff** nrPages **exists**
[4] **only where** status = 'accept'
[5] **only where** roomType **in** {'lab','lec'}
[6] **exists iff** roomType = 'lab'
[7] **exists iff** roomType = 'lec'

Figure 14.81 The final relational schema.

Exercise 14.8

1. The following relational schema was designed to record details about academics and subjects, but its performance is poor, mainly because many joins are needed for common queries. Also, developers find the Academic table and its qualified constraint to be awkward to think about. This constraint says that on each row of the Academic table there is exactly one nonnull value in the last three columns.

   The column "dept_where_lec" means the department (if any) where that academic is a lecturer, "dept_where_sen_lec" means the department (if any) where that academic is a senior lecturer, and "dept_where_prof" means the department (if any) where that academic is a professor.

   Here is one sample row from each table ("?" denotes a null):

   Subject: ('CS115', 8)
   Teaches: (30572, 'CS115')
   Academic: (30572, ?, 'Computer Science', ?)
   AwardedBy: (30572, 'PhD', 'UQ')
   AwardedIn: (30572, 'PhD', 1990)

Reengineer the model to improve its performance and clarity, using the following steps.

(a) Reverse engineer the relational schema to an ORM schema. Include all constraints.
(b) Optimize the conceptual schema to prepare it for relational mapping.
(c) Forward engineer your new conceptual schema by mapping it to a relational schema. Include all constraints.
(d) Populate the new relational schema with the sample data provided.

14.9 Data Migration and Query Transformation

Data migration, the fourth phase of reengineering, can sometimes be challenging. When the original database is populated with massive amounts of data, it is typically impractical to convert and renter the data manually into the new database schema. While most DBMSs provide tools to assist with data migration, these tools usually handle only simple data transformations, so you need to write your own transforms to deal with more complex cases. Such data transformations are normally written in SQL, although XSLT could be used if the data is available in XML form.

Once the data migration is completed, any stored queries on the old database need to be recoded to perform equivalent queries on the new databases. This section illustrates the tasks of data migration and query transformation using a small reengineering example. The exercise at the end of this section provides a larger example for you to try out yourself.

Figure 14.82(a) shows a fragment of a relational schema about martial arts clubs. In martial arts (e.g., judo and karatedo) there are exactly two kinds of rank: dan (black belt grade) and kyu (below black belt). The ClubNrRankedPlayers table stores facts about how many dan and kyu graded players belong to certain clubs. For example, the tuples (‘KDK’, dan, 300) and (‘KDK’, kyu, 250) would mean that the club with the clubcode ‘KDK’ has 300 dan grades and 250 kyu grades.

Suppose that the original database has a stored query to list the code and name of each club, together with the number of black belt players (if known). This query may be formulated in SQL as query Q1.

Figure 14.82 (a) Original relational schema and (b) reengineered relational schema.

```
Q1: select Club.clubCode, clubName, nrPlayers as nrDanGrades
    from   Club left outer join ClubNrRankedPlayers
      on Club.clubCode = ClubNrRankedPlayers.clubCode
         and ClubNrRankedPlayers.rankType = 'dan'
```

Now suppose that in reengineering the original database, these two table schemes are replaced by the equivalent, single table scheme in Figure 14.82(b). This optimization is based on the predicate specialization transform discussed in Section 14.2.

Suppose the original schema is called MartialArts1 and the new schema is called MartialArts2. We may now migrate the data from the original two tables into the new table using the following SQL command.

```
insert into MartialArts2.Club
select Club.clubCode, clubName, cityname, ClubDans.nrPlayers, ClubKyus.nrPlayers
from MartialArts1.Club left outer join MartialArts1.ClubNrRankedPlayers as ClubDans
                 on Club.clubCode = ClubDans.clubCode
                    and ClubDans.rankType = 'dan'
        left outer join MartialArts1.ClubNrRankedPlayers as ClubKyus
                 on Club.clubCode = ClubKyus.clubCode
                    and ClubKyus.rankType = 'kyu'
```

The original query Q1 may now be reformulated to work with the new database, as shown in Q2.

```
Q2: select clubCode, clubName, nrDanGrades
    from   Club
```

Even with this simple example, you can see that writing the code to do the data migration can require a little thought. In practice, it can get much more difficult than this. The following exercise, which covers most aspects of the data reengineering process, includes several more challenging data migration tasks. The solutions to this exercise (and all other odd numbered questions) are available online.

Exercise 14.9

1. The SQL Server diagram shown depicts the main aspects of a relational database schema CarSale1 used by a car retailer to record details about cars and sales. The developers do not fully understand the schema, and suspect that some aspects of the schema are incomplete or even unsafe. Also, the database has performance problems mainly because of the number of table joins required for common queries.

The tables were originally thrown together with little design discipline by someone who has now left the application team. You are asked to provide a new relational design that is both normalized and efficient.

Although the schema includes some primary key declarations, it is missing several constraints that need to be enforced. Moreover, it includes some non-implied functional dependencies due to denormalization. Some missing constraints are "obvious", and some others may be inferred from the brief, incomplete description that follows. In addition, some rows of data are provided for each table so you can apply ORM CSDP step 1. Although useful, the sample data are not fully significant (i.e., not all constraints may be deduced from them).

For this UoD, each car is identified by a vehicle identification number (vin) and has exactly one color (identified by a code). Car models are back wheel drive (bwd), front wheel drive (fwd), four wheel drive (4wd), or all wheel drive (awd). Each car color has a unique name, and optionally its RGB (Red, Green, Blue) composition may be recorded. At most two phone numbers are recorded for each customer. There is no restriction on what criteria may be used by magazines to rate car models, but each rating must be in the range 1..7 (the higher the rating the better). Whether a customer is an employee of the car retailer is not of interest. An SQL Server database with sample data for this question is available online.

(a) Which table schemes in the original schema are not fully normalized? Explain briefly what design flaws cause them to be denormalized.

(b) *Reverse engineer* the relational schema to an ORM conceptual schema. Do *not* concern yourself with efficiency or optimization at this stage. Your ORM schema, while being redundancy free with all fact types elementary, should in other respects directly reflect the design decisions made in the original relational schema. *Include all constraints* (graphical or textual) that apply to the application domain, including those constraints omitted from the original relational schema.

Formulate the following queries in SQL:

(c) For each car model that is reviewed, list its id and the magazines in which it is reviewed, as well as its economy rating(s) (if known).
(d) For each employee who has made a car sale and is either a novice or an expert in Spanish, list their employee number, name, and gender, as well as their sales total (i.e. the total amount they have earned for the company from car sales).
(e) For each car whose color has an equal measure of red and green, list its vehicle identification number, color code, car model id, and drive kind.

Criterion is now constrained to the values 'Safety', 'Economy', 'Performance', and 'Style'.

(f) Transform the ORM schema to an "optimized conceptual schema" that minimizes the number of tables obtained if the Rmap procedure is applied.
(g) Forward engineer your optimized ORM schema to a new relational schema CarSale2. Ensure that all tables and columns are named well. Annotate the relational schema diagram to display all constraints, using footnote numbers and text boxes where needed. Value constraints should appear in braces after relevant column name(s).
(h) The external uniqueness constraint that each RGB intensity combination applies to at most one Color cannot be specified as a unique clause in most DBMSs, but can be coded using a trigger. Write appropriate trigger code to enforce this constraint.
(i) Now write SQL code to perform the data migration needed to populate the CarSale2 schema.
(j) Using your new CarSale2 schema, reformulate the queries in (c), (d), and (e).

14.10 Summary

Conceptual schemas are *equivalent* iff they model the same UoD. Various *schema transformations* may be performed to reshape a conceptual schema into one that is either equivalent, or an acceptable alternative.

Predicate specialization fragments a predicate into two or more special cases, typically by *absorbing* an enumerated object type into it. For example, the association Person is of Gender {'M','F'} may be specialized into Person is male and Person is female by absorbing Gender.

The inverse transformation, *predicate generalization*, typically involves *extracting* an enumerated object type from a predicate. Predicate specialization/generalization (PSG) may also be performed by considering the n cases of a role with a frequency constraint of n or $\leq n$. For example, the association Person is a parent of Person with a frequency constraint of 2 on its second role may be specialized into Person is parent1 of Person and Person is parent2 of Person.

Nested object types may be interchanged with *coreferenced* object types, using the Nest/Coreference transformation (N/CR). Either representation may be replaced by *flattening* into one or more fact types. For example, consider the nesting: Enrollment(Student enrolled in Subject) results in Rating. If results are mandatory, this may be flattened into the ternary: Student for Subject scored Rating. If results are optional, the flattened approach requires an extra binary (Student enrolled in Subject) targeted by a pair-subset constraint from the ternary.

The *overlap algorithm* indicates how nesting may be performed when compatible subpredicates exist, depending on the constraint between them (subset, equality, exclusion, etc.). See Figure 14.35. Various other schema transformations exist. Sometimes a transformation converts a graphical constraint to a textual constraint (or vice versa).

Although nested and coreferenced approaches have the same relational map, many other cases of equivalent conceptual schemas result in different relational schemas when subjected to the standard Rmap procedure. The process of transforming a conceptual schema into another that results in a more efficient relational map is called *conceptual schema optimization*.

Many optimization moves aim to reduce the number of relational tables, typically by specializing nonfunctional predicates into functional ones that can be grouped into the same table as another functional fact type or by using nesting (or coreferencing) to unify compatible composite keys. This is particularly useful in reducing the number of table joins required for focused queries. Predicate specialization is sometimes used to enable comparisons between rows of the same table to be replaced by comparisons between columns on the same row, thus eliminating the need for a self-join to perform the query.

The Rmapping of a correctly designed conceptual schema automatically results in a safe, redundancy-free, relational schema where the number of tables has been reduced by grouping together functional fact types with the same key. In contrast to this deconceptualization approach, *normalization* provides a set of rules for achieving reasonable table designs by catering for dependencies between attributes.

Normalization by *synthesis* inputs a set of attributes and some basic dependencies, and groups the attributes into tables, with the aim of minimizing the number of tables required to satisfy these dependencies. Normalization by *decomposition* applies rules to split a poorly designed table into two or more tables in an effort to ensure that some basic dependencies are implied by the table key constraints.

This decomposition approach recognizes a number of *normal forms*. A table is in first normal form (*1NF*) if it has a fixed number of columns, all of whose values are atomic. A nonkey attribute is neither a key nor a part of a key. In a *2NF* table, every nonkey attribute is functionally dependent on a whole key. In a *3NF* table there is no FD between nonkey attributes.

An FD $X \rightarrow Y$ is trivial if X contains Y, and full if the FD requires all of X, not just part of it. An elementary FD is full and nontrivial. An elementary key has an elementary FD to some other attribute. In an *EKNF* (elementary key NF) table, each elementary FD (if any) comes from a whole key or ends inside an elementary key. In a *BCNF* (Boyce–Codd NF) table, each elementary FD begins at a whole key.

A *multivalued dependency* (MVD) $X \twoheadrightarrow Y$ exists if the set of Y values in the table depends only on X. An MVD $X \twoheadrightarrow Y$ is trivial if X includes Y, or X and Y together include all the attributes in the table. A *4NF* table is a BCNF table where all its nontrivial dependencies are FDs. So each nonfunctional MVD requires a separate table for itself. To properly apply this notion to a table scheme, we strengthen it to *semantic 4NF*, where a semantic MVD $X \twoheadrightarrow Y$ applies iff the set of Y values is dependent on X alone for all possible populations of the underlying fact types.

A table has a *join dependency* (JD) if it is equivalent to the join of some of its projections. If one of these projections is the table itself, the JD is trivial. A table is in 5NF if for each nontrivial JD, each projection includes a key of the original table.

These seven normal forms may be listed in increasing order of normality: 1NF, 2NF, 3NF, EKNF, BCNF, 4NF, and 5NF. Higher level forms satisfy all the lower forms. The 5NF form is often called "PJNF" (Project-Join NF), although some subtle differences exist between these notions.

If a fact type is elementary, all its FDs are the result of its UCs. If another FD exists, underlying it is another fact type targeted by a tuple-subset constraint from the original fact type, which then requires role removal (see Figure 14.72).

A table is in Domain-Key Normal Form (*DKNF*) if its only constraints are domain constraints and key constraints. A relational schema is in DKNF if all its tables are in DKNF and intertable constraints are either absent or simple (e.g., inclusion dependencies, which are typically foreign key constraints). In practice, more complex constraints often exist in the UoD being modeled, so DKNF is often inadequate for complete models. In general, normalization ignores various constraints that we have considered at both conceptual and relational levels.

Once conceptual optimization and mapping have been completed, the physical schema should be *tuned* (e.g., by adding indexes, working tables, or materialized views). Various choices are also made as to the best place(s) to enforce constraints (e.g., at the database and/or in screen forms). If performance is still poor, it may be necessary to *denormalize* some of the base tables, via *controlled redundancy* to reduce the need for joins.

If an existing database application is unsatisfactory, it may be necessary to *re-engineer* it. This remodeling may involve four main steps: conceptualize (or *reverse engineer*) the existing database schema to a conceptual schema; *optimize* this by transformation; *map* (or *forward engineer*) it to the improved database schema; and migrate the original data in order to populate the new database. Once the *data migration* is complete, *query transformation* is often needed to transform stored queries based on the old schema into equivalent queries on the new schema.

Chapter Notes

Schema equivalence has been studied within the relational model (e.g., Kobayashi 1986, 1990), the ER model (e.g., D'Atri and Sacca 1984), EER models (e.g., Batini et al. 1992) and UML (e.g., Blaha and Premerlani 1998). To keep our treatment intuitive, some technical issues have been glossed over. More formal treatments of schema transformation in ORM may be found in Halpin (1989b, 1991b) and Halpin and Proper (1995a). For formal proofs of some conceptual schema equivalences, see Halpin (1989b). Further reduction transformations and additional ORM constraints (e.g., unique-where-true constraints) are discussed by Halpin, Carver, and Owen (2007).

Most work on schema optimization focuses on subconceptual levels. Date (2000) provides clear discussion of various optimization aspects at the relational level. For an approach to schema optimization that evaluates fitness functions on randomly generated, equivalent internal schemas, see van Bommel and van der Weide (1992).

A simple introduction to normalization is provided by Kent (1983). Classic papers include Codd (1970), Fagin (1977, 1979, 1981), Rissanen (1977), and Zaniolo (1982). In Fagin's original DKNF paper (Fagin 1981), the term "key" is used in the sense of "superkey", so a UC may be implied by a smaller UC within it. Fagin uses the terms "relation schema" and "database schema" for what we call a "table scheme" and "relational schema", respectively.

Date and Fagin (1992) discuss simple, sufficient conditions for some of the higher normal forms. The extensions to semantic dependencies and semantic 4NF are due to Carver and Halpin (2008). A general study of join dependencies (including functional and multivalued dependencies) is provided by Aho et al. (1979). For an early paper on deductive normal forms see Thalheim (1984). For a thorough analysis of normal forms see Vincent (1994). An in-depth, formal treatment of normal forms and their relationship to ER modeling is given by Thalheim (1994). Chapters 11 and 12 of Date (2000) cover the main aspects of normalization and provide extensive references to the relevant literature.

15 Process and State Modeling

15.1 Modeling Dynamic Behavior

15.2 Processes and Workflow

15.3 State Models

15.4 Foundations for Process Theory

15.5 Modeling Information Dynamics in UML
- Activity Diagrams
- State Machine Diagrams
- Executable UML

15.6 Business Process Standards Initiatives
- Business Process Modeling Notation
- Business Process Definition Metamodel
- Business Process Execution Language

15.7 Standard Process Patterns

15.8 Summary

15.1 Modeling Dynamic Behavior

In this book, we have been mostly concerned with understanding information structures. This has very practical applications: if we want to build a business information system, it is crucial that we have an accurate definition of the information that we intend to use. We have seen how information can be modeled and how we can use the knowledge gained through modeling as the basis for storing and manipulating the information. Of course, modeling does not in itself determine what kind(s) of information fulfills a business need. For example, should a customer information system include each customer's shoe size? For a shoe retailer this is likely to be important information, but it would probably be irrelevant for an auto insurance company. Whatever the chosen scope, we've seen how it is possible to produce very precise definitions of the required business information.

Information models make minimal assumptions about how the information will be used. This is quite deliberate: a business may find that information originally intended for one purpose turns out to be useful for other purposes. Some implementation choices may be made below the conceptual level, such as denormalizing a database schema to improve query performance. However, it's generally desirable to have an information model that reflects the intrinsic structure of the information itself and is neutral with respect to information usage.

A side effect of this is that an information model does not provide a complete specification of the information dynamics: the temporal relationships among information creation, modification, and deletion. In order to represent these aspects fully we need some additional specification.

We can illustrate this point with an example. Figure 15.1 shows a fragment of an information model. Each of Data1, Data2, and Data3 could actually represent a complex set of data, but that's not important for our purposes here. Let us assume that our business has three activities, CollectData1, CollectData2 and CollectData3, which result in the creation of Data1, Data2, and Data3, respectively. Each of these activities takes the same amount of time and has the same cost in terms of resources required. We will further assume that our business is only interested in Data3 if both Data1 and Data2 meet certain criteria, otherwise Data3 is not needed. On average, 90% of cases meet the Data1 criterion, and 10% of cases meet the Data2 criterion.

Figure 15.1 A simple information structure.

Now let us consider the question of how the information represented by Data1, Data2, and Data3 could evolve. Given the business scenario, it would not seem very sensible to collect Data3 before trying to collect Data1 and Data2 so we can immediately eliminate some of the possible orderings of CollectData1, CollectData2, and CollectData3. The remaining options are as follows.

(a) CollectData1 followed by CollectData2, then CollectData3 if needed.
(b) CollectData2 followed by CollectData1, then CollectData3 if needed.
(c) CollectData1 and CollectData2 simultaneously, then CollectData3 if needed.
(d) CollectData1, CollectData2, and CollectData3 in parallel, accepting that we will be carrying out some additional unnecessary work.

We can compare these alternatives by considering an average set of 100 cases. For option (a) we carry out 100 CollectData1 operations. On average, 90% of the cases meet the Data1 criterion, so we then carry out 90 CollectData2 operations. On average, 10% of the cases meet the Data2 criterion so we finally carry out 9 CollectData3 operations. The total cost will be 100 + 90 + 9 = 199 units.

For option (b) we carry out 100 CollectData2 operations. On average, 10% of the cases meet the Data2 criterion, so we then carry out 10 CollectData1 operations. On average, 10% of the cases meet the Data1 criterion so we finally carry out 9 CollectData3 operations. The total cost will be 100 + 10 + 9 = 119 units.

For option (c) we carry out 100 CollectData1 operations and 100 CollectData2 operations. On average, 9% of the cases meet both the Data1 and the Data2 criterion so we finally carry out 9 CollectData3 operations. The total cost will be 100 + 100 + 9 = 209 units.

Finally, for option (d) we carry out 100 CollectData1 operations, 100 CollectData2 operations, and 100 CollectData3 operations. We end up with the same 9 cases, but this time at a cost of 300 units. Figure 15.2 summarizes these results.

From this analysis we can see that option (b) is likely to have the lowest cost. However, our business may have decided that speed of service is more important than lowest cost, and may prefer option (d), despite the higher cost. In fact, the company might choose to offer all of options (b), (c), and (d), passing the choice between cost and speed on to the customer.

Even examples as simple as this can point to some general conclusions. First, information can potentially evolve in many different ways, and the differences might be very important to a business—in the example just given, different options resulted in different costs and different processing times. Second, the relationship between information *structure* and information *usage* is one to many: a single information structure has the potential to be used in many ways, but a particular sequence of activities will always relate to one particular information structure. Third, we cannot define business activities (as in Figure 15.2) by considering the information structure alone (as in Figure 15.1). Although a model of information structure is necessary, it does not itself provide sufficient information to specify an information system fully: we also need to know something about how the information will be used.

Figure 15.2 Alternative processing strategies.

This separation of concerns turns out to be a useful aspect of information modeling. Businesses frequently review and revise their activities in the light of perceived opportunities or threats. Having some degree of orthogonality between a "structural" view and a "behavioral" view minimizes the ripple effect of changes. For example, a business could switch between "minimum cost" and "minimum time" strategies without necessarily requiring any change to the information model.

Although information structure alone does not fully specify activity sequencing, it may place some constraints on the order in which information can be created. This is easy to see if we look at the information structure in the form of a relational schema. Figure 15.3 shows an example.

Figure 15.3 A relational schema.

For this schema, we cannot enter information about a Fitting unless we have previously recorded the Employee and Item involved. Similarly, a Test cannot be entered unless the Item has already been recorded. These constraints are recognizable as foreign key references between the tables. However, there is a weaker ordering constraint for the tester information: this column is optional, and so we could create a Test record leaving the value for tester as null. The referenced Employee only needs to exist at the time we add the tester information, which could be some time after creating the Test record. Corresponding constraints apply to the removal of information: for instance, we would probably not want to delete an Employee while we still have Fitting records relating to that Employee. For the tester column, the foreign key reference would probably be specified as **on delete set null**, which would allow us to delete an Employee row that was still referenced in a Test, setting any existing tester references to the Employee table to null.

Even though constraints of this kind do not fully determine the sequence in which we can create, update or delete information, the constraints must still be observed in whatever modeling approach we adopt to define information behavior. This chapter concentrates on two approaches used commonly to model behavioral (sometimes called *dynamic*) aspects: process modeling and state modeling.

15.2 Processes and Workflow

It has always been important for an enterprise to manage its business processes. Process change, whether dramatic or subtle, is a common mechanism employed to keep a business efficient and competitive. Any medium to large business is likely to have a comprehensive set of processes that define how it intends to conduct its activities. The definition is needed by companies aiming to achieve quality accreditation to a standard such as the ISO 9000 series. Among other things, these quality standards require an organization to define its processes and to maintain adequate records of their execution.

Having a defined set of business processes does not necessarily imply any commitment to automation. Until the second half of the 20th century, the opportunities for process automation were limited by the available technology, and manual processes were commonplace. Today, it's more usual to have some degree of automation support. We can broadly distinguish several levels of automation, as shown in Table 15.1.

Table 15.1 Levels of process automation.

Level	*Characteristics*
Level 0	Process definition, execution, and record keeping are all carried out manually.
Level 1	As Level 0, but with records of process execution maintained in a computer-based information system.
Level 2	As Level 1, but with process design supported by computer-based tools.
Level 3	As Level 2, plus the automation of at least some of the process activities.

An organization may have different processes at different levels, but the general trend is for processes to migrate toward Level 3. At the extreme end of Level 3, we have processes in which all of the activities are automated, a position that is facilitated by the growing availability of technology to support Service-Oriented Architecture.

We can also classify processes based on the structure and the repeatability of the activities involved. Some processes are *ad hoc:* the activities involved are not predefined, and so, obviously, their order cannot be predefined either. In place of a predefined structure, ad hoc processes are usually subject to a set of rules that define what kinds of activities are permitted, how activities can be started or ended, and so on.

Most of the current interest in process modeling relates to more structured arrangements of activities, sometimes termed *framed* processes. *Individually* framed processes are well defined, but the process definition is executed only a few times—perhaps just once. An example is a project plan, which usually defines a detailed set of activities and their sequencing, but is unlikely to be re-used in any other project. *Loosely* framed processes, sometimes also known as "administrative processes", have defined activities and coordination rules, but with flexibility to vary the activities and their sequencing within some set limits. *Tightly* framed processes, sometimes also known as "production processes", have highly predictable activities and stable rules for coordination that are followed consistently.

An important characteristic of loosely framed and tightly framed processes is that they are generally repeatable. This makes investment in their automation an attractive proposition—the cost of development can be justified by the potential savings made through better control over process execution. In the remainder of this chapter, we will assume that we are dealing with processes of this kind.

A term that commonly appears in relation to process modeling is *workflow*. The original focus of workflow was the mechanization of business activities that, historically, involved passing paper documents between in trays and out trays. Consequently, workflow, and its embodiment in workflow systems, placed a great deal of emphasis on enactment. More recently it has been realized that workflow is part of a set of larger concerns that relate to the systematic handling of business information. In many contexts the term workflow can be used interchangeably with business process modeling, but workflow generally implies a slightly lower level viewpoint dealing mainly with operational and technical issues.

In contrast, business process modeling places more emphasis on the definition of the process, with the assumption that the process design can ultimately be mapped to some realization. These concerns recur throughout a process lifecycle: process definition and design naturally precede its realization in a technical form and its subsequent operation, but the relation between these is cyclic as business processes are reviewed and revised in the light of changing business circumstances.

Figure 15.4 summarizes the general picture, but it's worth emphasizing that the overlap is fuzzy and that the definitions of these terms vary a great deal from one person or organization to another.

Figure 15.4 Business Process Modeling and Workflow.

Modeling a process usually involves breaking it down into a set of activities that, taken together, are intended to produce a specific result of value to the business. One way of looking at process structure is as a hierarchical assembly of activities—a tree—as shown in Figure 15.5. Loops are not permitted: a process element (at any level) cannot contain itself, but the same activity could appear at several places in the tree.

The root node of the activity tree represents the process itself. The leaf nodes represent the smallest units of activity (with no lower level breakdown into finer grained activities), which in most methodologies and tools have some special name, such as "activity", "task", "elementary business process", and so on. The intermediate nodes may also have some special name, such as "block" or "subprocess". In this chapter, we adopt the terms "process", "subprocess", and "activity", respectively, for the top, intermediate, and bottom levels of the process structure. In some methodologies and tools, actual work is assumed to happen inside the top and intermediate nodes; in others, these nodes exist only as a mechanism for grouping the leaf nodes, which are assumed to contain all of the actual work, into convenient clusters.

Clearly, we could have both a model of business activity and a model of information structure relating to the same area of a business, and therefore it should be possible to establish some kind of relationship between them. In fact, most process modeling approaches focus mainly on flow-of-control issues and have surprisingly little to say about the information used by, produced by, or modified by the process. We look at some examples later in this chapter.

Figure 15.5 An activity hierarchy.

Figure 15.6 Main process-definition structure.

In order to establish a connection between process activities and the information they use, produce, or consume, we need to examine process structure in a little more detail. In particular, we need to establish how the "activity" level of a process structure can be aligned with identifiable elements in an information model. We will return to the detailed sequencing and control of the activities later, but our main focus for now will be on the *structure* of a process.

A model fragment containing the main structure of a process definition is shown in Figure 15.6. Processes can contain subprocesses to any number of levels, but loops are not allowed. We use the special term "activity" for the lowest level of the hierarchy and assume that a process definition (at any level) will contain subprocesses, activities, or both. The same type of activity could appear several times in a given process. For example, an activity of the type Charge Customer's Credit Card could appear at various stages of the same process and so could be considered as directly contained in several parent nodes. Similar considerations apply to subprocess definitions.

Each process, subprocess, and activity can have multiple instances. Figure 15.7 shows the relationship between the definitions of process elements and their instances. Each instance is an execution of the corresponding definition. A specific activity instance will always be running in the context of a specific process instance, even though the process instance could contain several instances of the same activity. Hence the relationship between ProcessInstance and ActivityInstance is one to many instead of the many to many relationship between ProcessDefinition and ActivityDefinition. The current behavior of an activity can be indicated by a status value. The corresponding definitions do not have this kind of dynamic behavior: they simply exist.

Figure 15.7 Definitions and instances of process elements.

Figure 15.8 Definitions and instances of resources.

Activities require resources in order to carry out useful work. The activity definition will normally identify the type of resource required. Instances of relevant resource types are associated with running instances of activities. One resource instance may meet the needs of more than one resource type. Some resources, such as electrical power, may be consumed by an activity. Other resources may be simply utilized for a while and then freed for use in other activities.

Figure 15.8 shows how resources relate to the process structure. For simplicity, Figure 15.8 shows resources as associated with activities. As discussed earlier, if process and subprocess definitions specify actual work, then resources definitions could also be associated with those (and, similarly, resource instances could be associated with the process and subprocess instances). "Role" and "OrgUnit" are examples of resource definitions: there are many other possibilities. A particular resource instance may satisfy more than one resource definition: for example, an individual might be both a "Level 3 Manager" and a "Mortgage Specialist".

It's worth noting that resource allocation, just as information structure, tends to be treated lightly by modeling approaches centered on flow control. The relationship of process elements to organizational units also tends to have weak support, with the allocation of activities into "swimlanes" (which are discussed again later in this chapter) being the only mechanism provided in some approaches.

We won't dwell any further on resources and organizational relationships because we are going to be mainly concerned with the information aspects of the process. The activity definitions will specify the work to be carried out, but for our purposes it's more important to focus on the data modified or accessed by the process. If the information structure (the business objects and their relationships) has been defined in ORM, this would correspond to references to relevant object types and fact types, as shown in Figure 15.9. Instances of these object types and fact types are used in executing instances of the corresponding activities. An ObjectTypeInstance (or a FactTypeInstance) is, of course, simply an Object (or a Fact), respectively: "instance" has been retained in the terms to emphasize the difference between the two sides of the model. As discussed previously, object types and fact types could also be related to processes and subprocesses if these contain actual work definitions.

Figure 15.9 Fact types and object types and their instances.

At the most primitive level, information manipulation in ORM can be reduced to adding or deleting instances of facts. At a slightly higher level we may wish to specify data usage in terms of whether the information is being Created, Read, Updated, or Deleted in the context of a particular activity. Figure 15.10 shows the possible ways of using a fact type (or its instance) in the context of a particular activity, where "C", "R", "U", and "D" stand for Create, Read, Update, and Delete, respectively. Again, these notions could be extended to processes and subprocesses if necessary.

We may also wish to specify that a particular activity should be explicitly denied usage of a particular fact type because of security or other similar considerations. The unary isDenied fact type allows this to be specified.

The hierarchical structure of the processes can be utilized to define information scoping. This is not illustrated here for reasons of space, but we can imagine specifying usage of a fact type at a subprocess level that could "trickle down" to its successive lower level process elements. This would avoid the need to specify data scoping only at the activity level. Mechanisms similar to the isDenied fact type could also come into play to prevent a lower level element from inheriting its parent scope where necessary.

Figure 15.10 Data usage from a fact-based perspective.

Figure 15.11 A simple process model.

In order to illustrate some of these points we can look at an example process model. Figure 15.11 illustrates a simple complaints process. The notation used is BPMN, which is discussed in more detail later. The light circle indicates a start event; the heavy circle indicates an end event. The process instance is active between these two events. The diamond shapes are "gateways", used in BPMN to indicate forking or joining of paths of activity. Gateways can be marked to indicate different fork or join semantics. The "+" and "X" symbols are used to identify AND and EXCLUSIVE-OR semantics for their respective gateways. The process proceeds as follows.

1. The initial complaint is received and recorded.
2. Contact is made with both the client who made the complaint and the department against which the complaint is directed to obtain further information. The activities ContactClient and ContactDepartment conceptually proceed in parallel because of the AND-split. The corresponding AND-join ensures that both activities complete before the next activity is started.
3. Based on the information now available, an assessment is made as to whether the client should be compensated or sent a polite letter rejecting the complaint. We can speculate that some business rules would probably come into play at this point.
4. Either the appropriate compensation is made or a letter rejecting the complaint is sent to the customer. The EXCLUSIVE-OR split ensures that only one of these is selected. The corresponding join waits until the selected activity completes.
5. Details of the case are filed in case of any future query, and the process terminates.

To provide a concrete example of the relationship between process and information, we'll look at the SendStandardLetter activity; a corresponding picture could be drawn for any of the other activities. We will assume that appropriate resources and manual services are available to the activity, such as printing, envelope stuffing, and franking, and focus on the information usage. Figure 15.12 shows a fragment of an information model, including the types of facts that might be relevant for the SendStandardLetter activity.

Figure 15.12 An ORM model fragment relating to the SendStandardLetter activity.

We are assuming here that the letter consists mainly of "boilerplate" text with the client information, the case reference, and dates inserted at the appropriate points. In order to identify the relevant facts, the activity requires two primary objects to be identified: a Case and a LetterType. From the Case we can uniquely identify the Client and hence the Client's Title, Name, and Address. From the LetterType we can identify the appropriate text to form the body of the letter. The current date is assumed to be available from the activity's environment.

The data usage for this activity is summarized in Figure 15.13, which shows the fact types relevant to this activity in a tabular form. The "primary object types" allow the specific facts required for an instance of this activity to be identified. For completeness, the date of the case filing is shown here as not used—it would actually be created by the following activity. In practice it would be simply omitted from the list of fact types for this activity. The same would be true for other fact types that are not relevant to this activity.

SendStandardLetter	
Primary Object Types in this context: Case, LetterType Environmental Data: the current date	
Fact Type	Data Usage
Case was filed on Date	-
Case was recorded on Date	R
Case refers to Client	R
Client has Title	R
Client has Name	R
Client has Address	R
Client was sent LetterType on Date	C
LetterType has LetterText	R

Figure 15.13 Fact type usage for the SendStandardLetter activity.

Most of the fact types provide information that is used but not modified in the activity ("R" in the table). If the activity included an opportunity to modify any of the facts ("U": possible) or delete any of the facts ("D": unlikely), then this could be shown against the relevant fact types. The only fact created ("C") in an instance of this activity is Client was sent Lettertype on Date.

Note that we deliberately avoid specifying how an instance of the activity would access any of the information (object passing, database lookup, etc.). These can be left as implementation decisions by a developer or by an automated tool. However, we do need to provide sufficient information to be able to tie process models and information models together.

15.3 State Models

The term "state" is used in several different ways in modeling. One usage relates to the changes made to information in a particular domain. An information processor can add or delete facts or modify existing facts in a fact store. After any of these changes, we could say that the fact store is in a different *state* than previously. In this sense, the number of possible states for an information model (or its equivalent realization) would be too large to count. Individual states of this sort are not identified by name or any other designation, and so the term is usually used in a before-and-after sense in relation to some update.

A different usage of the term state relates not to changes in *information* but to changes in *behavior*. As a simple example, let's say that we have a device that counts occurrences of a specific event. When a certain number has been reached it takes some action and then waits to be reset. An alarm clock is a practical device of this kind. While the device is counting, its information state is continually changing as the count is updated. However, its behavior remains consistent until the limit value is reached. From the information perspective, the device would have a large number of states (e.g., one for each count value), but from the behavioral perspective, it can only be in one of three states.

1. Counting events and checking the total against the predefined limit
2. Carrying out defined actions
3. Waiting for the user reset signal

In the early days of electronic computers, it was recognized that behavioral state descriptions provided a convenient way to define and build automata with the capability to react in seemingly complex ways to changes in their environment, and these reactive systems became know by the generic term *state machine*. Two alternative formulations appeared in the 1950s, named (after their inventors) Moore machines and Mealy machines. These are arranged slightly differently, but have similar capabilities—in fact a Moore machine can be transformed into an equivalent Mealy machine, and vice versa. Figure 15.14 gives a high level view of the two approaches.

Figure 15.14 (a) Moore machine. (b) Mealy machine.

Both approaches store information relating to the current state of the machine. The combination of the current state and the input to the machine determines whether a state change occurs. The main difference between these two approaches is that the outputs of a Moore machine are related to the machine being *in a particular state*, whereas in a Mealy machine, the output is determined by the input in the context of that state. For a Mealy machine, changes to the output therefore tend to be associated with *changes in state*.

We can illustrate some of the differences between the two using a bubblegum vending machine as an example. The machine sells only one type of item, which costs 15 cents (15¢). The machine takes coins of value 5¢ (a nickel) or 10¢ (a dime), but gives no change if excess money is inserted. The machine has a reset button: if this is pushed before vending, money already inserted is refunded and the machine returns to its initial condition (state 0). Once sufficient money has been inserted, the machine automatically delivers the item and resets ready for the next customer. We will ignore various practical issues such as running out of gum, jamming of the mechanism, timing out if an attempt to purchase is abandoned before completion, and so on.

Figure 15.15 A vending machine modeled as a Moore state machine.

Figure 15.16 A vending machine modeled as a Mealy state machine.

Figure 15.15 shows a Moore machine model of the vending machine. The states are shown as ellipses labeled with the state name. For convenience, the state names here are chosen to represent the total amount of money that has been inserted in the current transaction, so the possible states are 0, 5, 10, and 15. We are ignoring excess money being inserted, so 15¢, the sale price, is the maximum we need to consider. The state also shows the action the machine takes in that state, which is either Passive, if it's waiting for more money, or Vend, if sufficient money has been inserted. The links between the state nodes represent transitions between states. The transitions are labeled with the event that causes the transition. The transition between state 15 and state 0 has no associated event because it happens automatically once the activity in state 15 has completed. Again, for simplicity we're using the money amount inserted to identify the event, and so the possible inputs are either nickel (N) or dime (D).

We are taking state 0 as the initial state. The possible events in this state are either N or D, which take us to state 5 or state 10, respectively. From state 5, the event N or D takes us to state 10 or state 15, respectively. From state 10, either type of coin input (N or D) will take us to state 15. In state 5 and state 10, a reset will return the machine to state 0. In state 0, state 5, and state 10 the vending machine is passive, but once state 15 is reached it produces the output (some gum). The transition from state 15 to state 0 is not labeled with an event because the transition is taken automatically when the activity in state 15 is completed.

Figure 15.16 shows the same vending machine as a Mealy state machine. We have the same states, but there is no output (Passive or Vend) associated with a state. Instead, the outputs are associated with the transitions between states, which are labeled both with the event causing the transition and the output produced in that transition.

State machine descriptions have been popular for several decades as a way of specifying and implementing automata: we see many examples in everyday life in the form of digital watches, cell phones, video recorders, and so on. However, for a long time it was believed that state machines did not scale well because of the explosion of states required to deal with all of the combinations of variables. In a naïve state machine interpretation, the total number of states required is of the same order as the product of the number of different values for each involved variable. For instance, if we have three variables with 3, 4, and 5 possible values, respectively, the likely number of states is $3 \times 4 \times 5 = 60$ to cover all possible combinations of values. Of course, some combinations may actually be associated with the same state, but considering the large num-

ber of variables involved in a typical business system, the idea of a state-based description seems futile.

A major contribution was made by David Harel, who pointed out that state descriptions could be composed in more efficient ways, and also arranged in a hierarchical fashion so that state transition diagrams are not swamped with detail. The key notion is that states can be defined to represent the composition of other (sub-) states, allowing us to consider some states as a group, rather than having to deal with them individually.

For example, we could have the situation shown in Figure 15.17(a). States *U* and *V* both have a similar transition to another state *W*. By placing *U* and *V* in a composite state *X*, we only need to show a single transition from *X* to *W* instead of the individual *U* and *V* transitions to *W*. The small filled circle and the line that originates from it represent the initial transition into the default state when the state machine is started. Figure 15.17(b) shows a composite equivalent of Figure 15.17(a). The state machine begins in state *X*. When state *X* is entered, it immediately transitions to state *U*. In both of the cases shown in Figure 15.7, the state machine therefore begins in state *U*.This is known as OR composition because we can only be in state *U or* in state *V* within the composite state *X*.

A second form of composition allows a state to contain several regions. These are considered *orthogonal* because state transitions can evolve separately in each of the regions. Figure 15.18 shows a simple example. The states in Figure 15.18(a) have composite names to make it easier to see the relationship between the composed and the decomposed variants. In Figure 15.18(b), a new state *S* has been introduced with two regions, *R* and *T*. Informally, *R* and *T* could be visualized as substates of *S*. The crucial difference in this case is when we enter state *S both* of its regions become active, with a default transition to state *U* in region *R* and a default transition to state *X* in region *T*. The combination of the state *U* in *R* and state *X* in *T* in the composed variant is equivalent to the state *U-X* in the decomposed variant. For example, receiving event *J* in the composed variant has no effect in region *R*, but causes a transition to state *W* in region *T*. This is equivalent to the transition from state *U-X* to the state *U-W* in the decomposed variant.

Figure 15.17 OR composition of states: (a) decomposed and (b) composed.

Figure 15.18 AND composition of states (a) decomposed and (b) composed.

This approach is sometimes known as AND composition because we are in one state in region *R and* one state in region *T*. Another way of looking at this is to see it as an AND composition of OR compositions (since in any orthogonal region we can only be in one state at a time).

One additional feature shown in Figure 15.18(b) is the use of a guard associated with the transition from state *V* to state *U*. The guard condition must evaluate to "True" to enable the transition. The result is that event *F* will not trigger the transition in region *R* unless we are in state *Y* in region *T*. This is consistent with the equivalent transition in Figure 15.18(a) between state *V-Y* and state *U-Y*.

Although the savings made by OR and AND composition may appear trivial in simple examples, they can become very significant in larger models. For instance, in Figure 15.18(b), region R has two substates and region T has three substates. If these were increased to four substates and five substates, respectively, the orthogonal version would have $(4 + 5) = 9$ nodes, but the nonorthogonal version would have $(4 \times 5) = 20$ nodes.

It's important to recognize that state models are primarily intended to define the locus of control within a given scope. A state model does not in itself define any actions that should be taken. However, an action definition can be linked to the state model in order to identify the (relative) time that the action should be taken. The main possibilities are for actions to take place as part of a particular transition from one state to another (Mealy semantics) or in a particular state (Moore semantics). Within a state we can further distinguish between actions that are taken just after the state is entered, sometime during the state, or just before the state is exited.

To provide a more concrete foundation, we'll now look at in a little more detail at the constituent parts of a typical state machine. The following description is fairly generic: particular methodologies and tools may use different terminology or use some features in a slightly different way. We can make a few simplifying assumptions to avoid being swamped with detail—in particular, we'll assume that state machines can be arranged hierarchically (i.e., a state machine can contain other state machines) but we'll ignore orthogonal regions. From the discussion given earlier, it should be obvious

how the basic concepts could be extended. Later in this chapter we look at how these issues are handled in a particular methodology.

We are mainly concerned here with behavioral aspects, and so a useful starting point is to consider what kinds of element in a conceptual model might have dynamic behavior. The obvious candidate is an object type: but not every object type, because some consistently behave in the same way and therefore have no behavior that requires any special consideration. In programming terms, objects of this latter kind can be thought of as some data plus a group of functions that always operate on the data in the same way. Of course, there are methodologies that help us identify objects with more "interesting" behavior, but they lie outside the scope of this chapter. Here, we will just assume that we have identified some "interesting" object type and determined that we want to link it to a state definition. As before, we use ORM notation to illustrate the main concepts involved.

Since we are dealing with an object *type*, we could potentially have many instances of the object. This leads us to the same definition/instance partitioning that we have already seen in the context of process definitions. As discussed earlier, not all object types will warrant a state machine, but for those that do we will have a state machine definition associated with the object type and an instance of the state machine associated with each instance of the object type. Each state machine definition will include several state definitions and/or references to nested state machines (each of which could, of course, contain other states and/or state machines). For the rest of this discussion we assume that a state could be either a simple state or an embedded state machine.

States can have substates, but no state can be a substate of itself. At any point in time, a state machine instance will be in exactly one state, which must be a state defined in the state machine definition of which this state machine is an instance. This is illustrated in Figure 15.19.

A transition takes place between a source state and a target state. The source and target states can be the same state. There are a few special cases to consider. Some state types (e.g., "Initial") can only have outgoing transitions and some (e.g., "Final") can only have incoming transitions. Normal states have both incoming and outgoing transitions. Because of the simplifying assumptions we've made, both of the states involved in a transition must be defined in the context of the same state machine, as shown in Figure 15.20.

Figure 15.19 State machine definitions and instances.

Figure 15.20 Transitions in a state context.

A transition is typically defined as being triggered by a particular type of event. Some event types may be explicitly prohibited in some states (as opposed to just being ignored). A transition may optionally be guarded by an expression, which would have to evaluate to "True" in order for the transition to be taken. In general, we want to arrange things so that there is no ambiguity in the definition. If we are in a particular state, there should be only one possible reaction to a given event. One possibility is that the action is ignored in this state[1]. A second possibility is that the event is consumed internally by the state—for example, it may play a role in the actions that are performed during this particular state. A third possibility is that the event causes a transition to another state. In this last case, we need to be sure that the transition is defined unambiguously. It's possible that a particular event may cause one of several possible transitions, depending on circumstances. We can accommodate this by using guards on the alternatives. Ideally, the guards should be disjoint and exclusive (i.e., at any point in time, exactly one guard on the competing outgoing transitions will evaluate to "True" and all of the other guards will evaluate to "False"). Particular methodologies and tools may have different views on what to do if this is not the case.

Actions may be performed during transitions (Mealy semantics) or in a state (Moore semantics). Some methodologies allow both, some may allow finer definition of when actions are performed in a state (on entry, during, on exit, …), and some may allow lists of actions. In the case of a list of actions, the order of invocation is defined, but the order of completion may not be. For example, the first action in a list may take longer than all of the other actions in the list, and so it may actually be the last to complete. In computing terms, we often distinguish between actions that are synchronous or asynchronous.

1. This does not necessarily mean the event is ignored by the whole state machine: if we have orthogonal regions the event may be relevant in another region, but our simplified approach allows us to put aside that possibility.

Figure 15.21 Events, guards, and actions.

At a business level, the same concepts are present: we either wait for an action to complete before we proceed or we complete our part and carry on, assuming that any related activity will complete in due course. In the state machine definition we assume that we wait for actions that are synchronous and don't wait for actions that are asynchronous. Figure 15.21 shows the relationship among events, guards, and actions.

State machines provide an excellent approach for defining systems that react to external events. However, state machines can also respond to internal events—events created by another state machine in the same environment or even events created by the state machine itself. Table 15.2 shows some of the situations that could potentially create a triggering event.

To allow fine control over event processing, we will assume that each event notification is time stamped. A state machine instance does not necessarily have to process events in the strict order that they were notified. For instance, some types of event may be deemed to be particularly important, with the implication that they should be processed before any other kind of event. To allow for such possibilities, event processing is typically arranged in ascending order of both priority and time stamp (i.e., the earliest event of the highest priority event type is processed first). It's common to arrange the priorities so that each state machine processes the messages that it has itself originated as its highest priority (most implementations have some way for a state machine instance to send a message to "self").

Various mechanisms can be employed to check and control event passing. Stepping outside conceptual modeling for a moment to consider implementation environments, message checking is usually minimal within the same address space. However, in the kind of distributed environment that's commonplace in a large enterprise, such mechanisms may be very necessary.

Table 15.2 Some candidate events.

Feature	*Event*	*Explanation*
State *S*	Entered(*S*)	State *S* has been entered
	Exited(*S*)	State *S* has been exited
Value *V*	Changed(*V*)	Value *V* has changed
Expression *E*	True(*E*)	Expression *E* now evaluates to "True"
	False(*E*)	Expression *E* now evaluates to "False"
Action *A*	Started(*A*)	Action *A* has started
	Ended(*A*)	Action *A* has ended
	Suspended(*A*)	Action *A* has been suspended

A typical mechanism (not shown here) is for a state machine instance to put a sequence number on events it creates. At a minimum, this will allow missing messages to be detected. Figure 15.22 shows some aspects of event notification, including the possibility that we may not want to process events at the time that they are generated, but instead defer processing until some time in the future.

Figure 15.22 Event notification.

We have two possibilities here. Either we can defer event processing until a specific time and date or we can delay processing by specified time interval. If neither of these is specified, then the implied action is to process the event immediately.

Actions are initiated as a result of transitions, either directly (Mealy) or indirectly (Moore). The transition can be triggered by an event. It can also occur because certain conditions have been met[2]. A transition may have a guard expression, which must evaluate to "True" in order for the transition to be taken. The evaluation of the guard expression is normally carried out near the time of attempting to take the transition: for example, immediately after the triggering event has been recognized. A transition can also occur simply because the action (or final action if there are several) in the preceding state has completed. This is known as a *completion transition*. In Figure 15.23 we accommodate this possibility with the isDoneForNow flag.

In some methodologies, an explicit event is notified on completion of an activity instead of assuming the event implicitly. The flag in the model has to be isDoneForNow rather than just isDone because the next action that we take after the transition could be the same action type that we have just completed (i.e., the action type that we have just completed is not "done" for all time: it's simply completed an instance of its execution in the current context).

Figure 15.23 Transition triggering possibilities.

2. One could also consider "we recognize that we now meet condition X" as a kind of event, but in most implementations events are closely aligned with message-passing mechanisms. We might therefore expect conditions to be manifested as guards, rather than events.

Figure 15.24 Event and action data.

An action can use make use of some data (i.e., some fact instances). The action specification will show the fact types used by the action type (not shown here). The usage of fact instances by an action instance will be consistent with the relevant specification.

Events can also have parameters, each of which will reference a fact instance. Again, these will be consistent with the fact type references specified for the event type (not shown here). The relationship with fact instances is shown in Figure 15.24.

At this point we've reached the same ground level for state machines already seen for processes. In terms of an information system, we are defining controls over the primitive actions of creating, reading, updating, and deleting data. Of course, operating at this primitive level would be unbearably tedious for a business, which is why it's important for us to understand how we can construct information systems that carry out these operations in a way that's consistent with the intentions of the business.

15.4 Foundations for Process Theory

Process modeling is an application that has motivated many technologists to build automated systems to support process definition, execution, management, and so on. There are probably well over a hundred tools and techniques in current circulation, with more arriving all the time. Unfortunately, most of these approaches are based on vague intuitions about process features, with the result that their associated semantics are often a matter of guesswork. This is an unfortunate state of affairs if we have ambitions to derive information system elements from business models, because our model does not provide a definitive baseline to work from. The problems resulting from this

have prompted interest in more formal ways of defining processes. The formality is not necessarily something that needs to be exposed at the business level, but we need to be sure that our business models are built on firm foundations.

The technique that has attracted most attention in this regard is the use of Petri nets. These were first defined in Carl Adam Petri's Ph.D. thesis in 1962. They immediately attracted interest from theorists, but it took a little longer for them to find practical uses. Petri nets are now used in many fields, and the basic concepts have been extended in several ways. This section looks briefly at how they can be used to represent business processes. References to more detailed treatments of Petri nets are given in the chapter notes.

Fundamentally, a Petri net is a bipartite graph with variable markings at some of the nodes. There are two node types: *places* and *transitions*. Places and transitions are connected by directed arcs. The possible connections are place to transition or transition to place, but never place to place or transition to transition. A Petri net is used to represent a dynamic situation, but the structure of the graph does not change as the situation unfolds. Instead, the different conditions are marked by different arrangements of *tokens* at the places in the network. A particular configuration of tokens is sometimes referred to as the *state* of the network— "state" here does not have quite the same meaning that it does in the context of state machines. Tokens are created at places, or removed from places, according to certain rules. A token cannot exist at a transition. Graphically, places are usually shown as large open circles, tokens as small filled circles, and transitions as either rectangles or thick bars. Figure 15.25 shows a simple Petri net with tokens at some of the network places.

A transition is said to be *enabled* if it has a token at each of its input places. An enabled transition can *fire*, and in doing so it consumes a token from each of its input places and produces a token at each of its output places. For the configuration shown in Figure 15.25, transition t1 is enabled but t2 and t3 are not (t3 does not have a token at all of its input places). The resulting situation after transition t1 has fired is shown in Figure 15.26. A token has been removed from p1 and a new token added at p2.

Conceptually, tokens are consumed or created at transitions. Transitions can have input places (the arc from the place is directed toward the transition) or output places (the arc to the place is directed away from the transition). The same place can be both an input place and an output place relative to some transition. In Figure 15.25, transition t1 has p1 as an input place and p2 as an output place. Place p3 is both an input place and an output place for t3.

Figure 15.25 A simple Petri net.

Figure 15.26 The network of Figure 15.25 after transition t1 has fired.

Figure 15.27 The network of Figure 15.26 after transition t2 or transition t3 has fired.

Now both t2 and t3 are enabled. Since we have only one token at p2, only one of t2 and t3 can fire. In a basic Petri net this situation is nondeterministic: we cannot decide in advance which of the transitions will fire. Adding more tokens to place p2 would not solve the problem: transitions t2 and t3 would still compete for each token. However, in this particular case, the same state results whichever transition fires. If t2 fires, it consumes the token at p2 and produces a token at p4. If t3 fires, it consumes the tokens at p2 and p3 and produces a token at p3 and p4. The final result is shown in Figure 15.27. At this point, no further transitions are enabled and activity ceases.

In order to make use of a Petri net, we have to map its elements onto the real world. Generally speaking, tokens are taken to represent objects (people, goods, machines), information conditions, or object status; places represent buffers, channels, geographical locations, conditions, or situations; and transitions represent events, transportation, or transformation. In the specific case of business process modeling, transitions are usually taken to represent business activities, whereas the combination of places and tokens determines when the activity is carried out.

The basic constructs of the Petri net can be composed into more convenient forms to provide building blocks to model business processes. Some constructs appear frequently and it's useful to be able to insert these into a model without repeating the basic Petri elements that they are built from. Processes commonly need to split into parallel branches at some point and then reconcile the branches at some later point—for example, by waiting until activities on preceding parallel branches have completed. Various forms of "Split" and "Join" (or "Merge") are commonplace in process models. Both of these come in two flavors: OR and AND. An OR-split sends a token along one path or another; an AND-split sends a token along multiple paths. Similarly, an OR-join expects a token to arrive on one of several branches, whereas an AND-join expects a token on all branches.

Figure 15.28 Higher level split and join constructs.

Figure 15.28 shows the high level elements with their equivalent constructs in a basic Petri net. The notation used for the high level constructs is taken from YAWL: a workflow language based firmly on Petri nets. Note particularly that the construction for the OR-split is nondeterministic in its basic Petri net form. To make it more useful for business modeling, we assume the existence of a business rule (or rule set) that is able to steer a token in the appropriate direction.

Figure 15.29 (a) A “repeat-until” loop. (b) A “while-do” loop.

Given these constructs, we can begin to put together networks of increasing complexity. As an example, Figure 15.29 shows how we can define some classical looping constructs using Petri nets.

The standard Petri net allows us to construct models of dynamic behavior, but it has some shortcomings for process modeling. Nets that result from the primitive constructs become too large and clumsy for realistic cases, and it's difficult to represent fine-grained business information. Because of this, a number of extensions have been proposed to make Petri nets more useful in a business context.

The first idea is to use different types of token. In the basic Petri net, tokens are all of the same kind. If we retain the place/transition network but allow multiple token types, we open up some new possibilities. For example, we can make the firing of a transition dependent on token types and values. Conceptually, we can think of the different token types as being tokens of different colors, and so a network extended in this way is known as a colored Petri net.

A second important idea is to provide ways of organizing networks hierarchically. We have already seen this concept from other perspectives. In the case of Petri nets we have networks containing subnetworks, potentially nested to several levels. At a high level, we can represent a subnetwork (which might correspond to a subprocess) as a single block, allowing us to selectively hide detail in complex networks.

A third extension introduces the dimension of time. Activities are typically modeled as transitions, but in a basic Petri net, transitions are considered instantaneous. We can provide a more realistic simulation by time stamping tokens so that we have information about their temporal evolution, and allowing transitions to have delays, that would represent the passage of time while an activity is carried out in the real world.

Figure 15.30 shows an example of a hierarchical network for a process to handle complaints that uses different types of token to indicate different cases that are being progressed.

Figure 15.30 A process for handling complaints.

Figure 15.31 Parallel split pattern: (a) Petri net and (b) Pi calculus.

Several other candidate formalisms for business processes have been proposed. One is the use of state machines, of the kind we have already discussed. There is not a single state machine formalism, but several state machine frameworks have been formally defined, notably by David Harel. If we begin with a state machine that has a formal description, then we can build other constructs on top that inherit the formal foundations.

Another approach that has received a fair amount of publicity is the use of process algebras—the most prominent example being the *pi calculus* defined by Robin Milner. In these approaches, a series of algebraic expressions replaces a network as the basis for defining process structure. For space reasons, we won't go into detail here, but the following example provides something of the flavor of the approach. As a simple example we can take a common workflow pattern: a parallel split. (We discuss this and other patterns in more detail later in the chapter.) Figure 15.31 shows equivalent specifications of this pattern using both Petri nets and pi calculus.

There is currently no agreement on whether algebraic approaches offer significant advantages over other approaches such as Petri nets. They are certainly not at all useful at the business level, but their place is more likely to be hidden under the covers of a tool that presents a more user-friendly interface.

So far, we've looked at some basic ideas about describing information dynamics in a fairly generic way. The features of process- and state-based models discussed so far can be found in many methodologies and tools that address this area. To provide a more concrete illustration, we turn to one of the most widely used modeling approaches: the Unified Modeling Language (UML).

15.5 Modeling Information Dynamics in UML

The UML includes a range of different diagram types—13 types in all in UML Version 2 (an increase over the nine types available in UML Version 1). Figure 15.32 shows the range of diagram types: boxes with dashed lines are simply summary supertypes and not actual types of diagram.

UML supports information modeling through class diagrams and, to a lesser extent, object diagrams. These have been covered in some detail elsewhere in this book and we won't discuss them again here. UML also has several diagram types that can be used to model behavior.

Figure 15.32 UML 2 diagram types.

The four types of interaction diagram shown in Figure 15.32 are mainly of interest in software development and have limited application in business modeling. Sequence diagrams and communication diagrams describe messages passed between object instances. Interaction overview diagrams are a specialized variant of activity diagrams—we discuss activity diagrams in more detail shortly. Timing diagrams show the detailed timing of interactions between various system elements and are mostly only relevant to real-time systems.

We first take a very brief look at use case diagrams. A use case describes a connected series of actions performed by one or more actors in relation to the system under discussion. An actor may be human or machine, but, by convention, both are shown graphically as a stick figure icon. The use case description is usually expressed from the viewpoint of one particular actor—the principal actor. A use case diagram contains none of this description and is essentially no more than a list of the relationships between actors and use cases (and sometimes between use case and use case), depicted in a graphical rather than a tabular form. A simple example is shown in Figure 15.33.

The oval shapes in Figure 15.33 represent use cases. These are usually named with a verb phrase that summarizes the actions being performed in the use case. The optional rectangular boundary represents the system, which is slightly misleading since the use case describes the actor's interactions with the system (i.e., the use case is not "in" the system).

The UML standard describes various extensions to the basic concept. For instance, some shared activity that potentially appears in several use cases can be extracted out into a separate use case and referenced by an «include» relationship. Some activity that only occurs in exceptional cases can be separated out into a separate use case and referenced in an «extend» relationship.

Figure 15.33 A simple use case diagram for an automated teller system.

Actors and use cases can be treated as classes, with subtyping used to show hierarchical relationships between actors and between use cases. However, these and other elaborations often end up causing more problems than they solve and should be used with caution. The valuable part of a use case is the actual behavioral information contained in the use case description, but, strangely, this aspect is not defined in UML.

A common error is to equate a use case with a business process. This may be true in very small systems, but in most major enterprises, cross-functional business processes span many organizational units. This makes it unlikely that individual actors (from whose perspective the use case is being described) will see the whole of the process. It is more likely that a use case describes a segment of a process, as shown in Figure 15.34 (note: this figure is *not* a UML use case diagram!).

Use cases that overlap in their descriptions (as do UC2 and UC3 in Figure 15.34) should corroborate each other. If not, further investigation is required. Gaps in the coverage of the process (as between UC3 and UC4 in Figure 15.34) also require investigation. Perhaps we have a missing use case? Or perhaps our understanding of the process is not correct?

Figure 15.34 Relationship between use cases and processes.

Although use cases can help uncover some dynamic aspects of business information, they are not sufficiently rigorous to be our primary modeling tool, and they are best seen as a step on the path to producing a more comprehensive and precise business model. The remaining two UML behavioral diagram types can be pressed into service to model dynamic business behavior, and we will look in a little more detail at some of their features.

Activity Diagrams

Activity diagrams are mainly focused on the flow of control within a set of related activities. The level of granularity is not defined by UML, so the scope of each individual activity is at the discretion of the modeler. This has the advantage that activity diagrams can be used for both high level and low level descriptions, but places the onus of defining activity boundaries onto some other authority. Activities are considered to be composed of atomic actions. Activities can be interrupted, but actions cannot, which implies that any interruption to an activity must take place on a boundary between actions. We have already seen flow-of-control chaining of activities earlier in this chapter, and it's not surprising that UML activity diagrams are seen as a candidate notation for documenting business processes. The underlying semantics of activity diagrams changed between UML versions 1 and 2. In UML 1, activity diagrams were seen as a special kind of state machine description and shared a great deal of terminology with UML statecharts. In UML 2, the semantics of activity diagrams became realigned to be closer to the semantics of Petri nets.

An activity diagram consists of a number of nodes connected by arrowed lines (directed edges). There are three main types of node: action nodes, object nodes, and control nodes. Action nodes represent some work being carried out and are shown as a rectangle with rounded corners. The rectangle typically contains the name of the action: other information may optionally be added that we won't consider here. Object nodes, shown as rectangles with square corners, represent an object type, for example, a business entity such as an invoice that may be passed from one activity to another. Object nodes are generally only shown when some special consideration needs to be given to the objects being passed: routine use of business objects by activities is just assumed. The "flow" along the activity edges is either of control tokens, indicating the transfer of control from one activity to another, or of objects. Since the edges are directed, there is no assumed direction of flow, although diagrams are conventionally laid out so that the main flows are either left to right or top to bottom.

Control nodes are abstract activity nodes that coordinate flows in an activity diagram and come in various subtypes. An *initial* node is represented by a filled circle and represents a starting point for the activity diagram. There can be more than one initial node. There are two kinds of *final* node. An *activity final node* is shown as a target symbol, and an activity diagram can have more than one of these. As soon as an activity final node receives a token, the entire activity immediately terminates even if there are other control tokens active in other paths. A *flow final node*, shown as an "X" in a circle, destroys any token that reaches it, without affecting any other concurrent flows.

A *decision* node, denoted by a diamond shape, has one input edge and one or more output edges. A token arriving on the input edge will leave on only one of the output edges (i.e., exclusive-or). To control this, the output edges have guard conditions, denoted inside square brackets. The guard conditions should be complete and disjoint to ensure that token flow is not inadvertently halted or duplicated. A default branch can be provided by specifying it with the guard condition [else].

The diamond shape used for a decision node is also used for a *merge* node. This performs the complementary action to a decision node by bringing together multiple alternate flows on its input edges. Any token arriving on an input edge is offered to the output edge. There is no synchronization of tokens arriving on the incoming edges.

A *fork* node splits a flow into multiple concurrent flows. It is shown as a solid bar with one input edge and two or more output edges. Each token arriving on the input edge is duplicated and offered to each output edge. There is no sequencing implied between the outgoing flows.

A *join* node is used to combine parallel flow paths. The join notation is the same as for a fork, except that the join has multiple input edges and one output edge. An optional join expression determines how incoming tokens are combined to produce an output token. The default is "and", implying that the join waits for a token to arrive on each incoming edge before producing a token on the output edge (i.e., synchronization).

The UML 2 specification does not require that branches introduced in the diagram should be balanced. In other words, multiple paths introduced by decisions or forks do not necessarily have to be recombined by the complementary merges or joins. Although this provides maximum flexibility in defining a process, it can also provide subtle traps for the unwary and can lead to unsound process definitions.

Figure 15.35 shows an example activity diagram for a simple sales process. The diagram is based on an example given in the UML specification, modified slightly to contain examples of the features we discuss. Although the modified diagram is still syntactically legal, it represents an unsafe process design, as we will see shortly. The diagram is self-explanatory, but not unique, in the sense that the same process could have been depicted in a number of slightly different ways.

Figure 15.35 An activity diagram for a simple sales process.

For example, the join and the fork that immediately follow it could have been combined into a join/fork construct that would have appeared as a thick bar with two edges entering and two edges leaving. Combining a join and a fork in this way can make diagrams slightly tidier, but has no effect on the semantics.

The execution of the process can be simulated by tracing the flow of a token in a particular instance of the process represented by the diagram. The token originates at the initial node. A customer places an order, which is received by the company's sales department. The first decision is to either accept or reject the order. The two guard conditions, [rejected] and [accepted], are mutually exclusive. If the token takes the [rejected] branch, it immediately reaches the merge and appears on its output edge. After the order is closed, the token reaches the activity final node, which immediately ends the process instance. If the token takes the [accepted] branch, the order is filled. When the token reaches the fork, two tokens are generated. The upper token causes the order to be shipped, and the lower token stimulates a series of actions to invoice the customer and eventually receive payment.

The upper and lower branches have no implied timing constraints: either of them could be the first to complete. The following join waits for a token to arrive on both branches before emitting its own single token. This immediately forks into two other branches. The token on the upper branch reaches the merge that we have already discussed and causes the order to be closed. The token on the lower branch reaches a decision. If the customer is a new customer, then they are added to the customer database, but for an existing customer the token is immediately consumed by the flow final node.

The second fork in the diagram is not balanced (there is no corresponding join), but we eliminate the token in the lower branch because it eventually reaches a flow final node. This ensures that adding the customer to the database does not interfere with the closing of the order. However, the design is unsound because the lower branch is in a race with the upper branch. Once the order is closed, a token will reach the activity final node and immediately terminate the process, regardless of the status of the token in the other branch. This could result in a new customer *not* being added to the database, contradicting the process designer's original intentions.

Figure 15.36 An activity diagram with swimlanes.

One other feature of activity diagrams is support for swimlanes. These are optional, but can be used to indicate features such as separation of responsibilities, usage of resources, geographical distribution, and so on. As with other features, the precise meaning of a swimlane is not defined in UML: the interpretation is left up to the user of the activity diagram. Figure 15.36 shows the process model of Figure 15.35 with swimlanes added. In Figure 15.36 the swimlanes are drawn to indicate who carries out the actions (customer or sales clerk). The invoice object is used by both, and so is shown on a swimlane boundary. Multiple orthogonal swimlanes can be defined: for example, one axis could represent organizational units and the other could represent resource types. The result resembles a kind of Venn diagram with the actions placed in whichever swimlane intersection is appropriate. However, this quickly gets difficult to draw and can be confusing to read.

Activity diagrams also provide for the hierarchical composition/decomposition of actions. A structured activity of this kind can be treated at one level as a single action, but can be expanded if required to show the breakdown into its subordinate activities. Activities can be nested in this way to any level that's useful.

Activity diagrams have a number of other features that we haven't space to discuss here. Examples include signaling between system elements, events used to trigger activities, the definition of interrupts and interruptible regions of activity, and so on. Many of these features can be useful in the definition of software, but are less relevant for conceptual modeling.

State Machine Diagrams

In UML version 1, state machine diagrams were known as statecharts and were closely aligned with activity diagrams, to the extent that an activity diagram could be considered as a special type of statechart. In UML version 2, state machine diagrams and activity diagrams occupy positions that are rather more distinct.

UML state machine diagrams have many of the same main constituents as other state diagramming conventions. States are normally shown as rectangles with rounded corners. The name of the state can be shown free-form inside the rectangle, in a separate compartment in the rectangle, or as a name tag attached to the rectangle, as shown in Figure 15.37.

States are interconnected by transitions, shown as directed lines. The lines typically begin on the border of one state and end on the border of another state—we discuss some special cases later. A transition can begin and end at the same state, and it's possible for there to be more than one transition between a particular pair of states.

Figure 15.37 Alternative state notations in UML.

Figure 15.38 (a) Simple composite state. (b) Orthogonal state.

A state can be simple or composite. Composite states contain other substates within them. If a state has an active substate, then the containing state is also active. A composite state that is subdivided into multiple regions is known as an orthogonal state. Figure 15.38 shows examples of a simple composite state (containing exactly one region) and an orthogonal state (in this case, with two regions).

The small black nodes in Figure 15.38 are special *pseudostates* that represent the initial conditions for their respective states. Within a region, only one substate can be active at any one time, so the state shown in Figure 15.38(a) can be in state S1 or S2 or S3 or S4. In contrast, the state shown in Figure 15.38(b) can be in state S1 or S2 and also in state S3 or S4. A state can also contain nested state machines—called submachine state machines—which are semantically equivalent to composite states. Submachines are a convenient way of encapsulating common behaviors for reuse.

Actions can be carried out in a state at one of three times:

(a) on entry to the state, denoted as entry / action
(b) during the state (perhaps continuously in a loop), denoted as do / action
(c) on exit from the state, denoted as exit / action.

All of these are optional. The "during" actions do not begin until any entry actions are completed and will be completed or aborted before any exit actions begin. This ensures that entry actions are always carried out before anything else and that exit actions are always carried out after anything else. Actions carried out within a state correspond to the Moore machine semantics discussed earlier. If required, the actions can be shown on a state machine diagram by listing them in a compartment of the state symbol just below the compartment with the state name, as shown in Figure 15.39.

Figure 15.39 Showing state actions.

Figure 15.40 A labeled transition.

Transitions between states can have a label that consists of up to three parts. The normal notation is: e [g] /a , where

- e identifies the triggering event causing the transition (the event may have parameters),
- g is a guard expression, enclosed in square brackets, and
- a identifies a list of one or more actions, preceded by a "slash" character.

All three components are optional, so zero, one, two, or all three of the components could be present. Figure 15.40 shows a typical transition with all three components present.

If no event is specified, the transition is enabled when the actions in the preceding state have completed (a *completion transition*). If a guard expression is specified, it must evaluate to "True" for the transition to be taken. The guard expression is evaluated at the point in time that the transition is about to be taken. If it evaluates to "False", the transition will not be taken, even if it has been enabled by an event. If actions are specified on the transition, they are completed before entering the target state. This corresponds to the Mealy machine semantics discussed earlier. UML state machines can therefore take on the characteristics of a Moore machine, a Mealy machine, or both.

Events are handled by a state machine one at a time, with run-to-completion semantics. This means that an event is only recognized and processed after the processing of the previous event is fully completed. This simplifies the state machine because it can never get into a situation of processing an event while it is in some intermediate configuration (e.g., part way through a transition from one state to another). Events do not necessarily cause transitions: an event can be recognized and processed within a state. Events can also be recognized but *deferred* in a particular state. A deferred event is saved and passed to the following state, where it must be either accepted or deferred once again. This continues until the event is no longer deferred or is where it triggers a transition.

UML defines a number of *pseudostates* that can be either the source or the target of a transition. The notations used for each of the pseudostates are listed in Table 15.3.

The *initial* pseudostate is the source for a single transition to the default state of a composite state. There can be at most one initial pseudostate per region. A *final* pseudostate indicates completion of its containing region. The containing state for a region is considered completed when all of its contained regions are completed. If all regions contained in a state machine are completed, then the entire state machine terminates. This is equivalent to destroying the object (in the programming sense!) with which the state machine is associated.

Table 15.3 UML pseudostates.

Symbol	*Pseudostate*	*Incoming transitions*	*Outgoing transitions*
●	initial	Not permitted.	One. Guards and triggers not permitted.
◉	final	One or more.	Not permitted.
○	entry	One or more.	One or more. Triggers not permitted.
⊗	exit	One or more.	One or more. Triggers not permitted.
•	junction	One or more.	One or more. Triggers not permitted.
◇	choice	One or more.	One or more. Triggers not permitted.
▮	fork	One.	Two or more. Guards and triggers not permitted.
▮	join	Two or more. Guards and triggers not permitted.	One. Guards and triggers not permitted.
Ⓗ	shallowHistory	One or more.	One. Triggers not permitted.
Ⓗ*	deepHistory	One or more.	One. Triggers not permitted.
×	terminate	One or more.	Not permitted.

An *entry* point provides for the entry to a state machine or composite state that targets a specific state within the enclosing region. The entry actions associated with the enclosing state are executed before the transition to the enclosed state. A transition that enters an *exit* point implies an exit from the enclosing state machine or composite state. Any exit behavior of the enclosing state is executed after the transition from the enclosed state. Entry and exit points can be shown within the boundary of a state or on the boundary line of the state.

Junction pseudostates are used to combine together multiple transitions and/or to split a transition into several branches, each with a different guard condition. The latter case is known as a static conditional branch. A *choice* pseudostate is a dynamic conditional branch—it evaluates the guards on its outgoing transitions dynamically so as to reflect the results of actions performed in the same run-to-completion step.

Fork pseudostates split transitions that are targeted at states in different orthogonal regions. *Join* pseudostates perform the complementary operation of combining transitions that originate from states in different orthogonal regions.

A transition entering a *deepHistory* pseudostate restores the most recently active configuration of the composite state and all its substates. A transition into a *shallowHistory* state is similar, except that the configurations of any substates are not restored.

Figure 15.41 A state with a hidden decomposition.

A composite state can have at most one deepHistory pseudostate and at most one shallowHistory pseudostate, In either case, if the composite state had not been active previously (i.e., it has no history), the outgoing transition from the history state is taken.

Entering a *terminate* pseudostate implies that the execution of the containing state machine is terminated. This is equivalent to destroying the object with which the state machine is associated—again "destruction" is intended here in the programming sense.

In a hierarchical state machine, more than one state can be active at the same time. If a nested state is active, then its parent state and all of its ancestors back to the top-level state are also active. There can be at most one active state in any one region. An orthogonal state can have an active state in each of its regions. Entering an orthogonal state implies entering each of its regions (explicitly or by default). Similarly, exiting an orthogonal state implies exiting all of its regions.

It's sometimes convenient to hide the decomposition of a composite state: suppressing selected details can allow attention to be focused on larger issues. A composite state with a hidden decomposition is shown graphically by adding a small icon to the state that looks like two horizontally placed states with a connecting line, illustrated in Figure 15.41. A tool that supports state modeling would typically have some convenient mechanism for expanding back to the detailed view of the composite state. Hiding detail in this way does not change the state model, only the way that it is displayed.

We've glossed over some of the finer details of how state machines are defined in UML. For instance, there are situations that lead to nondeterministic results. One example is two transitions exiting the same state, triggered by the same event, but with different guard conditions. If the guard conditions overlap so that both evaluate to "True", both of the transitions are enabled but only one of them will fire. In such cases the algorithm for selecting which transition to fire is determined by the implementation. If guards on transition branches are framed so that it is possible for all guards to simultaneously evaluate to "False", the model is considered ill-formed. A simple cure for this latter situation is to include a branch with the guard [else], which will be chosen if the guards on all the other branches evaluate to "False". Another complex area is the processing of events in state machines that have extensive nesting of substates. A given event may be relevant to several substates, but deciding which one actually gets to process the event is not a trivial exercise.

State machines are very popular with some users of UML, but most of these appear to be in application areas dominated by either real-time constraints or the need to be highly reactive to external events. In a typical business information system, time constraints are less onerous: a processing delay is more likely to be annoying than fatal. Historically, business systems were also less concerned with being reactive: in fact,

early batch processing systems could hardly be said to be reactive at all. With the increasing trend toward online processing, reactivity is becoming more important. Anecdotal reports on the use of state machines in business applications suggest that business people find state-based descriptions less intuitive than process-based descriptions. It also seems to be true that a large proportion of the developers of business software are unfamiliar with the basic concepts of state machines. However, state machines do offer some powerful capabilities: in particular, the ability to generate application code from suitably constructed models. The next section discusses one approach that relies on state modeling to provide the behavioral aspects of an executable specification.

Executable UML

As the name suggests, Executable UML is a dialect of UML that is focused on a style of modeling that results in a model that can be compiled and executed. It is based on a fusion of ideas from the earlier Shlaer–Mellor methodology and standard UML. An executable UML model is built around three core concepts: specifying structural aspects, such as data, using UML class diagrams; specifying dynamic behavior and control using state machine diagrams; and specifying actions using an action language. The use of class diagrams and state diagrams is close to standard UML, but with a few detailed differences. At present, UML does not define any notation or syntax for an action language, although there *is* a specification for the semantics that must be supported by such a language. The consequence is that each variant of Executable UML has its own language. A proposal for standardizing an action language is currently in progress within the OMG.

The first stage of analysis separates different areas of concern into different *domains* to be modeled. A domain contains a set of cohesive concepts that can be considered without requiring detailed knowledge of other domains. Domains are interconnected by *bridges* that resolve intercommunications required between domains. The assumptions made in one domain can form the requirements for another domain. For example, a business-related domain may assume the existence of a communication network with certain characteristics. These business domain assumptions contribute to the requirements for the communication network domain. Bridges allow separation of concerns, so, for example, the elements in a business domain can be developed and refined without considering the details of any technical domain. This also makes the system design more resilient. For instance, modification to a technical domain does not necessarily have to impact a business domain, so we could "unplug" one technical realization and "plug in" a different one without impacting the business, as long as we can define a suitable bridge.

The functionality required in a domain can be established in any desired way. We assume here that this is done through use case analysis. The information contained in the use case descriptions can be dissected to determine an initial set of classes. It's likely that the initial class definitions will be extended and refined many times as the design progresses.

Associations between classes on a class diagram are handled in a slightly different way from standard UML: each association is given a unique identifier and the associa-

tion ends are named with verb phrases instead of the noun phrases of standard UML. Classes have operations defined in the usual way. These operations typically compute and return some value, or create, delete or modify some object (or perhaps a combination of these). Each instance of a class (i.e., an object) has a life cycle. For some classes, the behavior of their instances does not vary. An object is created, it reacts in a fixed way to any stimulus (a call to one of its operations) and eventually it goes away—not a very exciting life cycle. Other classes have behaviors that change over time, depending on the history of events to which the object has reacted. For these classes, a state machine can be defined to model their life cycle. State machines in most implementations of Executable UML adopt a subset of the possibilities defined in the UML specification. In particular, actions are typically encapsulated into procedures that are executed on entry to a state (standard UML allows other possibilities).

The procedures are defined in an action language. The language is at a higher level of abstraction than standard programming languages such as C# or Java in order to remove dependence on a particular software platform. Action language statements are conceptually concurrent: decisions about how to serialize these are left to the model compiler. Similarly, such concerns as data access, persistence, distribution, and so on are not specified in the action language. The only direct manipulation that can be specified in an action language is manipulation of the model elements themselves. At the time the model is compiled, the action language specification can be mapped onto the desired architecture. In principle, the same model could therefore be used in an embedded system, a Web service, a distributed application, and so on.

Executable UML has been applied successfully in a variety of applications. Most of these are real-time applications that lend themselves very naturally to state-based specification. Products are available from a small number of vendors who tend to be focused on the reactive systems market. However, there seems to be no reason why the concepts should not be equally applicable to general business systems, so the current focus may just be a reflection of historical market positioning, rather than any inherent feature of the technical approach.

15.6 Business Process Standards Initiatives

The last few years have seen an increase of interest in standards related to business process modeling. This has been fueled partly by the growing use of process modeling by business and partly by the desire of software vendors to establish a prominent position in an expanding market. Three areas are particularly prominent in current standards activity.

Notation. Most process models use some form of diagrammatic presentation of the process design. This typically takes the form of a flow diagram showing sequencing, branching, and so on, perhaps with some additional annotation to indicate related features such as the use of resources and organizational structure. The graphical syntax (including symbols and connection rules) vary from one notation to another. Perhaps more importantly, the semantics also vary. For example, a diamond shape may look similar in two different notations but have subtly different implications in each. A

common notational standard would simplify staff training, make process designs more readily understandable, and reduce the scope for semantic variation. A leading proposal in this area is the Business Process Modeling Notation (BPMN).

Metamodel. Although humans typically view a process design through some form of notation, automation of the process requires the construction of a data structure and procedures that can navigate through the structure. Transferability of process designs between tools and between organizations would be facilitated by having a common metamodel to ensure consistency. A leading proposal in this area is the Business Process Definition Metamodel (BPDM).

Execution. The rapid growth of Web services has led to a demand for a consistent way of expressing business processes in an executable form. The basic Web protocols are stateless, but business processes may require lengthy conversations between distributed participants in a process. One way of tackling this problem is to have a specialized programming language targeted at a business process execution engine. A leading proposal in this area is the Business Process Execution Language (BPEL).

Table 15.4 A partial list of initiatives related to process standards.

Organization	*Initiative*
BEA	Process Definition for Java (PD4J)—now absorbed into BPEL
IBM	WSFL—now absorbed into BPEL
Microsoft	XLANG - now absorbed into BPEL Windows Workflow Foundation (WF)
The Business Process Management Initiative (BPMI)	Business process standard activities—in particular, BPMN—were merged with OMG (see below) in mid-2005
The Organization for the Advancement of Structured Information Standards (OASIS)	Business Process Execution Language (BPEL)
The Object Management Group (OMG)	Business Process Modeling Notation (BPMN) Business Process Definition Metamodel (BPDM) Business Process Maturity Model (BPMM)
The Workflow Management Coalition (WfMC)	XML Process Definition Language (XPDL) Workflow XML protocol (Wf-XML) Workflow Application Program Interface (WAPI) The WfMC reference model for workflow
World Wide Web Consortium (W3C)	Web Services Choreography Description Language (WS-CDL) Web Service Choreography Interface (WSCI) Web Services Conversation Language (WSCL) Web Service Definition Language (WSDL)

We will look at BPMN, BPDM, and BPEL in more detail shortly, but it's worth noting a few other proposals to give an idea of the scope of current activity. Table 15.4 shows a partial list of organizations active in process-related fields and the initiatives with which they are involved. The "alphabet soup" created by the large number of abbreviations and acronyms can appear daunting, but we don't have space here for a full explanation of their relationships. The chapter notes give some additional references.

Business Process Modeling Notation

BPMN defines a graphical syntax for drawing flowchart-like process diagrams. It is intended to be primarily business facing, not a description of a technical realization. It does not specify how to execute a process on a computer: it assumes that BPMN models will be mapped to whatever execution environment is required. The BPMN specification provides a sample mapping to BPEL, and in future, mappings to various forthcoming W3C standards are anticipated. BPMN began life as part of the Business Process Modeling Initiative, but progress on the standard moved to the OMG in mid-2005. Various drafts of Version 1 of the standard have been available for some time, and work has now started on Version 2.

An *activity* represents work that is performed during a process, denoted in BPMN as a rectangle with rounded corners. Activities can be nested with the details of subactivities hidden. Compound activities are identified by adding a "+" icon to the activity symbol. BPMN also uses the term "Task" for atomic activities that have no lower level breakdown. Various other icons can be added to the basic activity shape to indicate other types of activity, such as looping, transactional, compensating, ad hoc (the sequencing of any internal activities is not defined), multiple instance (several instances of the activity can be performed in parallel), and so on. With some restrictions, these can be combined to indicate such things as a looping ad hoc compensation activity.

Events represent something that happens during a process and are denoted by circular symbols. There are three main subtypes: start events have a single-line circle, intermediate events have a double-line circle, and end events have a thick line circle. Icons can be added within the circle to give a more precise indication of the type of event, such as *message*, *timer*, *error*, and so on.

Splitting and joining in flows can be shown using a diamond-shaped *gateway* symbol. The symbol can carry an additional icon to indicate the nature of the split or join. Variants include *exclusive-or* (the default if no icon is shown), *and*, *inclusive-or*, and *complex* (where the split or join conditions are determined by expressions).

Activities, events, and gateways are linked by connecting objects, which come in three varieties. *Sequence flows* show the order (sequence) in which activities will be performed during a process and are denoted as solid lines with a filled arrowhead. Sequence flows can also carry additional markings to indicate that a flow is conditional, or is a default flow. *Message flows* show the interchange of messages between participants in a business process and are denoted as dashed lines with unfilled arrowheads. *Associations* are used to connect data, text, and so on with flow objects and are denoted as dotted lines with an optional open arrowhead. One use of these is to show the inputs and outputs of activities. Figure 15.42 shows a typical BPMN process diagram.

Figure 15.42 A BPMN diagram.

The process in Figure 15.42 starts with the receipt of a loan application (shown as a message). After the loan assessment activity the control flow passes to one of the three following activities. Each of these is a compound activity: the internal details are not shown here. The final exclusive-or gateway joins the split paths together and the process ends. In the referral case, there is a time event associated with the applicant review. If the review is not completed within the given period it is escalated to be dealt with by a supervisor. The escalated review process also has branching in the flow, but rather than using gateways, the subprocess uses decisions on sequence flows and implicit joining of flows. The escalated review subprocess also shows reuse of the definition of the compound activities for loan approval and loan rejection.

The current version of the BPMN standard has been criticized for a lack of clarity in some areas. Some aspects, such as the detailed rules for the processing of flow tokens at gateways, are not fully specified. In other areas, BPMN offers alternative notations for some constructs, such as splitting flows following an activity, but gives little or no guidance on when one approach should be preferred over another or whether subtle semantic differences exist between the alternatives. However, such issues will no doubt be addressed in future versions of the standard.

Business Process Definition Metamodel

BPDM provides a capability for representing business processes in a way that is independent of any particular notation or process modeling methodology. The concept of a

metamodel is intended to provide a kind of vocabulary of concepts related to process modeling, with a clear definition of the meanings of the concepts and the ways in which one concept can relate to another.

BPDM supports two complementary views of business processes—"Orchestration" and "Choreography". Orchestration is concerned primarily with what happens and when, and is usually defined in some flow-based notation (although the notation itself is outside the scope of BPDM). Choreography describes how business entities collaborate in a process, taking into account that the entities may be independent and may have their own internal processes. The focus is on the responsibilities of the role players in a process and the interactions between them.

BPDM is intended to separate out several potentially overlapping concerns. One is the distinction between the definition of a process and the notations that might be used to depict the process. BPDM process definitions are designed so that different tools can depict or utilize a process definition in different ways yet still be interoperable. For example, a drawing created using one vendor's modeling tool could be opened and executed using another vendor's business process management system. Another potential overlap is between the specification of the goals of a process and the definition of the actions required to achieve the goals. BPDM allows for the specification of process "contracts" independently of the definition of the implementations that correspond to the contracts, allowing processes to be defined at any appropriate level of detail.

Although BPDM and BPMN are now both proceeding under the auspices of the OMG, they had different origins and, for a while, were not wholly consistent. Recently, great efforts have been made to harmonize these two standards, so, for example, BPDM now uses the notation of BPMN and has some extensions to provide additional functionality that is specific to BPMN.

The BPDM Metamodel is based on the "Meta Object Facility" (MOF) that also underpins UML and some other OMG standards. As well as providing a baseline for concept definition, MOF also supports an XML syntax that can be used for transferring business process models between tools and between organizations. The specification of BPDM is split into a series of packages, each of which uses the classifier-oriented notation of MOF to show the relevant concepts and their associations. Effectively, each package forms a submodel within BPDM. Table 15.5, which is adapted from the descriptions provided in the BPDM specification, summarizes the packages and their major features.

As BPDM is not yet a fully ratified standard, experience in its use has so far been limited to a small number of companies and researchers. By its nature, it will be mostly relevant to organizations involved in process modeling technology, such as the builders of modeling tools. It is too early to tell whether the standard as defined will fully meet the needs of industry, but it clearly occupies an important position in the process modeling spectrum.

Table 15.5 BPDM packages.

Category	*Package*	*Features*
Common Abstractions	Composition Model	Relates metamodels to the real-world entities they ultimately represent. Facilitates integration with business process runtimes and rule engines, as well as uniform performance, enactment, and execution across business process management suites. Enables the construction of libraries of orchestrations and choreographies and custom frameworks for recording data about ongoing orchestrations and choreographies.
	Course Model	Extends the Composition Model to connect parts in time (Succession). For example, a succession connects one step in a process to another to indicate that the second step happens after the first.
Common Behavior	Happening and Change	Introduces dynamics, in particular, time ordering of life cycle events, such as starting and ending a process. This facilitates the integration of rule and monitoring systems with models of dynamics, such as orchestration and choreography.
	Processing Behavior	Enables behavioral happenings to be ordered in time as parts of other behavioral happenings. The model predefines a specific connection for races, where behavioral happenings start at the same time and abort each other when the first finishes. It also defines a change condition for detecting life cycle events in behavioral happenings.
	Simple Interaction	Enables interactions to be treated like any other step in a processing behavior, ordered in time, with start and end events. The model is the basis for flows between process steps and between participants in a choreography.
Activity Model		Captures orchestrations in way that facilitates modification to processes as business conditions change, for example, due to mergers and acquisitions. It uses interactions to represent inputs and outputs, enabling choreographies to be specified between the process and its environment, as well as between the performers responsible for steps in the process. Also provides specific additions for BPMN to provide BPMN names for special usages of BPDM concepts and additional functionality specific to BPMN.
Interaction Protocol Model		Defines choreographies, enabling interactions to be grouped together into larger, reusable interactions. For example, an interaction that exchanges goods between companies might be used with other interactions within a larger protocol representing a partnership of the companies.
Infrastructure Library		Defines a reusable metalanguage kernel and a metamodel extension mechanism for UML. The metalanguage kernel can be used to specify a variety of metamodels, including UML, MOF, and CWM.

Business Process Execution Language

BPEL is intended to extend the current standards for Web services standards to add explicit support for business processes. BPEL is essentially a programming language that uses XML syntax. The original version, called BPEL4WS, was based on Microsoft's WSFL and IBM's XLANG approaches. The more recent version has been renamed WS-BPEL. Unless otherwise qualified, BPEL refers to both of these versions. BPEL can be used for abstract process definition, but its main application is in the definition of executable business processes.

A process description in BPEL includes elements that define the main activities in the process, correlation sets (used to correlate messages with the process), partner links, variables used in the process, and various kinds of handler for compensation, faults, and events. BPEL assumes the existence of one or more Web Service Definition Language (WSDL) files that define the Web service interfaces. WSDL is a separate standard, which is used for other purposes besides support for BPEL.

BPEL was formed by combining the previous XLANG and WSFL initiatives and includes features of both block structured languages (from XLANG) and directed graphs (from WSFL). Some examples of BPEL elements are shown in Table 15.6.

As a consequence of the mixed inheritance of BPEL, the implementer may have a choice of alternative constructs that could potentially produce the same effect. For example, a sequence could be realized using either <sequence> or <flow> with <link>. In some cases, it is necessary to use the block structured part of the language: for example, a deferred choice can only be modeled using the <pick> construct. In other cases, it is necessary to use the graph-based part of the language: for example, two parallel processes with a one-way synchronization require a <link> inside a <flow>. Some of the semantics of the constructs are very subtle, and effects can be produced that may not be entirely what the process implementer intended.

Table 15.6 Some elements of BPEL.

Type	Element	Purpose
Basic activities	<invoke>	calls an operation
	<receive>	waits for an input message
	<reply>	sends an output message
	<assign>	updates variable values
	<wait>	blocks execution for a given period
Block structured	<sequence>	set of activities to be executed in the listed order
	<if>	conditional execution (replaces <switch>)
	<pick>	defines a list of event–activity pairs—when an event from list occurs, the corresponding activity is executed
	<while>	the activity is executed while the condition is true
	<flow>	specifies parallel execution of the contained activities
Graph oriented	<link>	used to control the order of execution inside a <flow> element

Figure 15.43 (a) An unstructured process. (b) A structured equivalent.

One consequence of the block structured elements of BPEL is that some process constructs that can be expressed in a graph-oriented approach cannot be expressed in BPEL, and some remodeling may be required. Figure 15.43 shows a nonstructured process flow and one possible way in which it could be remodeled. In order to get to a block structured form, it's important that any looping constructs are arranged so that there is only one entry point into the loop and only one exit point from it.

Even trivial processes produce many lines of BPEL XML code, so we don't show an example here. The code is also quite difficult to read for humans and is really intended for machine execution. Most process modelers are likely to prefer a tool with a user-friendly interface that has the ability to generate BPEL from a model, rather than writing BPEL code directly.

BPEL is firmly targeted at business processes and is probably the most powerful language for defining business process execution. Of course, conventional programming languages such as Java could be used in place of BPEL, but would probably require more effort and require a great deal of detailed code to produce an equivalent result. One negative aspect of BPEL is that it is syntactically quite restrictive, which causes problems when it is required to transform a non-BPEL model (for example, in BPMN) into BPEL code. BPEL has had little competition to date, but it is possible that some of the newer work based on Web services choreography and the Semantic Web may overlap with BPEL, and other approaches may emerge in the future.

15.7 Standard process patterns

A significant contribution to the establishment of a common foundation for business process modeling was made by the identification of 20 or so workflow patterns by van der Aalst, ter Hofstede, and others. The patterns have subsequently provided a valuable benchmark for describing the characteristics of alternative process modeling notations,

methodologies and tools. Many others have made use of the patterns, and some further references are given in the chapter notes.

Table 15.7 provides a summary of the original workflow patterns. The diagrams use mostly BPML notation. There is no particular significance to the numbering system used.

Table 15.7 Standard process patterns

Nr	*Pattern*	*Description*
1	Sequence	
		Sequence is the simplest pattern. It simply expresses the notion that one activity follows another, so in the diagram, Activity 2 starts when Activity 1 finishes, Activity 3 starts when Activity 2 finishes, etc.
2	Parallel split	
		In the parallel split pattern, a token released by the completion of a preceding activity is split into multiple tokens, perhaps at an AND gateway, enabling several succeeding activities. In the diagram here, there is no defined ordering between Activity 2 and Activity 3 (Activity 2 could precede Activity 3, Activity 3 could precede Activity 2, or Activities 2 and 3 could be simultaneous).
3	Synchronization	
		In synchronization, tokens from multiple branches merge together, perhaps at a gateway, so that one token is emitted once a token has been received on all of the preceding branches. In the diagram here, both Activity 1 and Activity 2 must complete before Activity 3 can be executed.

Table 15.7 Standard process patterns (continued)

Nr	*Pattern*	*Description*
4	Exclusive choice	
		An exclusive choice is a decision point at which the thread of execution will take one of several alternative branches. The branching decision is typically defined by a business rule or rule set. One of the branches may be designated as the default. In the diagram here, Activity 1 can be followed by either Activity 2 or Activity 3, but only one of these will be executed.
5	Simple merge	
		A simple merge is a point at which two or more alternative branches come together without synchronization. Only one of the preceding branches is assumed to be active (later patterns deal with merging multiple active branches). In the diagram here, either Activity 1 or Activity 2 must complete before Activity 3 can be executed.
6	Multiple choice	
		In multiple choice, the thread of control diverges into several parallel branches on a selective basis. The branching decision is typically defined by a business rule or rule set. One of the branches may be designated as the default. In the diagram here, Activity 1 can be followed by Activity 2, Activity 3, or both.

Table 15.7 Standard process patterns (continued)

Nr	*Pattern*	*Description*
7	Synchronized merge	
		Synchronized merge represents the completion of activities that have been started on parallel branches. One or more of the preceding branches can be active. All preceding active branches must complete before the succeeding Activity 3 can be executed. This assumes that the gateway has information about which preceding activities are expected to complete (sometimes called a "structured workflow" assumption). In the diagram here, Activity 1 or Activity 2 must complete (if they were started) before Activity 3 can begin.
8	Multiple merge	
		Multiple merge is the unsynchronized convergence of several alternative branches. More than one of the branches can be active. The succeeding activity will be executed when any of the preceding activities completes. If multiple branches complete, the succeeding activity is executed multiple times. In the diagram here, either Activity 1 **or** Activity 2 must complete before Activity 3 can be executed, and a new instance of Activity 3 will be started for each preceding branch that completes.

Table 15.7 Standard process patterns (continued)

Nr	*Pattern*	*Description*
9	Discriminator	 The discriminator provides for synchronized convergence after 1 out of *m* alternative branches completes. One or more of the preceding branches will be active. Completion of any preceding active branch will forward a token to the succeeding activity. The completion of subsequent branches is ignored. Use of a Complex gateway in place of the XOR gateway shown here would allow *n*-out-of-*m* merges: an expression would determine which path(s) actually have to join. In the diagram here, either Activity 1 or Activity 2 (or both) must complete before Activity 3 can be executed.
10	Arbitrary cycles	 Arbitrary cycles of activity that have more than one entry or exit point in a repetitive cycle of activities. This pattern supports unstructured repetition in a process without requiring special looping constructs. This is not possible in “block structured” modeling languages. In the diagram here, Activities 4, 5, and 6 form a loop that can be entered at either points p1 or p2.
11	Implicit termination	 Implicit termination specifies that when a process is terminated, any subprocesses should also be terminated. The intention is that on termination there are no active activities, there is no activity that can be made active, and the process is not deadlocked. In BPMN this is specified by ensuring that every thread of control in a process ends with an end event (as shown in the diagram here).

Table 15.7 Standard process patterns (continued)

Nr	*Pattern*	*Description*
12	Multiple instances, without synchronization	 Multiple instances of the same activity are initiated in parallel without requiring that they resynchronize on completion. Each instance of the activity runs independently of the other instances. Note: this is not the same as a parallel split because these are all instances of the same activity. In the diagram shown here, Activities y.1, y.2, and y.3 are all separate instances of Activity *y*. This diagram (and the other multiple instance diagrams) departs from the BPMN notation to make the patterns more distinct.
13	Multiple instances with a priori design-time knowledge	 Several instances of the same activity are initiated in parallel with intended resynchronization on completion. The number of instances is known when the process is defined. Each instance of the activity (such as Activities y.1, y.2, and y.3 shown here) runs independently of the other instances, but succeeding activities (such as Activity z above) cannot be triggered until every instance has completed. Note: this is not the same as a parallel split/join because these are all instances of the same activity.
14	Multiple instances with a priori run-time knowledge	 Several instances of the same activity are initiated in parallel with intended resynchronization on completion. The number of instances required is determined in the process itself, perhaps only at the very last moment before the instances have to be executed. Each instance of the activity (such as Activities y.1, y.2, and y.3 shown here) runs independently of the other instances, but succeeding activities (such as Activity z) cannot be triggered until every instance has completed.

Table 15.7 Standard process patterns (continued)

Nr	*Pattern*	*Description*
15	Multiple instances without a priori knowledge	
		Several instances of the same activity are initiated in parallel with intended resynchronization on completion. The number of instances required is determined in the process itself during execution of the parallel instances and is known only after the last instance finishes. Each instance of the activity (such as Activities y.1, y.2, and y.3 shown here) runs independently of the other instances, but succeeding activities (such as Activity z) cannot be triggered until every instance has completed.
16	Deferred choice	
		Deferred choice involves a single thread of control that is split into two or more threads, all of which could execute concurrently. Only one branch will complete, but which one of the branches will complete will not be known until after the split. This can be considered as a “race” between the activities in the alternative branches. Once a decision is made in one of the branches the other alternatives are immediately withdrawn. The special marker in the gateway in the diagram given here is the BPMN notation for an “Event-based split”.

Table 15.7 Standard process patterns (continued)

Nr	*Pattern*	*Description*
17	Interleaved parallel routing	
		This pattern provides for a partial ordering of activities, also constraints that prevent certain activities from running in parallel. In the diagram shown here, Activity 1 is followed by Activities 2 and 3 in any order (but not simultaneously), followed by Activities 4 and 5 in parallel, and then Activity 6. The properties of the "Perform A2 and A3" activity must be set appropriately (e.g., its subactivities must be defined as Sequential in BPMN). The tilde symbol on "Perform A2 and A3" is the BPMN notation for an ad hoc activity.
18	Milestone	
		A milestone is used to indicate that an activity in one branch of a process cannot be initiated once another branch of the process passes a certain point. In the example here, Activity *s* cannot be carried out once Activity *m* has completed. This is difficult to model accurately in BPMN because it relies on state, rather than the triggering of activity. Some limited workarounds have been proposed, but these do not totally capture the intention.
19	Cancel activity	
		The cancel activity pattern indicates that a process has progressed to a point where it becomes clear that a particular activity should be cancelled. In the example given here, the event created after Activity 2 causes a message to be sent that causes Activity 3 to exit at the point shown. The symbol used on the event in the diagram here indicates an error.

Table 15.7 Standard process patterns (continued)

Nr	*Pattern*	*Description*
20	Cancel case	 The progression of the process instance makes it clear that a particular case in progress should be canceled. This is similar to the previous pattern, except that we are canceling a complete case (and all of the activities relating to it) instead of just an activity. If the case represents the whole process, we can just use a Terminate event, which kills every activity in progress.

The original workflow patterns focused on flow-of-control aspects. A similar analysis has been applied to workflow data patterns by Russell and colleagues, who define five main categories of pattern: Data Visibility (7 patterns), Internal Data Interaction (6 patterns), External Data Interaction (12 patterns), Data Transfer (7 patterns) and Data Routing (7 patterns). The patterns illustrate the way in which many data-related features are handled by current workflow tools. Although no claim is made that the list of features is exhaustive, the patterns probably represent a reasonable catalog of concepts that need to be considered in the context of process data.

Since the data patterns were gleaned by examining current practice, they effectively make statements about the way that process data is typically handled in a software implementation. However, here we are mainly concerned with modeling processes at a conceptual level. This corresponds to the distinction between an ORM information model and its implementation in (say) a Relational Database Management System. Examination of the documented data patterns shows that a number of them are primarily abstractions of implementation patterns, rather than information usage patterns.

As an example, we could consider two activities, *A* and *B*, with a dependency between them. Activity *B* needs some information created by Activity *A*, and so it would be natural for an instance of Activity *A* to precede temporally the corresponding instance of Activity *B*. This is true whether we are considering the process at a purely conceptual level or looking at the implementation. If the activities are implemented in some object-oriented software language, a standard approach would be to create some object within the Activity *A* instance and then pass it to the corresponding Activity *B* instance—in other words we could say that Activity *A* *pushes* the data to Activity *B*. If the two activities are operating in the same memory space, this could be accomplished

by passing a reference to the relevant object, otherwise an additional step of serializing the object may be required.

From a conceptual viewpoint, these last details are irrelevant. Whether Activity *A* "pushes" an object to Activity *B*, whether Activity *B* "pulls" an object from Activity *A*, or, indeed, whether any "object" is involved at all is purely an implementation issue. From a conceptual standpoint we certainly need to be concerned about the rights and responsibilities of activities with regard to data, but we explicitly do not want implementation issues to become entangled with the process definitions.

Processes elements (at any level) utilize data in three main ways: (a) to compute some result of value to the process, (b) to determine a course of action, and (c) as information to guide process activity, for example, as reference material for a human involved in the process. Our main need is to define data scoping: which activity needs which item of data and for what purpose. If we have ambitions to automate the construction of automation systems to support business processes, this is clearly crucial. Even if we have more modest ambitions, such as simply providing a checklist to implementers, this still seems to be an important facet of process definition.

From a high level perspective, we can classify an activity's usage of a data item in terms of whether it is limited to reading the data item or whether it can also modify it. If modification is allowed, we can further distinguish between the ability to create new data, modify existing data, or delete existing data. (One could also predict other uses for such information, such as identifying the data privileges that should be given to particular user roles, but that is outside the scope of this chapter.) In contrast, the types of features that we would *not* want to specify at a conceptual level include the following.

Communication mechanisms
- between different levels (process, subprocess, activity)
- between different elements (e.g., activity to activity)
- between process elements and the environment

Data transfer mechanisms
- "push" (data explicitly sent to process element)
- "pull" (data requested by process element)
- pass by value versus pass by reference
- with or without locking
- with or without transformation
- etc.

These issues are largely outside the scope of a business process model, and so we don't consider them in detail here. Of course, a business model alone does not contain sufficient information to determine the detailed mechanisms (and deliberately so). What we do have is the ability to identify the relationships between low level atomic activities and low level data manipulation and query operations. We are not limited to operating only at this level because, as we have seen, we can compose collections of low level elements into higher level elements until we reach a more convenient level of expression. However, this still leaves the realization underconstrained: there could be

many ways of linking data to process for the same conceptual design. The choices between the alternatives are somewhat orthogonal to the conceptual elements. For example, are we building a self-contained system or one that is distributed over a network? How many users do we intend to support? Are we concerned about issues such as availability and scalability? The additional information required is what might be termed the *technical architecture* of the system, which is more concerned with the engineering aspects. In principle, given a definition of the technical architecture of a system, we should be able to generate the necessary system elements required from a business model. We have already seen how this could work in examples such as Executable UML. In the future, we can expect similar ideas to be widely adopted as a natural outgrowth of conceptual modeling.

Exercise 15.7

1. We are going to make a series of cabinets in a really organized way! First we'll select a plan and get the wood required. Next we'll lay out the parts on the pieces of wood according to the plan. We'll cut out the parts and then assemble them. This will require us to check the fitting of the joints and adjust them as necessary before gluing them together. Finally we'll sand the cabinet smooth, paint it, and then move on to the next cabinet. We have the following resources in our workshop: saw, ruler, pencil, sandpaper, paint, screwdriver, chisel, and glue.
   (a) Draw a process structure for making our cabinets, using any suitable notation.
   (b) Identify which resources are required for each activity.
   (c) Are the resources consumed?
   (d) Are there any points where the process should be suspended?
   (e) Could we have multiple parallel instances of any of these activities?

2. The following schema shows information related to the testing of manufactured items.

   (a) Given the schema here, are there any implied constraints on the sequence in which business objects can be created?

(b) What can we say about this information model from the standpoint of business processes?

3. Assume that we have a hair dryer controlled by two switches: SW1 controls the motor speed and SW2 controls the heating element. The possible switch positions are shown here.

SW1 has three positions and SW2 has two positions. Assume switches can't go past their range (e.g., "Up" has no effect if the switch is already all the way up).

A possible Moore state machine diagram is shown, using Up and Down as the input events for the two switches.

(a) From a common-sense point of view, does this model seem sound?
(b) Produce an equivalent diagram using composite states. Assume the starting position is Heat = Cool, Fan = Off.
(c) Add a "Swirl" switch to the composite state diagram, with three positions: Off, Medium, and High. Assume the starting position is Swirl = Off.
(d) What would the diagram look like without composite states?

4. A company has a process for planning new products. Management sets out the product strategy. Sales identifies product needs from the market. Once these have been done, Sales develops proposals for new products.

After the proposal is complete, Sales carries out market testing and then estimates the likely sales volume. At the same time, Finance estimates the costings and then either reserves a budget from existing funds or sets up a loan facility with the company's bank. Once this is done, Finance produces a financial plan for the new product

When the financial plan and the sales estimates are available, Management carries out a feasibility study. If the feasibility is rated good, Production carries out production planning and the process ends. If the feasibility is rated poor, the viability of the product concept is reassessed.

If the product concept is judged not viable, the process ends. If the product is judged still potentially viable, Sales is asked to develop new proposals

Draw a BPMN-style process diagram for the scenario just described. Show the Sales, Management, Finance, and Production departments in lanes. Use only the following symbols:

5. The following diagrams show two workflow designs. The workflow on the left is unstructured. The workflow on the right is intended to restructure the workflow on the left into a form that is appropriate for a block-structured workflow language. Assume that the values of w and x are set by Activities D and F, respectively.
Are these two workflows equivalent?

15.8 Summary

ORM provides a powerful methodology and notation for capturing the structure of information at a conceptual (i.e., implementation-free) level. From an ORM model one can infer some partial ordering on the dynamics of the information. For example, information on mandatory fact types associated with a primary object type must be available no later than the time at which that object type is instantiated. References between primary fact types must be ordered in a way that maintains referential integrity—in a database system these typically appear as foreign key references.

However, these limited constraints fall far short of a definition of a process. Some other forms of expression are necessary to define explicitly the intentions of a business in executing its activities. Business process models provide one way of doing this; state models provide another. Nor does ORM provide any way of expressing other aspects of process elements, such as the resources involved. This is not a criticism of ORM: it's just that a description of a business involves many aspects in addition to the structure of the information it uses.

Clearly, it may be appropriate to consider a business from many different perspectives, but these different points of view must be reconciled because they are ultimately an expression of the same business system. Trying to model these different perspectives with ORM alone would be like trying to model an object-oriented system in UML and being limited to using only class diagrams. This implies a multiviewpoint approach to business modeling. However, combining different perspectives does not mean using independent models: we are really talking about different views onto one common model.

Both process models and state models can be used to express behavioral aspects. The techniques that they embody have evolved in similar ways because they have responded to the same pressures, such as the need to deal with large, complex sets of activities. UML includes both activity and state modeling, but its use in business modeling is compromised by its primary orientation toward defining software artifacts.

Standards are becoming more important in process modeling. Informal standards, such as the popularly quoted workflow patterns, have helped establish a better understanding of important features of business processes. Standards of a more official kind, such as BPMN, BPDM, and BPEL, are helping facilitate the uptake of process modeling by businesses of all kinds.

Chapter Notes

Rummler and Brache (1995), Burlton (2001) and many others have described methodologies for capturing business process definitions, and Hammer (1997) and others have emphasized the restructuring of businesses around processes. Havey (2005) gives a broad summary of several business process modeling approaches. A good introduction to Workflow is given by van der Aalst and van Hee (2004). Many valuable contribitions to state modeling have been made by David Harel: see for example Harel and Politi (1998). Recently, the systematic modeling of business processes has become an area of great interest in Information Technology. Several stan-

dardization initiatives are under way within organizations such as the Object Management Group (OMG) and the World Wide Web Consortium (W3C), including the proposed Business Process Modeling Notation (BPMN), Business Process Definition Metamodel (BPDM), Business Process Maturity Model (BPMM), Business Process Execution Language (BPEL) and many others, including related initiatives in the area of Web Services. The latest public information on OMG standards (BPMN, BPDM and BPMM and also UML) can be found at *www.omg.org*. Members of OMG also have access to the latest work in progress on each of the standards. The latest information on BPEL (now WS-BPEL) is available at *www.oasis-open.org*. The relevant W3C standards relating to process orchestration and process choreography can be accessed at *www.w3c.org*. The use of UML for business modeling is discussed by Eriksson and Penker (2000). Lassen and van der Aalst (2006) and Ouyang et al (2007) discuss techniques for translating workflow processes to BPEL. Many papers have used the "classic" workflow patterns defined by van der Aalst, ter Hofstede and others, and there is now a workflow patterns Web site (*www.workflowpatterns.com*). This provides detailed descriptions with a Petri-net orientation. Coverage of the same patterns from a pi-calculus viewpoint can be found at *www.pi-workflow.org*. Additional discussion of state modeling and ORM is given in (Morgan 2006).

This page intentionally left blank

16 Other Modeling Aspects and Trends

16.1 Introduction
16.2 Data Warehousing and OLAP
16.3 Conceptual Query Languages
16.4 Schema Abstraction Mechanisms
16.5 Further Design Aspects
16.6 Ontologies and the Semantic Web
16.7 Postrelational Databases
16.8 Metamodeling
16.9 Summary

16.1 Introduction

Section 2.4 included an overview of the information systems life cycle. So far, we've focused on the stages of this cycle that deal with the conceptual and logical design of the data. We've seen how to perform conceptual data modeling using ORM, ER, or UML, and applied the Rmap algorithm to map a conceptual schema to a relational database schema. We've discussed database manipulation using SQL and XML, modeled business processes and states, and applied conceptual schema transformations and lower level optimizations to improve the efficiency of relational designs.

This chapter discusses other aspects and recent trends related to modeling and querying information systems. We'll be covering a lot of ground, so the treatment of each topic is necessarily brief. You can use the references in the chapter notes to dive deeper into the areas examined.

Section 16.2 provides a brief introduction to data warehousing and online analytical processing. Section 16.3 discusses some very high level languages for querying information systems. Section 16.4 outlines some ways of performing schema abstraction, enabling the modeler to focus on various aspects of a schema by hiding other details.

Section 6.5 discusses further design aspects, such as coordinating data and process models, and user interface design. Section 16.6 outlines some recent work on ontologies and the semantic web. Section 16.7 examines recent extensions to or replacements of relational databases, such as object-oriented capabilities and XML. Section 16.8 provides an introduction to metamodeling, in which schemas themselves are treated as instances of a higher level metaschema. Section 16.9 furnishes a summary to complete this final chapter.

16.2 Data Warehousing and OLAP

Most commercial information systems are built to support heavy volumes of transactions by many users on a daily basis. Examples include banking, insurance, and order processing systems. These *Online Transaction Processing (OLTP)* systems typically require quick throughput for their largely predefined range of transactions, especially update transactions. To improve performance, historical data is often archived once it reaches a certain age, reducing the size of the data sets used for daily operations. A single organization may have several OLTP systems (e.g., purchasing, sales, inventory, customer records), possibly implemented using different kinds of DBMS or other software applications, and the coupling between such systems may be weak or even nonexistent.

Over time, businesses became aware that the collective information contained in their various systems had great potential for analyzing market trends and improving their business processes. However, their OLTP systems were unsuitable for executives to perform this task, given the poor performance and complex interface for ad hoc analysis queries (e.g., aggregated multitable joins and nested correlated subqueries). Moreover, insufficient integration or history made some queries simply impossible. Partly to address the problem of integrating data from prerelational systems for analy-

sis purposes, IBM proposed the notion of an "information warehouse". Although performance problems delayed acceptance of this idea for some years, a later proposal for a "data warehouse" by Bill Inmon (1993) was enthusiastically embraced by the business community, and nowadays most large companies have a data warehouse.

A *data warehouse* is an enterprisewide, integrated, historical database of information extracted from individual data sources for the purpose of supporting analysis of the business by management. During analysis it is read-only. Since the patterns and trends sought from the analysis tend to evolve slowly, and some imprecision is acceptable, updates are performed in bulk according to an agreed schedule (e.g., weekly).

The construction of a data warehouse for a large enterprise can be a lengthy task. To exploit the benefits as soon as possible, the data warehouse is often built iteratively, one subject area at a time. As subject areas are added to the data warehouse, they may be used to load data marts (see Figure 16.1). A *data mart* is a smaller "departmental warehouse" focused on one subject area, often containing more summarized and less detailed data than the data warehouse. For end users who perform their analysis within one subject area, a data mart provides a simpler model adapted to their needs, and the smaller data volume often leads to better performance.

Many different approaches to data warehousing exist. Sometimes, data marts are built first and used to incrementally construct the data warehouse. However, if an analytical query may span multiple subject areas, it is critical that an overall enterprise architecture be in place to make the appropriate connections.

It has been argued that the data warehouse should be implemented as a fully normalized relational model, based directly on the enterprise data model, with no summary data, postponing any denormalization and aggregation to the data marts loaded from the warehouse (e.g., Moody and Kortink 2000). Current practice however usually does incorporate denormalized and summarized data in the data warehouse (Silverston et. al. 1997). A nice overview of alternative approaches is provided by Jukik (2006).

Figure 16.1 Data is extracted into the data warehouse then loaded into data marts.

Before extracting data to the data warehouse, the business analysis needs should be determined and the relevant data sources identified. The data sources may include operational databases as well as spreadsheets and legacy systems. Any details in the data sources irrelevant to the analysis needs should be removed. For example, the phone number and street address of a store are probably not of interest, but its city probably is. The remaining data now needs to be transformed to remove errors (applying integrity rules). For example, a gendercode field with many 'F' and 'M' entries might include a few instances of 'D', 'G', 'J', or 'N'. Assuming that these are typographical errors, can you guess the intended letter? (*Hint:* look at the keyboard.)

We must also ensure that all facts of the same type are represented in the same way. As a trivial example, a customer's birthdate might be recorded as a character string in a 'DOB' field in one source, and elsewhere in a 'birthdate' column based on a date data type. Once the data is "cleansed" it is transformed into a uniform representation in the data warehouse (typically a relational database).

To facilitate the analysis of historical trends, appropriate temporal data should be included, at the desired granularity (e.g., daily, weekly, or monthly). For example, suppose an operational data source stores the following fact type: Employee manages Store. Over time, a store may have different managers. To retain history of these changes, when store management facts are loaded into the data warehouse, the load date may be inserted into the key, to populate the historical fact type: Employee on Date managed Store. Basic aspects of temporal modeling were discussed in Section 10.3.

To improve query performance, data marts (and usually the data warehouse) often contain derived data and denormalized structures. As a simple example of derived data, suppose an operational source stores the fact type Customer was born on Date. For demographical analysis, we may be interested in how product preferences are influenced by the age of customers. In this case it may be more appropriate in a data mart to store the age, or even just the age group, of the customer rather than their birthdate, using a fact type such as Customer on Date belonged to AgeGroup. The snapshot dates are inserted when the fact type is updated incrementally.

A more typical example incorporating both derivation and denormalization is a data mart for analyzing Sales trends. An ORM conceptual model of such a mart (simplified) is shown in Figure 16.2. In this UoD, the company makes sales from many stores, located in various cities within the same country (e.g., the United States).

A city is identified by combining its name and state (e.g., Portland, Oregon differs from Portland, Maine). States are grouped into regions (e.g., Oregon and Washington belong to the Northwest region). Items sold have a unique code and title, and belong to a category (e.g., Developer Tools). A line item is identified by its line number on a given invoice, and records the sale of an item, as well as the quantity and unit price for that item. The line total for each line item is derived by multiplying the quantity by the unit price and is then stored. To support sales analysis over time, the month number, quarter number, and year for each date are derived and stored. Although calendar years are used here, we could use fiscal years instead or as well. In Figure 16.2, fact types that are derived and stored are marked "**", but for simplicity the derivation rules are omitted.

Figure 16.2 A conceptual schema for the Sales data mart (derivation rules omitted).

Using the default Rmap algorithm, this conceptual schema maps to the following six table schemes: *LineItem*(invoiceNr, lineNr, itemCode, quantity, unitPrice, lineTotal); *Invoice*(invoiceNr, saleDate, storeNr); *Item*(itemCode, itemTitle, category); *Store*(storeNr, stateCode, cityName); *StateLocation*(stateCode, region); *TimeDimension*(saleDate, saleYear, QuarterNr, MonthNr).

However, to improve query performance it is decided to denormalize the relational schema to four table schemes, as shown in Figure 16.3. The Item and TimeDimension tables are normalized, but the Sale and Store tables are not. The Sale table is denormalized to 1NF, since saleDate and storeNr are functionally dependent on invoiceNr, which is just part of the primary key. The Store table is denormalized to 2NF since region is functionally dependent on stateCode, a nonkey attribute.

The decision to denormalize in this way is indicated by annotating the conceptual schema, as shown in Figure 16.2. Here the *key object types* are *shaded*. An object type is a key object type if and only if its preferred identification scheme is used as the primary key of a table. Graphically, each key object type forms the root of a tree, where each node is an object type and each edge is a functional (n:1 or 1:1) predicate.

Figure 16.3 A denormalized, relational star schema for the model in Figure 16.2.

For example, from the Store object type we run down functional chains to the leaf object types CityName and Region, gathering all the fact types on the way to group them into a single table based on the identifier for Store.

A functional chain stops if it runs into a key object type (or a leaf or a nonfunctional predicate). For example, starting at LineItem we gather up all its functional fact types, as well as those for Invoice, but we cannot proceed past Date, Store, or Item, since these are key object types. This leads to the Sale table in Figure 16.3.

The denormalized Sale and Star tables contain embedded functional dependencies (e.g., stateCode → region), but there is no need to enforce these because they have already been enforced in the operational tables from which the data mart is derived. Since the operational tables are used for base updates, and not the data mart, it is acceptable to denormalize the data mart in this way. Reducing the number of tables eliminates the need for many joins, leading to faster queries.

The schema in Figure 16.3 is composed of a central table (Sale) linked by foreign key connections to outer tables (Item, Store, and TimeDimension). This is called a *star schema*, since the central table may be viewed as the center of a star pattern, with its outer tables becoming "points of the star". In data warehousing terminology, the *central table* is called a "*fact table*" and the outer tables are *dimension tables*. Since all tables contain facts, the term "fact table" is rather inappropriate here. Moreover, ORM uses the term "fact table" to mean an atomic fact table. To avoid confusion, we'll use the more descriptive "central table" instead of the more popular "fact table" in this context.

Some approaches require the primary key of the central table to include all the keys of its dimension tables. This would require the (invoiceNr, lineNr) key of the Sale table in Figure 16.3 to be expanded to the superkey (invoiceNr, lineNr, itemCode, saleDate, storeNr). From a purely logical standpoint this is not required, since joins can be made on nonkey attributes.

If some dimension tables are themselves used as central tables for other stars, the overall schema is called a "snowflake schema". A set of star schemas with shared dimension tables is sometimes called a "galaxy".

Data warehouses and data marts are used for *online analytical processing (OLAP)* and *data mining*. The term "OLAP" was introduced by Edgar Codd to describe interactive analysis of dimensioned and aggregated data for decision support (Codd et al. 1993). Data mining involves deeper analysis of the data, typically using sophisticated statistical techniques and complex algorithms for detecting patterns. Nowadays, many tools for OLAP and data mining are in use. Either topic deserves a book in itself. The remainder of this section provides a brief overview of OLAP.

There are three main approaches to OLAP. Each uses base data as well as aggregated data (e.g., sales figures might be summed and grouped at various levels). Multidimensional OLAP (*MOLAP*) stores both base and aggregated data in multidimensional structures rather than tables. Relational OLAP (*ROLAP*) stores both base and aggregated data in relational tables. Hybrid OLAP (*HOLAP*) stores base data in relational tables and aggregated data in multidimensional structures. Some DBMSs, such as Microsoft SQL Server, support all three kinds of OLAP.

The multidimensional structures used for OLAP are popularly known as *cubes*. In geometry, a cube is a three-dimensional box structure. In OLAP theory, a cube can have as many dimensions as you like. Cubes provide an intuitive way to visualize and browse data, as well as fast access to aggregate data. Let's consider a simple example.

With reference to the star schema in Figure 16.3, suppose we wanted to list the number of units sold in the years 2007 through 2009 for each geographic region and each item category. As an exercise you might like to formulate the SQL query for this. You need to perform the natural join of the four tables, group by saleYear, region and category, and compute sum(qty). An extract of a possible result is shown in Table 16.1, assuming only two categories (SW = Software, HW = Hardware) and four regions (N = North, S = South, E = East, W = West). The fact type underlying this table is the quaternary: Year in Region had sales of items of Category in NrUnits. The full table display of the result would include 24 rows (8 for each year); only the first 9 rows are shown here.

Table 16.1 Units sold.

Year	*Region*	*Category*	*NrUnits*
2007	N	SW	10000
2007	N	HW	500
2007	S	SW	12000
2007	S	HW	330
2007	E	SW	7500
2007	E	HW	440
2007	W	SW	12000
2007	W	HW	…350
2008	N	SW	14500
…	…	…	…

Figure 16.4 Cube depiction of units sold (only the software figures are shown here).

An alternative way of displaying the data using a cube is shown in Figure 16.4. Here the independent variables (year, region, and category) appear as the *dimensions* making up the edges of the cube, and the values for the dependent variable or *measure* (units sold) appear in the relevant cells. Only the software sales figures are shown here in the 12 cells making up the front half of the cube. You can imagine seeing the hardware sales figures by rotating the cube to see its other side.

OLAP cubes can be much more complex than this example. For any set of independent dimensions, there may be more than one measure (e.g., units sold, and revenue). These different measures are often collectively referred to as the *Measures dimension*. Moreover, each independent dimension typically has a *hierarchy* of *levels.* For example, a Location dimension might have regions at its top level, composed of states at the second level, with cities at the third level, and stores at the fourth level. Similarly a Time dimension may be decomposed into Years, then Quarters, then Months, and then Days. Finally, the Item dimension may be decomposed into categories and then items.

So our data mart example can be used to construct a cube with three independent, hierarchical dimensions (Location, Time, and Item) and one dependent dimension for UnitsSold and Revenue measures. The neat thing about the cube structure is that it enables aggregate values for the measures to be efficiently stored and accessed for all levels of the hierarchies. When analyzing a cube, you can choose to consolidate or *rollup* these aggregates (e.g., roll up sales figures for cities to regional sales figures). You can also do the opposite, *drilling down* to a finer level of granularity. Moreover you can *slice and dice* the cube whichever way you like by taking a subcube of it (e.g., if you restrict the item category in Figure 16.4 to software, you get the front slice of the cube)

Because of such advantages, star schemas in data marts are often used to create a variety of cubes for easy analysis. SQL Server provides wizards to simplify the task of cube creation and also supports a Multidimensional Expression (*MDX*) language for querying cubes.

For example, assume SalesCube is based on the hierarchical Location, Time and Item dimensions discussed earlier. The following MDX query will result in a two-dimensional grid that lists the software units sold and revenue in Washington State in each of the first three quarters of 2007.

```
select { [Measures].[UnitsSold], [Measures].[Revenue] } on columns,
       { [Time].[2007].[Q1] : [Time].[2007].[Q3] } on rows
from   SalesCube
where ( [Item].[SW], [Location].[N].[WA] )
```

In MDX, braces are used to delimit sequences, and square brackets may be used to delimit identifiers or values. The dot notation is used to move down a hierarchy. Notice that Time is drilled down to quarters and Location is drilled down to states. A colon ":" is used to indicate a range (like ".." in English). The result grid from the aforementioned query has two columns for units sold and revenue, and three rows for the first three quarters of 2007. A full coverage of MDX would require another chapter or two, so it is not pursued further here. An extensive treatment of MDX and OLAP for SQL Server is provided by Thomsen et al. (1999).

Different OLAP tools work differently. SQL:1999 onwards includes support for cubes and rollup, so it is hoped that some OLAP features will become more standardized in the future. MDX queries can be very complex, and Microsoft SQL Server provides a very high level language called *English Query* that can be used by end users to generate MDX queries. The topic of natural language-based queries is discussed in the next section.

16.3 Conceptual Query Languages

An information system may be modeled at any of four levels: conceptual, logical, physical, and external. In principle, it may also be *queried* at any of these levels. In practice however, although conceptual modeling is widely used, it is comparatively rare for systems to be queried at the conceptual level. Instead, queries are typically formulated either at the external level using forms, at the logical level using a language such as SQL, or at the physical level using a programming language. This section briefly indicates some problems with these lower level approaches and explains how these problems are avoided by conceptual query languages, especially those based on ORM.

At the external level, Query-By-Form (QBF) enables users to enter queries directly on a screen form by entering appropriate values or conditions in the form fields. This form-based interface is well suited to simple queries where the scope of the query is visible on a single form and no complex operations are involved. However, this cannot be used to express complicated queries. Moreover, saved QBF queries may rapidly become obsolete as the external interface evolves. For such reasons, QBF is too restrictive for serious work.

For relational databases, SQL and Query-By-Example (QBE) are more expressive. However, complex queries and even queries that are easy to express in natural language (e.g., who does not speak more than one language?) can be difficult for non-technical users to express in these languages. Moreover, an SQL or QBE query often needs to be changed if the relevant part of the conceptual schema or internal schema is changed, even if the meaning of the query is unaltered. Finally, relational query optimizers ignore many semantic optimization opportunities arising from knowledge of constraints.

Logical query languages for postrelational DBMS's (e.g., object oriented and object relational) suffer similar problems. Their additional structures (e.g., sets, arrays, bags, and lists) often lead to greater complexity in both user formulation and system optimization. For example, Object Query Language (OQL) extends SQL with various functions for navigation, as well as composing and flattening structures, thus forcing the user to deal directly with the way the information is stored internally (Cattell and Barry 2000). At the physical level, programming languages may be used to access the internal structures directly (e.g., using pointers and records), but this very low level approach to query formulation is totally unsuitable for end users.

Given the disadvantages of query formulation at the external, logical, or physical level, it is not surprising that many *conceptual query languages* have been proposed to allow users to formulate queries directly on the conceptual schema itself. Most of these language proposals are academic research projects, with at best prototype tool support, and are typically based on ER schemas (e.g., ERQL, Super, Hybris) or deductive models (e.g., CBQL). By and large, these are challenging for naïve users, and their use of attributes exposes their queries to instability, since attributes may evolve into entities or relationships as the application model evolves. References for these are included in the chapter notes.

Some commercial tools allow users to enter queries directly in English, and then translate these into database queries in a language such as SQL or MDX. Early tools of this type often suffered from problems with ambiguity and expressibility, as well as the correctness of the SQL code generated. Most of the natural language input tools we've looked at require you to spend considerable effort setting up a dictionary to relate conceptual terms to the underlying database structures, and most of them do an imperfect job at handling natural language input. This is not really surprising, given the inherent ambiguity and complexity of natural language.

One approach to address the ambiguity problem is taken by *English Query*, which was included some years ago in Microsoft SQL Server. When you enter a question in English, it's rephrased to indicate the interpretation taken. Sometimes more than one rephrasing is given, and you can pick the one that conveys your intended meaning.

There are of course limits on what kinds of questions can be understood, and you need to spend time setting up the connections between your semantic model and the database structures. For example, if you want to make a join between tables other than a foreign key to primary key reference, you need to explicitly add this beforehand. Also, the SQL that's generated might not be as efficient as you might construct yourself.

However, once you've done all this work, you can make it possible for nontechnical end users to query the database in plain English. Although common, predictable queries are best made available through prewritten procedures, English Query is especially useful for ad hoc queries. For example, with reference to the data mart schema in the previous section, and assuming adequate setting up, the question "Which region has the most stores?" might generate the following SQL code:

```
select top 1 with ties dbo.Store.region as "Region", count(*) as "count"
from   dbo.Store
where dbo.Store.region is not null
group by dbo.Store.region
order by 2 desc
```

In addition to returning the region(s) with the most stores, this query returns the number of stores in those regions, which we might not want. Moreover, the **where** clause and the **order by** clause are both redundant. But these are small issues for the nontechnical user who is incapable of formulating the SQL query. For OLAP applications, English Query can also be used to generate MDX queries. For example, referencing the SalesCube discussed in the previous section, the question "List the three items with the most unit sales" might result in the restatement "Show the three items with the highest total unit sales" and the following MDX query:

```
select { Measures.[UnitsSold] } on columns,
      topcount([Item].[ItemCode].members, 3, Measures.[UnitsSold] ) on rows
from   [SalesCube]
```

As these examples illustrate, free input in natural language has a lot of potential for opening up databases to anybody who can write. As language recognition technology matures, it will also open up databases to anybody who can speak. However, we still have a long way to go before this becomes a totally reliable technology. In particular, it can be very difficult to set up semantic dictionaries to always capture the intended interpretation of user questions.

These problems of ambiguity and onerous dictionary preparation can be eliminated in one stroke by taking a different approach: *conceptual-schema-based queries*. Here the user issues queries by selecting paths through the existing conceptual schema, rather than using free text input, so no ambiguity can arise. As discussed later, if an ORM conceptual schema is used, there is no need to predeclare any join paths, since ORM object types are the semantic domains on which all joins are based. Since ORM is attribute-free in its base conceptual model, it also avoids the instability problems of attribute-based queries, which require reformulation when attributes evolve into relationships (see later example). Moreover, ORM's sole data structure is the relationship type, providing a simple, sentence-based framework. For such reasons, we believe that ORM conceptual query technology has the greatest potential of any existing approach.

The first significant ORM-based query language was Reference and Idea Language (RIDL), a hybrid language with both declarative and procedural aspects (Meersman 1982a; Meersman et al. 1984). RIDL is a textual language for defining, populating,

and querying models in NIAM, an early version of fact-oriented modeling. It was developed at the Control Data research laboratory in the University of Brussels, and was the first truly conceptual query language ever implemented. Let's look at some RIDL examples. Consider a military conceptual schema with the following fact types: Officer has Rank; Officer was born in Year. In RIDL, these fact types would normally be declared as: Officer having Rank; Officer born-in Year. To find which sergeants were born before 1950, the following RIDL query may be used:

```
LIST Officer (having Rank 'sergeant' AND born-in Year < 1950)
```

In RIDL, reserved words are in uppercase. To illustrate the procedural side of RIDL, the following query lists pairs of officers of the same rank who were born before 1950:

```
FOR EACH o1, o2 IN Officer born-in Year < 1950 DO
  IF Rank OF o1 = Rank OF o2
     THEN LIST o1, o2
  END-IF
END-FOR
```

The original RIDL software mapped RIDL queries into internal statements that were directly executable on some early Control Data computers. RIDL was developed around the same time that SQL systems started to appear on the market. If you compare the aforementioned queries with equivalent SQL queries, you can see that RIDL was far ahead of its time. In later years, the data definition part of RIDL was implemented graphically in the RIDL* tool, but the query component was not supported. Another ORM query language is Language for Information Structure and Access Descriptions (LISA-D). Although very expressive, it is technically challenging for end users, and currently lacks tool support (ter Hofstede et al., 1996).

Like ORM, the Object-oriented Systems Modeling (OSM) approach avoids the use of attributes as a base construct. An academic prototype has been developed for the graphical query language OSM-QL based on this approach (Embley et al. 1996). For any given query, the user selects the relevant part of the conceptual schema, and then annotates the resulting subschema diagram with the relevant restrictions to formulate the query. Negation is handled by adding a frequency constraint of "0", and disjunction is introduced by means of a subtype-union operator. Projection is accomplished by clicking on the relevant object nodes and then on a mark-for-output button.

Another ORM query language is *ConQuer* (the name derives from "CONceptual QUERy"). ConQuer is more expressive than OSM-QL, easier for novice users. Its tool implementations transform conceptual queries into SQL queries for a variety of back-end DBMSs, and do not require the user to be familiar with the conceptual schema or the ORM diagram notation. The first version of ConQuer was released in InfoAssistant (Bloesch and Halpin 1996). Feedback from this release led to the redesign of both the language and the user interface for greater expressibility and usability, resulting in a tool called ActiveQuery (Bloesch and Halpin 1997). Typical queries can be constructed by just clicking on objects with the mouse and adding conditions. Owing to acquisitions, the ActiveQuery technology is now owned by Microsoft.

A basic understanding of the ConQuer technology will help provide insights into what a truly conceptual query environment can offer. The ConQuer language and its tool support were designed for expressibility, clarity, simplicity, semantic stability, and semantic relevance. These design principles were explained in Section 3.1. A ConQuer query can be applied only to an ORM schema. Using a software tool, an ORM schema may be entered directly, or instead reverse engineered from an existing logical schema (e.g., a relational or object-relational schema). While reverse engineering is automatic, some refinement by a human improves the readability (e.g., the default names generated for predicates are not always as natural as a human can supply).

Although ConQuer queries are based on ORM, users don't need to be familiar with ORM or its notation. A ConQuer query is set out in textual (outline) form (basically as a tree of predicates connecting objects) with the underlying constraints hidden, since they have no impact on the meaning of the query.

With ActiveQuery, a user can construct a query without any prior knowledge of the schema. On opening a model for browsing, the user is presented with an object pick list. When an object type is dragged to the query pane, another pane displays the roles played by that object in the model. The user drags over those relationships of interest. Clicking an object type within one of these relationships causes its roles to be displayed, and the user may drag over those of interest, and so on. In this way, users may quickly declare a query path through the information space, without any prior knowledge of the underlying data structures. Users may also initially drag across several object types. The structure of the underlying model is then used to automatically infer a reasonable path through the information space: this is called a *point-to-point query*.

Items to be displayed are indicated with a check mark "✓". These check marks may be toggled on/off as desired. The query path may be restricted in various ways by the use of operators and conditions. As a simple example, suppose a company has branch offices in several cities around the world. Now consider the query: List each employee who lives in the city that is the location of branch 52. This may be set out in ConQuer thus:

```
Q1      ✓Employee
            └─ lives in City
                    └─ is location of Branch 52
```

This implicit form of the query may be expanded to reveal the reference schemes (e.g., EmployeeNr, BranchNr), and an equals sign may be included before "52". Since any ConQuer query corresponds to a qualified path through an ORM schema, where all the object types and predicates are well defined, the meaning of the query is essentially transparent. ActiveQuery also generates an English verbalization of the query in case there is any doubt. This ensures *semantic clarity*.

Since ORM object types are semantic domains, they act as semantic "glue" to connect the schema. This facilitates not only strong typing but also query navigation through the information space, enabling joins to be visualized in the most natural way (recall Section 4.6). The underlying ORM schema fragment for query Q1 is shown in the shaded path within Figure 16.5, along with the identification scheme for City.

Figure 16.5 Only the shaded path is relevant to query Q1.

Notice how City is used as a join object type for this query. If attributes are used instead, the query formulation becomes more cumbersome. If composite attributes are allowed we might use **List** Employee.empNr **where** Employee.city = Branch.city **and** Branch.branchNr = 52. If not, we might resort to **List** Employee.empNr **where** Employee.cityName = Branch.cityName **and** Employee.stateCode = Branch.stateCode **and** Employee.country = Branch.country **and** Branch.branchNr = 52. Apart from awkwardness, both of these attribute-based approaches violate the principle of *semantic relevance*. Since the identification scheme of City is not relevant to the question, the user should not be forced to deal explicitly with it.

Even if we had a tool that allowed us to formulate queries directly in ER or OO models, and this tool displayed the attributes of the current object type for possible assimilation into the query (similar to the way ActiveQuery displays the roles of the highlighted object type), this would not expose immediate connections in the way that ORM does. For example, inspecting Employee.city does not tell us that there is some connection to Branch.city. At the relational level, we have two tables *Employee*(empNr, countryCode, stateCode, cityName, …) and *Branch*(branchNr, countryCode, stateCode, cityName, …). Since there are no foreign key references to relate the city attributes in these tables, free text query tools rely on the join connection being explicitly entered into the dictionary beforehand, which is a risky assumption. The only way to automatically reveal all join possibilities is to use the domains themselves as a basis for connectedness, and this is one of the distinguishing features of ORM.

Because ConQuer queries are based on ORM, they continue to produce the desired result as long as their meaning endures. In other words, you never need to change a ConQuer query if the English meaning of the question still applies. In particular, ConQuer queries are not impacted by typical changes to an application, such as the addition of new fact types or changes to constraints or the relative importance of some feature. This ensures *semantic independence* (i.e., the conceptual queries are independent of changes to underlying structures when those changes have no effect on meaning). This results in greater *semantic stability* than attribute-based approaches.

As a simple example, suppose that in our model each employee is identified by an employee number and has exactly one gender and at most one title. In the real world, an employee may have many titles (e.g., ‘Dr.’, ‘Prof.’, ‘Sir’), but we decide to record at most one of these.

Figure 16.6 Schemas and queries to list titled employees and their gender.

Now suppose we want to list the titled employees and their gender. An ORM schema and ConQuer query for this are shown in Figure 16.6(a). There is an implicit **and** between the two branches stemming from Employee. The rest of the figure shows an equivalent UML class diagram, relational schema, and SQL query. Not only is the SQL code subconceptual (nulls are an implementation detail) but it is unstable, since a simple change to a conceptual constraint on the title relationship requires the code to be changed as well.

For example, suppose we change our business rules to allow the recording of many titles for the same employee, and we also decide to record any required orders for listing titles (e.g., the combination 'Prof. Dr.' should be listed that way, not 'Dr. Prof.'). From an ORM viewpoint, this new situation is depicted in Figure 16.7(a). The uniqueness constraint on the title association is relaxed from *n*:1 to *m:n*, and a title precedence fact type is added. These schema changes have no impact on the ConQuer query, whose meaning relates just to the shaded fact types, not to their constraints and not to the additional fact type. Since the shaded path has unchanged semantics, the original ConQuer query means the same, so it may be reused.

Figure 16.7 ORM has greater semantic stability than attribute-based approaches.

For attribute-based approaches, the business change requires major changes to both the model and the query. The rest of Figure 16.7 shows the new UML schema, the new relational schema, and the new SQL query. Substantial changes are needed because facts about titles are now stored in an association rather than an attribute. Another case forcing facts modeled with title and even gender attributes to be remodeled with associations was discussed in Section 10.1, to record restrictions of some titles (e.g., 'Sir') to specific genders. In contrast, the ConQuer query stays valid; all that changes is the SQL code that gets automatically generated from the query.

An OO query approach is often more problematic than an ER or relational approach because of the many extra choices on how facts are grouped into structures, and the user is exposed to these structures in order to ask a question. Moreover these structures may change drastically to maintain performance as the business domain evolves.

In the real world, UoD changes often occur, requiring changes to the database structures. Even more work is required to modify the code for stored queries. If we are working at the logical level, the maintenance effort can be very significant. We can minimize the impact of change to both models and queries by working in ORM at the conceptual level and letting a tool look after the lower-level transformations.

The ActiveQuery tool generates *semantically optimized* SQL queries, wherever possible using knowledge of ORM constraints to produce more efficient code. For example, if the title role were actually mandatory, the SQL generated by the previous ConQuer query would, in both cases, simplify to **select** empNr, gender **from** Employee. The tool can also generate different code for different back ends to cater for their different support for the SQL standard, sometimes generating chains of queries over temporary tables to enable the query to run at all.

The simple examples just given illustrate how ConQuer achieves semantic clarity, relevance, and stability. Let's look briefly at its semantic strength or expressibility. The language supports the usual comparators (=, <, **in**, **like**, etc.), logical operators (**and**, **or**, **not**), and bag functions (**count**, **sum**, etc.), as well as a modal operator (**possibly**) for conceptual left outer joins. Subtype/supertype connections appear as "**is**" predicates.

For example, assuming that Manager is a subtype of Employee in the ORM model, the following query Q2 asks: List each manager and their cars (if any) where the managers do not work at branch 52.

```
Q2    ✓Manager
          └─ is Employee
                  ├─ possibly drives ✓Car
                  └─ not works at Branch 52
```

The language is fully orthogonal and supports arbitrary correlation, using subscripted variables (e.g., Person_1, Person_2) when necessary to distinguish different instances of the same object type. As a simple correlation example, consider the query "Who supervises an employee who lives in the same city as the supervisor but was born in a different country from the supervisor?" Assuming birth and supervision fact types in the ORM model, query Q3 shows one way to express this in ConQuer.

```
Q3    ✓Employee1
        ├─ lives in City1
        ├─ was born in Country1
        └─ supervises Employee2
                    ├─ lives in City1
                    └─ was born in Country2 <> Country1
```

When an object type appears more than once, ActiveQuery automatically appends subscripts to distinguish the occurrences. You can equate instances by equating their subscripts (e.g., $City_1$). More generally, you can use comparators to compare instances (e.g., $Country_2 <> Country_1$). Try this in SQL. It's not that hard, but you have to admit it's easier in ConQuer!

As a final example, suppose the ORM model includes the *m:n* fact type Employee has Rating. Now consider the query "List each branch and those of its employees whose maximum individual rating exceeds the average of his/her branch" (remember that an employee may have many ratings). Query Q4 shows how to do this in ConQuer, illustrating the use of bag functions in for-clauses.

```
Q4    ✓Branch
        └─ employs ✓Employee
                    └─ achieves Rating
                                └─ max(Rating) for Employee >
                                   avg(Rating) for Branch
```

At the relational level, two tables are needed: *Employee*(empNr, branchNr, …); Achieves(empNr, rating). The SQL for this query is shown here. This is tricky, especially the final correlation between branch numbers, and even experienced SQL programmers might have difficulty with it. The equivalent ConQuer query would be easy for most end users who have even minimal experience with the language. This gives some indication of the potential of a language such as ConQuer for empowering end users.

```
select max(X1.branchNr), X2.empNr
from   Employee as X1, Achieves as X2
where X1.empNr = X2.empNr
group by X2.empNr
having max(X2.rating) >
   (select avg(X4.rating)
    from  Employee as X3, Achieves as X4
    where X3.empNr = X4.empNr
      and X3.branchNr = X1.branchNr)
```

ActiveQuery allows you to define derived predicates, and store these definitions. These derived predicates (or "macros") can then be used just like base predicates in other queries. A subtype may be thought of as a derived object type, with its definition provided by a ConQuer query. For example, the population of the subtype MalePatient can be obtained by executing the following ConQuery query: **Find each** Patient **who** is of Gender 'M'. Given its generality, ConQuer could be adapted to provide a very high language for specifying derivation rules.

A constraint may be viewed as a check that a query searching for a violation of the constraint returns the null set. Hence constraints may also be expressed in terms of queries. Various high level constructs can be provided in the language to make it more natural than the not-exists-check-query form provided in SQL. Although there is no room here to go into detail, it should be clear that this approach is quite powerful.

ORM's rich graphical constraint notation still needs to be supplemented by a textual language to provide a complete coverage of the kinds of constraints and derivation rules found in business domains. While ORM's graphical constraints can be verbalized automatically, input textual languages such as ConQuer could be used to define other business rules mapped automatically to SQL. Although the ActiveQuery tool is currently no longer available, work on the NORMA tool is under way to extend the FORML textual language for ORM with the expressive power of ConQuer. This should result in a very high level textual language for both conceptual modeling (including fact types, constraints and derivation rules) and conceptual queries.

16.4 Schema Abstraction Mechanisms

At times we might feel almost swamped by the level of detail captured on an ORM schema diagram, or even on a UML, ER, or relational schema. Nevertheless, such detail is important when developing, transforming, or mapping schemas. The ORM notation has been crafted to facilitate these tasks. As we've seen, the diagrams may be verbalized naturally and populated with fact instances, their object-role-based notation allows many constraints to be expressed intuitively, and their object types reveal the semantic domains that glue the schema together. All of this helps the modeler to get a complete and correct picture, and to transform the model in a rigorous way with formal control of information loss or gain.

However, once a schema has been developed, we may at times wish to hide some of the information in order to obtain a quick overview or to focus on some aspects. This is particularly the case if the schema is large and complex. Hence there is a need for *abstraction mechanisms*, by which unwanted details may be removed from immediate consideration. This section outlines a few of the more useful ways of doing this. The chapter notes provide references for further study in this regard.

One obvious abstraction strategy is *modularization*. Here the complete schema is "divided up" into a number of conveniently sized modules or subschemas. One way to do this is to use different schemas for different modeling *perspectives*. For example, UML uses class and object diagrams for a data view, state charts and activity diagrams for behavioral views, and so on. In addition you might use a purely conceptual schema for analysis and annotated or modified conceptual schemas for a design view.

Within a given modeling perspective, the global schema is still likely to be too large to conveniently inspect in one go, even though wall-sized schemas can be very useful at times for seeing far-reaching connections. One trivial way to divide a global schema diagram is to overlay a grid to partition the schema into separate cells.

However, it is typically much more useful to allow modules to overlap so that, for example, the same object type might appear in more than one module. This technique also allows greater flexibility in basing modules on semantic groupings. One useful technique is to divide the global schema into *reusable subschemas*, which can be imported into other schemas or projects at a later time. The most popular technique of all is to divide a given schema into *pages*. Each page should be given a meaningful name and include a reasonable number of tightly coupled modeling elements.

Various means may be used to specify connections between modules, such as hyperlinks, annotations, and border overlaps (cf. a directory of road maps). Electronic browsing opens up greater possibilities than hard-copy browsing (e.g., scrolling, zooming in and out, hypertext navigation), but printed documentation on standard pages needs to be catered for as well.

With large applications, the original schema might be developed by different teams as separate modules that are later integrated. Additional care is then required for the *schema integration* process. In particular, global consistency must be ensured, either by agreeing on a uniform treatment of terminology, constraints, and rules or by specifying appropriate translations. Ideally, a basic architectural framework will be agreed upon early in the development, and all the modelers will attach the same meaning to words by resolving any synonym and homonym problems.

For example, if the same object type is called "Subject" in one module and "Course" in another, then one of these will be chosen as the standard term and the other term replaced by it. If different object types or fact types in different modules are given the same name or reading, one of these must be changed. As long as they uniquely refer, alternative names and readings may be stored as aliases.

Global identification schemes and constraints should be agreed upon, and where necessary contextual identification may be supported via subtyping, and textual constraints can be added to strengthen global constraints for a given context (e.g., restricted mandatory role and restricted uniqueness constraints). For some applications (e.g., federated databases), partial integration between relevant modules may be favored instead of global integration.

Another useful abstraction mechanism is provided by *display toggles,* or *layering*. For instance, the display of role names and selected classes of constraint or derivation rule may be toggled off when they are not of immediate interest in order to obtain a simpler picture. On an ORM schema for example, we might toggle off value lists, ring constraints, frequency constraints, set-comparison constraints, textual constraints, and derivation rules in order to focus on uniqueness and mandatory role constraints. Each constraint class may be thought of as a constraint layer, with the modeler choosing which layers to view at any given time. An extreme option would be to toggle off all constraints.

Subtype display can be suppressed in many ways. We can hide the subtype defining rules or graphic constraints (e.g., exclusion and totality). We can hide just some of the subtypes, preferably indicating somehow that others exist (e.g., with an ellipsis or one of the UML annotations discussed in Chapter 9). We can also collapse subtypes into their top supertype. Depending on the model type, various other modeling elements

can be hidden (e.g., reference schemes or attributes). When additional layers of detail are needed, their display can be toggled on again. Although this layering concept can be adapted to hard copy (e.g., by using a series of overlaid transparencies or progressively detailed printouts), it should ideally be exploited interactively with a CASE tool.

Yet another abstraction mechanism is *object-type zoom*. Here the modeler selects an object type of interest in order to have the display focused on that object type and its immediate *neighborhood*. The object type is displayed with all its fact types. By specifying a *logical radius* for the zoom (e.g., in predicate units), the neighborhood may be expanded to include fact types of the object types in the first level zoom, and so on. ActiveQuery uses this mechanism to progressively open up the universe to the user, displaying a neighborhood of one-predicate radius for the currently highlighted object type. The NORMA tool also provides a context window for this purpose, allowing expansion up to three predicate lengths.

A simple example will help illustrate some of the basic ways of abstracting. Figure 16.8 depicts an ORM schema for a small application. Here movies are identified by numbers, although the combination of their title and director is also unique. Some movies are based on another. For example, the western *The Magnificent Seven* was based on an early Japanese movie *The Seven Samurai*.

When available, figures about the gross revenues of a movie are recorded, and the net profit too if this is known. The only people of interest in this application are movie directors and movie stars. We record their country of birth and, if known, their birthdate. The country in which a movie was made is noted, as well as any export countries for it.

As a mental exercise, imagine toggling off the display of the constraint types on Figure 16.8 one at a time (ring, then set-comparison, then mandatory role, then uniqueness) to visualize abstraction by peeling off constraint layers.

Figure 16.8 A small, but detailed ORM schema.

Figure 16.9 Zooming in on the Person object type.

Figure 16.9 depicts a zoom on the object type Person, with a logical radius of 1, so that only the fact types directly connected to Person are shown. To highlight the zoom object type, a thicker line is used for it. Visualize for yourself a zoom on Country; this would display three fact types, one of which is also contained in the Person zoom.

Another abstraction mechanism that is especially useful with large ORM schemas is that of *refinement levels* based on *major object types*. This can be used in conjunction with *attribute abstraction* to provide an ER schema or UML class diagram.

When developing a conceptual schema it is useful to treat all object types equally, since the relative importance of an object type is based on all the parts it plays in the global schema. Once the schema is complete, however, this information is available and can be used to determine which are the most important, or *major object types*. A procedure for deciding the major object types on ORM schemas was developed by Campbell and Halpin (1993) and was used as a basis for generating default screen forms for the external user interface (since application screens basically enable users to perform operations on major objects).

Once the major object types are decided, an ORM diagram can be lifted to a higher level of abstraction. For example, we may choose to display only the *major fact types* (i.e., those fact types in which at least two roles are played by a major object type). With large schemas, this diagram itself may be subjected to the same procedure, yielding a higher level abstraction, and so on, until the top level view is obtained. As long as the fully detailed schema is accessible, this bottom-up abstraction may be reversed, allowing the top level view to be successively refined down to the detailed bottom level.

The detailed schema diagram in Figure 16.8 has six object types: Movie, MovieTitle, Person, MoneyAmount, Country, and Date. Intuitively, these are not all equally important in this domain. Which do you consider to be the major object types? Apart from using your intuition, you can make use of the constraint patterns to help decide. In particular, mandatory role and uniqueness constraints are relevant. If an object type has an *explicit* mandatory role, it is major. Usually, an object type playing a functional role is also major, but this is not always true (e.g., MovieTitle would still be a minor object type even if its role was functional; its role is still only implicitly mandatory).

Figure 16.10 Only the major fact types are shown.

There are some finer points to determining major object types, but this is enough to get the basic idea. Based on the previous reasoning, Figure 16.8 has only two major object types: Movie, and Person. Figure 16.10 provides one overview of the application by displaying only the major fact types.

Instead of being hidden, minor fact types may, if desired, be viewed in terms of *attributes* of the major object types. Although this attribute viewpoint hides structural information (e.g., domains and certain constraints), the ER or UML diagram obtained provides a compact picture that is useful for quick overviews. Figure 16.11 shows a UML class diagram for the current example. Unlike most ER notations, UML allows the attribute domains to be specified if desired, and the missing constraints (here disjunctive mandatory, subset, exclusion, external uniqueness, and asymmetric constraints) could be documented in notes, although at the price of losing compactness.

Notice that country information is displayed here in terms of attributes (single and multivalued) rather than as associations with a Country class. This is because Country is a minor object type in the original ORM schema. If instead Country played an explicit mandatory role, it would become major, and we would display it as a class. This is usually the case (e.g., we normally store both the code and name of countries).

Attribute-based diagrams can be made more compact by hiding all their attributes. For example, Figure 16.12 shows just the major fact types in both UML and Barker ER notations. Again, some ORM constraints are missing (inclusive-or and asymmetric constraints). By using ORM to actually generate such diagrams, or detailed UML and ER diagrams, we can retain the benefits of ORM (e.g., for validation and completeness) and still have compact views when desired.

Figure 16.11 A UML class diagram abstracted from Figure 16.8.

Figure 16.12 Major fact types displayed in (a) UML and (b) Barker ER.

16.5 Further Design Aspects

We have seen how a data model may be specified at the conceptual level (e.g., with an ORM schema) and the logical level (e.g., with a relational schema). The overall information systems life cycle includes other tasks, as discussed in Chapter 2. Chapter 15 discussed how to specify business processes using UML and BPMN. This section provides a brief overview of other design tasks such as relating processes to data, .organizing large models, and specifying the user interface.

Suppose we are asked to design an information system for an automated teller machine (ATM) to be operated by a credit union. As part of the requirements analysis phase, we determine that four main functions of the system are to enable clients to make deposits, withdrawals, and transfers (e.g., from a savings account to a checking account) and obtain printouts of the account balances. In UML, each of these functions may be treated as a use case and included in a use case diagram to provide an overview of the processes supported by the system. We might then flesh out these functions in a set of activity diagrams in UML or workflow diagrams in BPMN.

Suppose that when a client inserts a credit card into the ATM, the system checks whether the card is one of its own and, if so, accesses the client's identifier (client number) from the card. The same account could be used by many clients (e.g., a joint account for husband and wife), and each client has a distinct client number. In this way, the system can track which client actually makes any given transaction.

Suppose in diagramming the ValidateClient and GetBalance processes, we see that the combined process of validating a client and printing a requested balance involves two input data flows (clientNr, accountType) from the Client and one output data flow (balance) to the client. If we consider the credit union's account database system to be separate from the ATM, it may be thought of as another actor that actually provides the balance details in response to a request (e.g., an SQL query) sent from the ATM.

In such high level views, little or no detail is provided on the structure of the data store or on how the process manages to find the relevant data. Since such details are so far undefined, process diagrams often fail to provide an adequate basis for modeling the data perspective. Despite such informality, they can be of use in clarifying the UoD and for identifying functions to be included in the application's screen menus. Indeed,

ordinary cartoons with intuitive icons can be quite useful for communication between modeler and domain expert in the early stages of requirements analysis.

In principle, a process model could be successively refined to elementary transactions that expose the underlying fact types. However, this approach is simply too time-consuming for data intensive systems and is often too unstable, since business processes tend to change far more rapidly than the underlying data. To expedite the data modeling process we should seed it with data use cases and apply a modeling procedure such as the ORM CSDP. If examples already exist, use these. If they don't, sit down with the domain expert and generate them. These could be in the form of tables, forms, graphs, or whatever. By now, you should have the verbalization skills to do CSDP step 1 properly on any of these varieties, and since clients find it easier to work at the instance level, this is the safest way to go. Moreover, by doing this you've already started prototyping external screens and reports the clients want.

Process models should be formally connected to the data models. This is best done after the conceptual and logical data models are determined. Although skilled modelers can refine process models into data models, this approach tends to be error-prone. For example, suppose we bypass conceptual data modeling, instead refining the process models mentioned earlier for validating clients and getting balances into the following table scheme and SQL procedure. Here "&1" and "&2" are input parameters or placeholders. When the procedure is executed, actual values are input to these parameters.

D1: *Account* (clientNr, accountType, balance)

```
P1:  select balance from Account
     where clientNr = &1
       and accountType = &2
```

Can you spot a potential problem with this design? It's easiest to see the problem if you do a conceptual schema for D1. This is depicted in Figure 16.13. Each account has one or more users and has exactly one account type (e.g., L = loans, C = checking, S1 = main savings, S2 = other savings). That's fine. The problem lies in the identification scheme for Account. Although the combination of clientNr and accountType does identify an account in this domain, this should not be used for preferred reference, since (clientNr, accountType) pairs do not relate to accounts in a 1:1 fashion.

Figure 16.13 The preferred reference scheme for accounts is unsafe.

D1': *Account* (accountNr, accountType, balance)

AccountUse (accountNr, clientNr)

```
P1':  select balance
      from   Account inner join AccountUse
        on   Account.accountNr = AccountUse.accountNr
             and clientNr = &1 and accountType = &2
```

Figure 16.14 The preferred reference scheme for accounts is now safe.

For example, suppose your spouse's clientNr is 1001, yours is 1002, and you share a main savings account (type = 'S1'). This joint account may now be referenced by (1001, 'S1') and by (1002, 'S1'). This will not do for preferred reference. For instance, the Account table used for D1 will accept the population {(1001, 'S1', 9000), (1002, 'S1', 0)}. But this means that the same account has two different balances, which is nonsense. The obvious solution is to introduce another preferred identifier (e.g., accountNr), as shown in Figure 16.14. This figure also includes the new relational schema (D1') and the new procedure for computing the balance (P1').

This example is simplified (e.g., Exercise 6.4 discussed how balances might be derived from account transactions) but it does illustrate how conceptual modeling facilitates a correct data design, and that a detailed process specification is of little use unless the data structure on which the process operates is defined accurately.

In practice, the use of accountNr for preferred reference should be obvious by inspecting other data examples associated with this use case or a related one for the accounting system (e.g., a balance statement). As discussed in Section 5.3, a reference scheme within a given context might not be suitable for the global application. This is something to bear in mind when discussing data use cases with domain experts.

Although a rigorous process model is best built on top of a data model, an overview of the processes can be useful as a precursor to the data modeling, especially if the application is large or only vaguely understood. It is often helpful to get a clear picture of the functions of the application first.

Starting with the overall objectives, the main functions and processes of the application may be specified systematically using a variety of diagrams. This kind of analysis helps us divide the domain into coherent modules of manageable size and facilitates the task of specifying information examples where none exist. At this point, CSDP step 1 may be applied to each module.

Figure 16.15 A function tree for a hospital information system.

Functional requirements may be laid out in top-down form using a *function tree*. For example, Figure 16.15 lists functions of a hospital information system. The four main functions are numbered 1..4. Each of these is refined or decomposed into a number of lower level functions, numbered 1.1, and so on.

For example, function 3 is refined to functions 3.1 and 3.2. This may be refined further. For instance, function 3.1 is refined to functions 3.1.1, 3.1.2, and 3.1.3. To help fit the tree on a single sheet, this third level has been laid out vertically rather than horizontally. Tools such as the Org-Chart solution in Microsoft Visio allow a very flexible layout of such diagrams. The other three main functions may be refined in a similar way. Other kinds of diagram may be used to show dependencies between functions, which can help in deciding how to group functions into processes.

One common way of specifying information flows between processes is by means of *data flow diagrams*. Although invented long before UML use case diagrams, which have a somewhat similar purpose, data flow diagrams tend to be more useful in practice (Jeyaraj and Sauter 2007) and are still popular.

By decomposing the processes, a data flow diagram at one level may be refined into several lower level diagrams, and so on. As a precursor to this refinement, we might specify a very high level view in the form of a *context diagram*.

Figure 16.16 A context diagram for the pharmacy subsystem.

A context diagram provides an overview of the interaction between the information system and the environment, treats the system as a single process, and ignores any data stores. A context diagram for the pharmacy module of our hospital application is shown in Figure 16.16. External agents appear as shadowed boxes. Material flows are not shown, but some data flows may be associated with material flows (e.g., drug supply details).

At a lower level of detail, Figure 16.17 shows a data flow diagram for a fragment of the pharmacy system. This could be further refined (e.g., decompose the order placement process into processes for computing totals, printing order forms, etc.).

Figure 16.17 A data flow diagram for part of the pharmacy subsystem.

However, since such refinements are time-consuming, somewhat arbitrary, and lack the safeguards of population checks, there is usually little point in specifying low level data flows.

In general, then, once the requirements of the application are clear and modules of manageable size are identified, modeling efforts are best focused on obtaining a set of information examples and applying the CSDP. Once the conceptual schema is specified, the logical schema is obtained by applying a mapping procedure, such as Rmap.

The external schemas may now be developed for the various categories of user. It is through the interaction of users with the external interface that the original functions specified in the requirements analysis phase are ultimately executed. Hence it is important to design the external interface in a way that naturally reflects these functions.

Fundamentally, all that happens when users interact with an information system is that elementary facts are added, deleted, retrieved, or derived. In principle, we could specify the process perspective in terms of these two update and two query operations. But this is too fine a level of granularity for humans to work with efficiently.

We need a way of grouping these elementary operations into larger units. It is often natural to think of operations being performed on significant *objects*, such as patients and stock items. The major object types abstracted from an ORM schema (see previous section) correspond closely to these object types, and hence provide a good basis for defining operations. The more important classes or entity types in UML class diagrams or ER diagrams provide another good choice for this.

A generic set of external update operations may now be generated with three varieties: add object, delete object, and modify object. Some generic query operations may be generated (e.g., list all facts about an object), but specific queries require an appropriate selection of the relevant fact types (stored or derived).

Users perform such operations through an *external interface*, typically *screen* forms or menus. For each screen, we need to consider c*ontent* (which fact types and operations to include), *format* (how the screen is displayed), and *use* (how the operations are invoked by the user). As discussed, default decisions about screen content may be based on operations on the major object types. For example, if Patient is a major object type for the hospital system, a screen is required for it with at least the operations: add Patient, modify Patient, delete Patient, show Patient.

Unlike conceptual and logical tables, forms may overlap in their content. This redundancy at the external level conveniently allows users to perform an operation whenever relevant, without having to switch screens. For example, consider the *m:n* fact type Doctor treated Patient. When working with a Doctor screen, we might query which patients were treated by that doctor. When at a Patient screen, we might query which doctors treated that patient. In the conceptual and logical schemas, facts of this type are stored only once. As long as the external screens have been properly related to the underlying data structures, this overlap is safe. For a discussion on the use of ORM schemas to generate screen forms and transactions, see Campbell and Halpin (1993).

Although there is only one (global) conceptual schema for an application, there may be many external schemas. Different user groups might have different *authorization* levels that can be specified in *access rights tables*. This may be done at the con-

ceptual level (e.g., for each fact type, which groups have read, add, delete rights) or at the logical level (e.g., for each table or column, which groups have read, insert, delete, update rights).

Regarding access to options on screen forms, it is generally best to display to users only those options that they are authorized to select. As usual, access rights to screen options may be soft coded as data in tables rather than hard coded in procedures. This avoids having to recompile the forms when access rights are changed, and simplifies reuse of common code. As part of the overall security, passwords are required, and users are given the power to change their passwords.

Apart from access rights, users might also differ in their *ability* level. Designing application screens to optimize the way various users can perform their tasks is an important aspect of *human–computer interface (HCI) design*. Many kinds of interface can be used. For example, dialog with the user may use natural language, short questions, command lines, menus, forms, tables, diagrams, images, or sound. Input devices include keyboard, mouse, joystick, pen, and microphone.

In general the user interface should be *easy to learn* and *easy to use*. To achieve this, the interface should be consistent, simple, structured, efficient, and adapted to the user. Let's examine each of these in turn. *Consistency* of interface implies that all the components use the same basic conventions for display format and operations. This promotes positive transfer (learning how to use one screen helps you with the others) and avoids negative transfer. Consistency may be achieved at different levels: within the one screen, within the application, among applications from the same company, with external standards. For example, many Windows applications are being developed nowadays in conformity with Microsoft's user interface guidelines.

Simplicity of interface implies it is intuitive to use, with minimal effort for users. Each screen should provide the right amount of relevant information (e.g., limit the number of menu options seen at once). Don't clutter the screen. To allow many operations directly from the one screen, remove all but the main options from the permanent screen display, allowing the user to toggle extra options on as required. For example, use pull-down menus, pop-up windows (e.g., for look-up codes), and status-line help (and pop-up detailed help). Facilitate navigation within and between screens (e.g., use arrow keys for movement). Allow windows and menus to be cascaded or tiled.

Efficiency of interface entails that options may be selected quickly and intuitively. A *graphical user interface* (*GUI*) provides a more intuitive means of operation, especially where graphics or sophisticated layouts are involved: here it is important that "what you see is what you get" (WYSIWYG). However, key-based shortcuts (e.g., hot keys) are sometimes faster and should be provided for the experienced user.

Menu design has a major impact on efficiency. Menus come in various shapes and sizes (full-screen, pull-down, pop-up, ring, button bars, etc.). They provide most, if not all, of the application's operations. Options on the same menu should be semantically related. Navigation between menus or screens requires planning (state charts can help here) especially for web-based applications that use hyperlinks to navigate from one place to another. Some navigation patterns are shown in Figure 16.18. Hyperlink access is typically cyclic.

Figure 16.18 Some ways of navigating between menus or screens.

In addition, the following principles should be observed. Order menu options in decreasing frequency of use. Vary color and fonts, but don't overdo it. For options, use either intuitive icons or terse, clear textual descriptions that are mostly in lowercase. Allow selection of options by mouse clicks or hot keys, and support undo.

Adapt the interface to the user as follows. Show only what is relevant to the user. Provide simple procedures for casual users and shortcuts for experienced users. Let users configure the environment to their own taste. Provide levels of context-sensitive help: short messages on status lines, detailed help in pop-up windows. Trap errors as soon as possible and give short and long error messages. Provide online tutorials, user manuals, technical support, and training courses.

If you use some software packages that you find particularly intuitive, have a close look at the interface design and see if you can adapt its good features to your own (as long as you don't violate any copyright in doing so!). Once you develop your prototype interface, test it on typical users and use their feedback to finesse it.

That's a quick run through several topics that could easily fill a book on their own. Indeed, whole books are devoted to interface design; many refinements exist for modeling processes, including some very technical approaches that attempt to formally integrate these models with data models. Some interesting work in this regard is referenced in the chapter notes. Various methods and tools are also in common use for associated tasks such as project management (e.g., Gantt charts and PERT charts), testing, performance measures, documentation, and versioning.

16.6 Ontologies and the Semantic Web

Etymologically, the term "ontology" means the study of existence. In philosophy, ontology is the branch of metaphysics concerned with the fundamental nature of being, addressing deep questions such as "Do nonphysical things exist?", "Does an object remain identical to itself when it undergoes change", and so on. The informatics community later adopted the term "ontology", typically using it to mean a conceptual model about some business domain, where the model is designed to facilitate sharing information about that domain by conforming to some standard set of constructs.

With the rise of the Internet, millions of documents are now readily accessible over the World Wide Web. However, these documents were created by many different people, typically with little or no thought as to sharing or combining their information with other documents. In 2001, Sir Tim Berners-Lee, the founder of the Internet, proposed a new vision for a *semantic web* as extension of the current web "in which information is given well-defined meaning, better enabling computers and people to work in cooperation" (Berners-Lee et al. 2001). The main idea is to add global identifiers and structure to the documents (e.g., using uniform resource identifers and embedded tags) to reveal the underlying semantics of what is to be shared, in a way that is accessible to automated agents.

In 1999, the W3C produced the first version of the *Resource Description Framework* (RDF) language as a standard on top of XML to capture metadata for web resources (e.g., who authored a web document). Other work on adding automatically accessible meaning to language via markup tags was conducted in the United States by the Defense Advanced Research Products Agency (DARPA), which produced the DARPA Agent Markup Language (DAML). With similar objectives, researchers in the European Union (EU) developed the Ontology Inference Layer (OIL). In 2001, joint work between the United States and the EU incorporated many concepts from OIL into DAML, resulting in the DAML+OIL language, which was submitted in late 2002 to the World Wide Web Consortium (W3C).

In parallel with this effort, RDF was extended by a simple ontology language, *RDF Schema* (RDFS), which was adopted in 2004 by the W3C. In the same year, the W3C formally recommended the *Web Ontology Language* (OWL), which incorporates many aspects from DAML+OIL and RDFS, has a cleaner formal semantics based mainly on description logics, and is currently the most popular language for developing ontologies for the semantic web.

Figure 16.19 summarizes the typical dependencies between layers, where upper layers depend on lower layers. We now look briefly at some of the main concepts underlying RDF, RDFS, and OWL. Detailed coverage of these topics is accessible online using the links in the chapter notes.

RDF models are basically directed graphs, where each node is either a resource or a literal, and each directed arc represents a binary relationship between the nodes. A resource is anything that can be identified by a *Uniform Resource Identifier* (URI). A URI is a *Uniform Resource Locator* (URL) or a *Uniform Resource Name* (URN).

Figure 16.19 Foundational layers for the semantic web.

A URL (e.g., http://www.orm.net) includes a network access type (e.g., http for hypertext transfer protocol) plus a network address to locate the resource. A URN (e.g., urn:isbn:1-55860-672-6) includes a name space (e.g., ISBN for International Standard Book Number) plus a name that identifies that resource within that name space. Unlike a URL, a URN does not provide a way to locate the resource on the web. Both URLs and URNs are global identifiers, so when used in different documents they refer to the same resource.

To be more precise, RDF identifies resources by *URI references*. A URI reference (or URIref) is a URI, optionally followed by a fragment identifier. For example, the URI reference http://www.w3.org/TR/rdf-primer/#basicconcepts consists of the URI http://www.w3.org/TR/rdf-primer/ and (separated by the "#" character) the fragment identifier basicconcepts. The character strings used for URIrefs are composed of Unicode characters.

Figure 16.20 shows a simple RDF model depicted as a graph. Here we show resource nodes as named, soft rectangles; literals as named, hard rectangles; and relationships as named arrows from the subject node to the object node. The graph is intended to convey the semantics that there is a book identified by the ISBN 1-55860-672-6 that is authored by a person named “Terry Halpin”.

RDF captures the information as a set of simple *statements*, each of which may be represented as a *triple* (subject, predicate, object). The subject and predicate are resources, and hence are identified by a URIref. The object is either a resource or a literal. Literals may be untyped (as here) or typed (by pairing the value with a data type, that itself is identified by a URIref to a standard XML Schema data type). Each predicate (also known as a property of the resource subject) includes a predicate reading (e.g., “authored_by”, “has_name”, “type”), but since the same predicate reading may be used to mean different things in different contexts, a full URIref is required to assign a unique meaning to a predicate.

The resource nodes on the left of Figure 16.20 denote an individual book and an individual person. However, the resource nodes on the right denote the types Book and Person. The type predicate relates an instance to a type, and means “is an instance of”.

Figure 16.20 A simple RDF model.

RDF also allows descriptions of resource collections (e.g., sets and lists), and RDF graphs can be serialized into XML format using RDF/XML, but we ignore these feature here.

As the example in Figure 16.20 shows, RDF graphs allow nodes that are instances or types, and allow arcs that represent relationships between instances, or relationships between instances and types. RDF also allows relationships between types and types (e.g., Woman is_a Person). RDF even allows you to state that a type is an instance of itself (e.g., Class is_of_type Class), which can lead to formal problems such as Russell's paradox. Moreover, no special semantics is assigned to any predicate, so the is_an_instance_of relationship and the is_a_subtype_of relationships are treated the same as the has_name relationship (apart from different strings used for the URIrefs). So there is no ability to perform inferences based on knowledge of special properties for certain predicates (e.g., transitivity of is_a_subtype_of).

RDFS builds on RDF by providing inbuilt support for classes and subclassing. For example, resources can be typed as classes (using rdfs:Class) and the predefined predicate rdfs:subClassOf is defined to be transitive. For a given predicate, the set of instances playing the subject and object roles are called the *domain* and *range*, respectively. RDFS allows predicates to be typed by restricting their domain (using rdfs:domain) and range (using rdfs:range), respectively. Membership in a collection can also be specified using the rdfs:member property.

Despite these improvements, RDFS is still both too inexpressive (e.g., it cannot constrain associations to be other than *m:n*, and it cannot express complex properties by conjoining existing properties) and too expressive (e.g., like RDF it allows you to state that Class is an instance of itself, and its underlying logic is undecidable).

Like RDFS, OWL is restricted to binary fact types (no *n*-aries or unaries). OWL has a simpler formal semantics than RDFS, disallowing some freedoms in RDFS, while at the same time adding the ability to capture further constraints. OWL is used to specify the ontology schema (called a terminological box or *Tbox*) for a business domain, and RDF/XML is used to mark up conforming instance data (the fact population is called an assertional box or *Abox*). Since information on the web is often incomplete, OWL adopts the open world assumption (the failure to find or infer some proposition does not imply that proposition is false). The OWL specification has three dialects. In increasing order of expressibility, these are OWL Lite, OWL DL, and OWL Full.

OWL Lite is designed for simple ontologies composed mainly of classification hierarchies and relationships with simple constraints. Based on the description logic $\mathcal{SHIF(D)}$, it has low formal complexity and is decidable (an algorithm exists to determine whether any given formula in that logic is a logical truth, hence all its computations are guaranteed to be completed in a finite time).

By default, all relationship types are optional and *m:n*. OWL Lite allows fact roles to be declared mandatory and fact types to specified as *n*:1, 1:*n*, or 1:1 using minCardinality and maxCardinality restrictions. Figure 16.21 explains the meaning of these restrictions by relating them to an equivalent representation in ORM. The ORM fact type *A R B* is diagrammed as an RDF graph, where the subject and object nodes are assumed to be classes. The predicate *R* is directed from *A* to *B*.

Figure 16.21 Some correspondences between ORM and OWL.

OWL itself has no graphic notation, so the fact type and its restrictions are declared textually (see example later). Restricting *R* with a minCardinality of 1 means that each instance of *A* plays in that relationship type with at least one instance of *B*. In ORM terms, the first role of *R* is mandatory. In OWL DL and OWL Full, a minCardinalty above 1 may be specified (in ORM this adds a frequency constraint to the mandatory constraint).

Assigning a minCardinality of 0 means that the role is optional. Restricting the maxCardinality to 1 means that each instance of *A* plays in that relationship type with at most one instance of *B*. In ORM terms, the first role of *R* has a simple uniqueness constraint. In OWL DL and OWL Full, a maxCardinalty above 1 may be specified (in ORM this replaces the uniqueness constraint by a frequency constraint).

Restricting the maxCardinality to 0 means that each instance of the class (let's call this class *C*) being discussed (perhaps in a query) plays in that relationship type with no instance of *B*. In ORM terms, this could be handled explicitly by a subtype definition as shown, or simply included as a condition in an ORM query. For example, we might define a teetotaller as a person who drinks no alcoholic drink.

As an example of some cardinality decarations, the following OWL code declares that the hasFaxNr predicate is optional and *n*:1.

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#hasFaxNr"/>
  <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">0</owl:minCardinality>
<owl:Restriction>
```

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#hasFaxNr"/>
  <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:maxCardinality>
</owl:Restriction>
```

As a shortcut when the minimum and maximum cardinalities are the same, the value may be assigned to the cardinality property. So setting owl:cardinality to 0 means forbidden, and setting it to 1 means exactly one.

In OWL Lite, predicates may be declared to be transitive (e.g., ancestorOf) or symmetric (e.g., spouseOf). One predicate may be specified as the inverse of another (e.g., isOwnedBy is the inverse of owns). Class extensions may be equated, and the intersection of two classes may be derived.

Two URIrefs (perhaps from different documents) may be declared to identify the same individual using the owl:sameAs predicate. The owl:differentFrom predicate can be used to indicate that individuals differ.

While OWL Lite provides an easy migration path for simple ontologies such as taxonomies, and can be implemented efficiently, it is too weak for modeling complex ontologies or business domains. It cannot express most of the constraints found in such domains, and its derivation capability is limited to simple inferences such as exploiting transitivity.

OWL DL (the "DL" refers to description logic) is based on the stronger $\mathcal{SHOIN}(\mathcal{D})$ description logic. This dialect increases the expressive power of the language, while retaining decidability. It includes all the language constructs of OWL Full, but places restrictions on their use. For example, OWL DL is a fragment of first order logic, so it does not allow a class to be an instance of another class.

OWL DL allows classes and values to be enumerated and allows classes to be declared mutually exclusive (using owl:disjointWith). Arbitrary Boolean (and, or, not) combinations of classes are allowed using owl:intersectionOf, owl:unionOf, and owl:complementOf properties.

By allowing arbitrary values for cardinalities, OWL DL allows single role frequency constraints to be specified perhaps combined with mandatory constraints (e.g., each duet has two members). Simple subclass definitions may be specified using the owl:hasValue predicate to restrict subclass instances to those instances of the specified superclass that have certain values (e.g., female patients are patients with their gender value is restricted to 'F').

OWL Full adds even more expressibility by removing various restrictions, allowing any RDF expression whatsoever. For example, OWL Full allows a class to be treated simultaneously as an instance. Hence OWL Full goes beyond description logics and even first order logic. Despite the undecidable nature of OWL Full, it is still not expressive enough to capture many common business rules. Moreover, there is no standard graphic notation for OWL, nor is there a standard high level language for its expression. Compare the earlier example of OWL code with the ORM verbalization **Each** Person has **at most one** FaxNr. Hence the direct use of OWL tends to be restricted to formal logicians or technically trained developers.

An obvious solution to this problem is to first express the ontology in a conceptual language such as ORM and then have a tool map it automatically to OWL as required. Some recent research efforts have investigated mapping from conceptual models in ORM, UML, and ER to OWL as well as other description logic-based languages. For example, Jarrar (2007) discusses mapping ORM to $\mathcal{SHOIN}(\mathcal{D})$, the description logic underlying OWL DL, and Keet (2007) discusses mapping ORM to the more powerful description logic $\mathcal{DLR}_{ifd}$.

Much of ORM can be mapped cleanly to these description logics, facilitating the use of efficient DL-based theorem provers for consistency checking of ORM schemas, and interoperability with other semantic web applications. Unlike OWL Full, ORM has a clean, first order formalization, so problems about types being instances are avoided. However, ORM is richer than OWL in some respects, since some of its structures and constraints cannot be captured in OWL or even $\mathcal{DLR}_{ifd}$. This has both good and bad consequences.

On the plus side, ORM is capable of capturing, in a natural fashion, several complex aspects of reality that commonly occur in business domains but cannot be captured in popular description logics. Hence ORM enables higher fidelity models of typical business domains. On the negative side, ORM is so expressive that its underlying logic extends a little beyond decidable fragments of first order logic. This makes it difficult to achieve efficient automated checking of aspects such as schema satisfiability (e.g., in very large schemas with millions of constraints, checking that none of the constraints contradict one another).

This is another example of the trade-off between expressibility and tractability. The more expressive the language, the harder it is to efficiently verify the logical consistency of models expressed in it. Since we really ought to ensure that our models accurately reflect the business domains they are intended to model, it seems reasonable to capture the intended semantics in a language such as ORM, apply efficient automated checking where possible, and use heuristic techniques for the rest.

Examples of ORM features that can't be translated into $\mathcal{SHOIN}(\mathcal{D})$, and hence OWL DL, include uniqueness and frequency constraints spanning multiple roles (of the same or different relationships), set-comparison constraints involving multirole arguments, acyclic ring constraints, and deontic constraints. The $\mathcal{DLR}_{ifd}$ description logic adds support for identifier and functional dependency constraints, so it can capture internal uniqueness constraints spanning multiple roles as well as external uniqueness constraints.

Unlike ER and UML, ORM is attribute free and hence is easier to relate to RDF, RDFS, OWL, and description logics, which also represent all facts in the form of relationships. Although RDF and OWL use binary relationships only, it is a trivial matter to transform unaries and *n*-aries into binary form, as discussed in earlier chapters. So there is a fairly obvious mapping of facts and fact types in binary ORM to these semantic web languages. The only aspects that are lost in the mapping are those ORM constraints and derivation rules that have no counterpart in the logic underlying the target language. These can be mapped over as unambiguous verbalizations in ancillary notes.

Given the significant potential of ORM for capturing ontologies, it is not surprising that research laboratories are actively developing ORM-based ontology tools, such as DOGMA and T-Lex (Trog et al. 2006). An atomic (elementary or existential) fact type in ORM offers an ideal semantic unit for sharing semantics between different models. Such a unit is constraint free, allowing different models to apply different constraints without altering the semantics of the fact type. For example, in one model the fact type Person drives Car might be optional and *m:n*, while in another it might be mandatory and *n*:1. Using a fact-based approach instead of an attribute-based approach facilitates this kind of exchange. This is one reason why the OMG's SBVR approach for exchanging semantics of business rules is also fact oriented.

At the time of writing, the dream of Sir Tim Berners-Lee for a truly semantic web has not yet materialized. But given the rapidly increasing research efforts underway in both academia and industry on semantic web technology, it seems likely that this dream will be at least partly realized within the next decade or so.

16.7 Postrelational Databases

A conceptual schema may be mapped to various logical data models. The Rmap procedure discussed in Chapter 11 assumes that the application platform is a centralized, relational DBMS. It is a fairly simple task to specify other procedures for mapping to pre-relational systems such as hierarchic or network DBMSs. However, it has been argued that relational database systems are "out of date" and should be replaced by something better (e.g., object-oriented or deductive database systems). Moreover, there is a growing trend for databases to include special support for nontraditional kinds of data (e.g., spatial data, images, and sounds) and for databases to be decentralized in one way or another. This section outlines some issues behind these movements, and some other recent trends in database research.

Although relational DBMSs have long been dominant in the commercial marketplace, traditional systems based on the hierarchic or network data model are still in use. While purely relational DBMSs suit most business applications, they may be unsuitable for complex applications such as CASE tools, computer-aided design tools (e.g., VLSI design, mechanical engineering, architecture), document processing, spatial databases, expert systems, scientific databases (e.g., genetic engineering), and communications management. Note that most of these applications involve complex objects. Moreover, these account for at most 10% of database applications. For the other 90%, a relational DBMS is quite satisfactory.

Many reasons are cited for dissatisfaction with *purely* relational DBMSs for complex applications. They may be too slow—they don't perform well with complex objects mainly because they require too many table joins. They often model objects in an unnatural way (e.g., information about a single object such as a person may be spread over several tables—this gets worse with complex objects). They are dependent on value-based identifiers. They don't facilitate reuse (e.g., they have no direct support for subtyping). They require access to a procedural language for difficult tasks (e.g., spe-

cial rules and behavior), which leads to an "impedance mismatch" with the declarative, set-based, relational query language. They might not support binary large objects (BLOBs) such as images (icons, pictures, maps etc.), sound tracks, video, and so on.

Over the last few decades, various research efforts aimed to develop a next generation of DBMS to overcome such deficiencies. Some new kinds of database that emerged are object databases, object-relational databases, deductive databases, spatial databases, temporal databases, and XML databases. We'll briefly examine each of these in turn.

Object Orientation

An *object database* (ODB), or *object-oriented database* (OODB), incorporates various object-oriented features. Historically, ODB research drew upon related developments in four areas: programming languages, semantic data models, logical data models, and artificial intelligence.

With programming languages, the need was seen for user-definable abstract data types and for persistent data. The object-oriented programming paradigm began with Simula (1967), and Smalltalk (1972). Nowadays many object-oriented programming languages exist (e.g., Eiffel), and many traditional programming languages have been given object-oriented extensions (e.g., C++ and C#).

Some object-oriented features were taken from semantic data modeling, which as we know models reality in terms of objects and their relationships and includes notions such as subtyping (e.g., ORM, UML, and extended ER). Various ideas were also borrowed from work on logical data models (network, hierarchic, relational, and especially the nested relational model). Finally, some concepts were adapted from artificial intelligence, where structures such as frames are used for knowledge representation.

What features must a DBMS have to count as "object-oriented"? There is no commonly agreed-upon answer. A classic paper ("The OODBMS Manifesto") on this issue was presented at the first international conference on object-oriented and deductive databases (Atkinson et al. 1989). To distinguish ODBMSs from OO-programming languages, five essential features of a DBMS were identified and then object-oriented features were added, of which eight were considered essential and five optional (see Table 16.2). Let's look briefly at the essential object-oriented features in this proposal.

Complex objects are built from simpler ones by constructors. The manifesto proposed that these constructors should include at least set, list, and tuple, and be orthogonal (they can be applied in any order, recursively). For example, one object might be a list of sets of sets of tuples. Constructors are not orthogonal in the relational model (only sets of tuples of atomic values are allowed) or the nested relational models (e.g., the top level construct must be a set). The notion of complex objects is considered important since it allows us to model a complex structure in a direct, natural way.

The basic idea behind *object identity* is that objects should be identified by system-generated object identifiers (oids) rather than by the values of their properties. This is in sharp contrast to the relational model, where for instance tuples are identified by the value of their primary key.

Table 16.2 Feature list in the "OODBMS manifesto" (Atkinson et al. 1989).

DBMS features	*Essential OO features*	*Optional OO features*
Persistence	Complex objects	Multiple inheritance
Secondary storage	Object identity	Type checking and inferencing
Concurrency	Encapsulation	Distribution
Recovery	Types or classes	Design transactions
Ad hoc query facility	Inheritance	Versions
	Overriding and late binding	
	Computationally complete	
	Extensibility	

Among other things, oids can help keep track of objects whose external, value-based identification may change with time. This may occur because of simple renaming. For example, television channel 0 becomes channel 10, or a woman changes her family name on marriage. More drastically, the reference scheme itself may change (e.g., a student identified by a student number becomes an employee identified by an employee number).

Object identifiers in the OO sense partly overcome this problem since they are rigid identifiers (i.e., they refer to the same object throughout time). They are system generated, nonreusable, immutable, and typically hidden. In ODB systems, they might be implemented as surrogates (logical identifiers or autoincremented counters, mapped by indexes to physical addresses), as typed surrogates (which makes migration between types awkward), or as structured addresses.

Encapsulation involves bundling the operations and data of an object together, with normal access to the object being through its operational interface, with implementation details hidden. For example, hiring, firing, and promoting employees are regarded as operations on Employee and are encapsulated with it. Implementations of operations are often called "methods".

Encapsulation includes the idea that, as in a conceptual schema, objects should be classified in terms of *types* or *classes*. Moreover, some form of *inheritance* mechanism should be provided (e.g., so that a subtype may inherit the data and operations of its supertype(s)). A subtype may have a specialized version of a function with the same name as one its supertype functions. In this case, the specialized version will typically *override* the more general version when the operation on the subtype is invoked. For example, the display procedure for a colored, equilateral triangle may differ from the general display procedure for a polygon. Various overriding options are possible.

The requirement for *computational completeness* means that any computable function can be expressed in the data manipulation language (if necessary, by calling programming languages). The *extensibility* requirement means that users may define their own types, and the system should support them just like its built-in types.

Partly in response to the OODB manifesto, a group of academic and industrial researchers proposed an alternative "3rd generation DBMS manifesto" (Stonebraker et al. 1990). Here they referred to hierarchic and network systems as first generation, and relational systems as second generation. Under this scheme, third generation DBMSs

are the next generation. They specified three basic tenets and 13 detailed propositions to be adhered to by the next generation DBMSs (see Table 16.3). This proposal essentially argued for extending existing relational systems with object-oriented features and effectively laid the groundwork for today's object-relational database systems, which we'll discuss shortly.

Although supporting several of the object-oriented features, the group argued against implementing them in such a way as to negate key advances made by the relational approach. For example, all facts in a relational system are stored in tables. In contrast, some facts in hierarchic, network, and object database systems may be specified as links between structures, requiring navigation paths to be specified when the data is accessed.

Table 16.3 Next generation DBMS features proposed by Stonebraker et al. (1990).

Basic tenets	*Detailed propositions*
1. Besides traditional data management services, next generation DBMS will provide support for richer object structures and rules.	1.1 Next generation DBMS must have a rich type system. 1.2 Inheritance is a good idea. 1.3 Functions, including database procedures, methods, and encapsulation, are a good idea. 1.4 Unique identifiers (uids) for records should be assigned by the DBMS only if a user-defined primary key is not available. 1.5 Rules (triggers, constraints) will become a major feature in future systems. They should not be associated with a specific function or collection.
2. Next generation DBMS must subsume previous generation DBMS.	2.1 Essentially all programmatic access to a database should be through a non-procedural, high level access language. 2.2 There should be at least two ways to specify collections, one using enumeration of members and one using the query language to specify membership. 2.3 Updatable views are essential. 2.4 Performance indicators have almost nothing to do with data models and must not appear in them.
3. Next generation DBMS must be open to other subsystems.	3.1 Next generation DBMS must be accessible from multiple high level languages. 3.2 Persistent X for a variety of Xs is a good idea. They will all be supported on top of a single DBMS by compiler extensions and a (more or less) complex run-time system. 3.3 For better or worse, SQL is intergalactic dataspeak. 3.4 Queries and their results should be the lowest level of communication between a client and a server.

Figure 16.22 An ORM schema about university records.

Hence OO queries typically require specification of access paths. Although path expressions are often more compact than relational queries, their reliance on existing navigation links can make it difficult to perform ad hoc queries efficiently.

As an example, consider the UoD schematized in Figure 16.22. Students either hold a degree or are currently seeking one (or both). The exclusion constraint forbids students from re-enrolling in a degree that they already hold. Subjects are recorded only for current students, as shown by the subset constraint. Since this is not an equality constraint, students may enroll in a degree before choosing subjects. This aspect of the schema may alternatively be modeled by introducing the subtype CurrentStudent, to which the optional takes role is attached, using the definition **Each** CurrentStudent **is a** Student **who** seeks **some** Degree.

Each subject is identified by its subject code, but also has a unique title. For some subjects, a lecture plan may be available. This lists the topics discussed in the various lectures. For example, lecture 10 for CS114 might discuss the following topics: relational projection, relational selection, and natural joins. The footnoted textual constraint qualifies (and implies) the external uniqueness constraint, requiring lectures to be numbered from 1 within each subject.

The conceptual schema of Figure 16.22 (minus some constraints) might be specified in an ODB schema roughly as follows, using a syntax based on the Object Definition Language (ODL), as specified in Cattell and Barry (2000).

```
class Student {
   attribute unsigned long studentNr;
   attribute string studentName;
   attribute set<string> degreesHeld;
   attribute string currentDegree; };

class CurrentStudent extends Student {
   relationship set<Subject> takes
         inverse Subject::is_taken_by; };
```

```
class Subject {
   attribute string subjectCode;
   attribute string title;
   attribute unsigned short credit;
   relationship set<CurrentStudent> is_taken_by
         inverse CurrentStudent::takes;
   relationship list<Lecture> has
         inverse Lecture::is_for; };

class Lecture {
   attribute unsigned short lectureNr;
   relationship Subject is_for
         inverse Subject::has;
   attribute set<string> topics; };
```

Although readable, this OO schema is incomplete and needs additional code for the extra constraints in the conceptual schema. We could display this on a UML class diagram, with the extra constraints in notes (e.g., exclusive-or, pair exclusion, and subset constraints). In general, OO schemas are best developed as abstractions of ORM schemas.

Notice that the OO classes correspond to the major object types abstracted from the conceptual schema. Hence they provide a useful basis for building screen forms for the application. Encapsulation involves adding generic operations as well as type-specific operations (e.g., a graduate operation might be added to Student).

One problem of the ODB approach is that it mixes too many levels together—an OO schema includes conceptual, logical, and physical elements. It also involves redundant specification of associations. For example, consider the fact type: Student takes Subject. In the OO schema this is specified twice: once on CurrentStudent and again on Subject.

In specifying the OO schema, inverses were used to declare bidirectional object links between CurrentStudent and Subject, as well as between Subject and Lecture. This enables queries in either direction to be specified using path expressions. This versatility is lost if the link is made unidirectional. Notice that no such navigational problem can exist in the relational model, although of course many queries will require joins. While joins might slow things down, there is no restriction on their use, since they do not require links to be set up beforehand. In contrast, the ODB approach obtains its efficiency by "hardwiring" in the links to be used in queries. This makes it difficult to optimize ad hoc queries.

The problem of relying on predeclared access paths in the schema itself to achieve efficiency is exacerbated when the schema evolves. We may then have to reset navigation pathways to optimize them for the new situation rather than simply relying on the system optimizer to do the job, as in current relational systems.

This is not to say that ODBs are a bad idea, or that complex objects should not be modeled as such at the implementation level. Many complex structures are awkward to model in relational terms. However, we should be able to specify such structures in a clean way, without resorting to low level mechanisms.

Unlike ODBs, relational databases are based on a single data model formally based on predicate logic. Moreover, relational DBMSs are now dominant, and many are being extended to address several of the deficiencies mentioned earlier. Such extended relational database systems are usually called *object-relational database* (ORDB) systems. These are essentially relational DBMSs extended by adding support for many OO features, such as extra data types (spatial, image, video, text …), constructors (arrays, sets, …), and inheritance. Some of the major commercial systems support extended data types by allowing modules to be plugged in. Such modules may be developed by the vendor or a third party. These modules are variously called "relational extenders", "datablades", or "data cartridges".

The SQL:2003 standard has also been extended significantly to include several object-oriented features (e.g., user-defined types and functions, encapsulation, support for oids, subtyping, triggered actions, and computational completeness), as well as deductive features (e.g., recursive union).

Given the massive installed base of relational DBMSs, the ongoing extensions to these products, the cost of porting applications to a new data model, and the widespread adoption of the SQL standard, it may well be that the next generation of DBMSs will evolve out of current relational products. Just as Hinduism absorbed features from other religions that threatened its existence, the relational model can probably absorb the interesting features of the object-oriented faith without being replaced by it.

It's debatable whether all the transformations taking place in SQL-based DBMSs are actually desirable. While the use of extended data types (spatial, video, etc.) is clearly a step forward, the use of constructors (arrays, multisets, etc.) is questionable in many cases, since they can make it harder to design efficient databases for ad hoc queries and updates, and they complicate the task of writing good optimizers. For a detailed critique of the way in which many OO features have been grafted onto the relational model, see Date (2000) and Date and Darwen (1998). At any rate, relational and object-relational systems currently dominate commercially, and other contenders such as object database systems are struggling to gain any significant market share.

Although object-relational systems may well dominate the market for the near future, they may be "overkill" for some specialized or small applications. And there are things that even SQL:2003 can't do (e.g., it doesn't allow triggers on views) or does awkwardly (e.g., recursion).

SQL:2003 includes arrays, row types, and multisets. One challenge then is to model such collections and provide mapping algorithms to implement them efficiently. Various constructors (e.g., for sets, bags, sequences, and schemas) have been added to some versions of ORM and ER. In addition, both ORM and UML allow some kinds of collections to be expressed as mapping annotations on the conceptual schema. Although the use of collection types in conceptual models can facilitate mapping to equivalent implementation structures, it is typically much harder to model complex objects directly at the conceptual level, and they are extremely awkward to validate by verbalization and population. Complex objects also tend to be overused, and simpler solutions overlooked. For further discussion on collection types, see Section 10.4.

Nowadays many applications are built using a 3GL such as C# or Java to specify an object model for the transient, in-memory storage, and using SQL to specify a relational or object-relational database for the persistent storage. As data needs then to be moved between the object model and the relational model, considerable effort is spent on the *object-relational mapping*. Using a tool such as NORMA to generate both object and relational models from ORM models simplifies the task of creating an object-relational mapping process. As another way to facilitate moving data between transient and persistent models, Microsoft is incorporating LINQ (Language Integrated Query) into .NET languages such as C# and Visual Basic, allowing the programmer to use an SQL-like syntax to query databases from inside the 3GL.

Other recent trends

Deductive databases offer elegant and powerful ways of managing complex data in a declarative way, especially for information that is derived by use of recursion. Deductive systems typically provide a declarative query language such as a logic programming language (e.g., Prolog). This gives them a strong rule enforcement mechanism with built-in backtracking and excellent support for recursive rules. For example, the ancestor relation can be derived from a base parent relation and the following two rules: X is an ancestor of Y **if** X is a parent of Y (basis clause); X is an ancestor of Y **if** X is a parent of Z **and** Z is an ancestor of Y (recursive clause).

In contrast, SQL-92 cannot express recursive queries at all. SQL:1999 introduced a recursive union operator, but its syntax is more complex and its execution does not enjoy built-in back-tracking. Despite their elegance however, deductive database systems have major problems to be solved (especially in the performance area) and in the short term are unlikely to achieve more than a niche market.

Although purely deductive databases are not popular, there is a growing need to enforce many Event-Condition-Action (ECA) rules. For example, on the event that client X requests an upgrade to class 'B' on flight Y, if the condition that X.rating = 'premier' and count(vacant B seats on flight Y) > 0 is satisfied, then perform the following action: upgrade client X to class 'B' on flight Y. Most relational DBMSs effectively support ECA rules by using triggers or procedures, and triggers are included in the SQL:1999 standard.

Two specialized database varieties that have recently become popular are spatial databases and temporal databases. *Spatial databases* require efficient management of spatial data, such as maps (roads, land, etc.), two-dimensional designs (circuits, town planning, etc.), and three-dimensional designs (visualization of medical operations, molecular structures, flight paths, etc.). They provide built-in support for spatial data types (points, lines, polygons, etc.), spatial operators (overlap, contains, intersect, etc.) and spatial indexes (R-trees, quad trees, etc.). This allows efficient formulation and processing of spatial queries, such as: How many houses are there within 5 km of the proposed shopping center? Which flights fly over the greater LA area? Which diagrams are similar to diagram 37?

Previously, special systems were used for spatial data while a relational DBMS was used for alphanumeric data. The current trend is to manage both standard and spatial data in the one system. A typical application of a geographic information system (GIS) might proceed as follows: standard data on traffic accidents is entered; the road maps are displayed to highlight the accident sites; and this is now used to determine regions where extra precautions need to be taken (e.g., radar traps).

Historically, GIS vendors adopted three main approaches: hybrid (georelational—objects in a spatial file store are given identifiers that can be referenced in relational tables); integrated (spatial and non-spatial data are stored in relational tables); and object oriented (all data is stored in an ODB). More recently, many spatial database applications as well as other applications using non-standard data have been implemented using object-relational technology.

Although time has only one dimension, unlike space's three dimensions, the efficient management of *temporal information* is no easy task. If historical rather than snapshot records need to be maintained about objects, time will feature largely in the modeling.

As discussed in Section 10.3, a variety of approaches may be adopted for modeling time. In some cases, we simply include object types such as Time and Period on the conceptual schema and map these like other object types. Distinctions may be needed between transaction time (when the system records a fact) and valid time (when the fact is true in the UoD being modeled). Often we need to make use of temporal relations (such as before and after) and temporal operations (e.g., to compute an interval between two time points).

The SQL standard currently includes only basic support for temporal data, and sometimes an ordinary relational database does not allow temporal aspects to be implemented efficiently. For such applications, special DBMSs known as "temporal database systems" are sometimes used; these provide in-built support for automatic time stamping and the various temporal operators.

Most work on temporal databases focuses on maintaining relevant histories of application objects through time, with the assumption that the conceptual schema itself is fixed. Moving up one level, the problem becomes more complicated if we allow the conceptual schema itself to change with time. This is one aspect of the schema evolution problem.

Moving up another level, we might allow the conceptual metaschema itself to change with time (e.g., we might decide at a later stage to allow constructors for complex object types in our conceptual schema language). The management of such higher order evolution has been addressed in research on evolving information systems. This topic provides one motivation for the next section on metamodeling.

Apart from the kind of database used, the size and spread of databases have seen a continued upward trend. Many databases are becoming very large, with users at many different sites. For this situation we need to decide whether the overall system will be centralized, distributed, or federated.

In a *centralized* system, the database management is controlled at a single site. Any site may send update and query requests to the central site, and results are sent back. If

the sites are far apart, the communication times involved in such transactions can be very significant.

To reduce the communication overhead, a *distributed* database system allows the data to be spread across various sites, with most of the data relevant to a given site stored locally at that site. In the simplest case, the population of a schema might be partitioned (e.g., each branch of a bank stores data about its clients only). Typically, however, there is a need to replicate some data at more than one site, thus requiring measures to be enforced to control the redundancy. As you might guess, optimizing the performance of a distributed system requires attention to a whole new batch of problems. The research literature on distributed databases is vast, and many commercial systems provide distributed capabilities, to varying extents.

Federated databases deal with situations where there is a need for data sharing between several existing database systems, possibly heterogeneous (e.g., some relational, some hierarchic, and so on). In this framework, each individual system maintains its local autonomy and communicates with other sites on a needs basis. As the heterogeneity problem requires translation between different data models, the control of federated systems is non-trivial. The size of the problem can be reduced by supporting only partial integration; any two sites need only share the common data relevant to both of them rather than share all their data.

Different solutions have arisen to address the problems of communicating between different database systems, possibly of different types. For a long time, SQL has been used as a common language between relational DBMSs. In today's world of eCommerce, the variety of systems that need to exchange data has grown significantly. Currently, the most popular solution to this problem is to use *XML* for communicating structured information between different systems.

As discussed in Section 13.9, XML is a low level, hierarchically structured, textual language that allows the specification of both schema and data. The SQL:2003 standard now includes built-in support for XML. Most major relational DBMSs also provide support for XML, including automatic conversion between relational and XML structures.

Like SQL, XML is good for communication between computer systems, but is not high level enough for humans to easily visualize and model their business domain. Hence an XML schema is best developed by first modeling in a high level language such as ORM, ER, or UML and then mapping the model to XML. For a discussion of mapping ORM to XML schema, see Bird et al. (2000).

One of the most exciting, and perhaps almost frightening, trends in modern computing has been the recent progress in artificial intelligence. If you'd like an insight into where computer science may well go in the next 50 years, have a read of Denning and Metcalfe (1997). For a radical view of where artificial intelligence may take us in the next 100 years, see Kurzweil (2005)—we disagree with some of his projections, which, among other things, assume a materialist philosophy, but they are worth thinking about.

The chapter notes provide some further discussion and references for the topics covered in this section.

16.8 Metamodeling

Modeling involves making models of business domains. *Metamodeling* involves making models of models—this time things being modeled are themselves models. Just as recursion is one of the most elegant and powerful concepts in logic, metamodeling is one of the most beautiful and powerful notions in conceptual modeling. This section uses a simple example to convey the basic idea.

A database holds fact instances from a business domain, while its conceptual schema models the *structure* of the domain. Figure 16.23 recalls our basic view of an information system, where the information processor ensures that the database conforms to the rules laid down in the conceptual schema. Essentially, a DBMS is a system for managing various databases; for each database that models some UoD, it checks that each database state agrees with the structure specified in the conceptual schema for that UoD.

Figure 16.23 The database must conform to the structure of the conceptual schema.

Among other things, a conceptual modeling tool is a system for managing conceptual schemas. Each valid schema diagram in this book may be thought of as an output report from this system. The trick then is to treat a schema as an instance of this higher level system. As long as we can verbalize the diagrams into atomic facts, we can use the CSDP to develop a conceptual schema for such conceptual schemas. We would then have a conceptual *metaschema* (schema about schemas).

Suppose that Table 16.4 is part of an output report from a movie database. Other reports from the same domain provide further information (e.g., people's birth places).

Table 16.4 One report extracted from a movie database.

MovieNr	*Title*	*Director*	*Stars*
1	Wilderness	Tony O'Connor	
2	Sleepy in Seattle	Anne Withanee	Ima Dozer Anne Withanee
3	Wilderness	Anne Withanee	Paul Bunyip

Figure 16.24 A conceptual schema for Table 13.4.

The conceptual subschema for this report is shown in Figure 16.24. The inclusive-or constraint on Person is shown explicitly since Person plays other roles in the global schema (e.g., as in Figure 16.8 from an earlier section).

In this case, the information system architecture in Figure 16.23 still applies, but the user is an information modeler, the database holds a conceptual schema, the UoD is about conceptual schemas, and the conceptual information processor ensures that only valid conceptual schemas are placed in the database by checking that they satisfy the structure specified in the meta-conceptual schema.

Rather than developing a complete metaschema for ORM conceptual schemas, let's confine our discussion here to simple examples such as Figure 16.24, ignoring nesting, subtyping, derivation, and all constraints other than uniqueness and mandatory role constraints. If you've never done metamodeling before, it seems a bit strange at first. As a challenge, try to perform CSDP step 1 using Figure 16.24 as a sample report.

Metamodeling is like ordinary modeling, except the thing being modeled is itself a model. You might begin by describing Figure 16.24 roughly. For example: "It has two entity types (Movie and Person) and one value type (MovieTitle). The first role of the 'was directed by' predicate is mandatory and has a uniqueness constraint", and so on. This verbalization conveys the information, but we need to express it in terms of atomic facts. For example, using "ObjectKind" to mean "kind of object type", we might say: "The ObjectType named 'Movie' is of the ObjectKind named 'Entity'".

In previous chapters we saw that the same UoD may be modeled in different ways. This applies here too, since there are many different ways in which the information in Figure 16.24 can be verbalized as atomic facts. For example, consider the information that both roles of the predicate called "starred" are spanned by the same uniqueness constraint. How would you say this in atomic facts?

With diagrammatic applications like this, you often find that you want to talk about an object (such as a constraint) but it *hasn't got a name* on the diagram. You would naturally identify it to somebody next to you by *pointing* to it, but this won't help you convey the information over the telephone to someone.

In such cases it is often convenient to introduce an artificial name, or *surrogate*, to identify the object. This is done in Figure 16.25, where each constraint is given an identifying constraint number. For convenience, we've also introduced role numbers and predicate numbers (although we could have identified roles by their positions in predicates, and predicates by their expanded fact type readings).

Figure 16.25 Surrogates are added to identify constraints, roles, and predicates.

If readings are used to identify predicates, we need to expand them with the object type names (e.g., to distinguish the starred predicate in Movie starred Person from that in Play starred Person, or the runs predicate in Horse runs Race from that in Person runs Barbershop). Note that in these examples, the two "starred" predicates have the same meaning, whereas the two "runs" predicates have different meanings. We could extend the metamodel to capture this distinction, but for simplicity we ignore this issue here.

The metaschema shown in Figure 16.26 is only one of many possible solutions. As an aid to understanding, it is populated with the database, or set of facts, that corresponds to the conceptual schema shown in Figure 16.25. Here, "UC" and "MR" abbreviate "uniqueness constraint" and "mandatory role", respectively. Whether a uniqueness constraint is internal or external is derivable by checking whether its roles come from the same predicate. Whether a mandatory constraint is simple or disjunctive (inclusive-or) is derivable by checking whether it spans multiple roles. For simplicity the diagram omits these derivations.

Recall that other constraints are ignored in this discussion. For simplicity, only one reading is stored for each predicate, and nesting is ignored. This solution also ignores the implicit reference types implied by reference modes. If you developed an alternative metaschema, don't forget to do a population check.

The metaschema actually contains features not present in our original example (e.g., value constraints and subtyping). So it is not rich enough to capture itself. As a nontrivial exercise you may wish to extend the metaschema until it can capture any ORM schema. For instance you can capture subtype links by adding the fact type: ObjectType is a subtype of ObjectType. To test a full ORM metaschema, you should be able to populate it with itself.

Metamodeling is not restricted to conceptual schemas. Any well-defined formalism can be metamodeled. Apart from being used to manage a given formalism, metamodels can also be developed to allow translation between different formalisms (e.g., ER, ORM, and UML). This is sometimes referred to as metametamodeling.

Because of ORM's greater expressive power, it is reasonably straightforward to capture data models in UML or ER within an ORM framework. Although less convenient, it is possible to work in the other direction as well. To begin with, UML's graphic constraint notation can be supplemented by textual constraints in a language of choice (e.g., OCL). Moreover, the UML metamodel itself has built-in extensibility that allows many ORM-specific constraints to be captured within a UML-based repository.

Figure 16.26 A conceptual metaschema, populated with the schema of Figure 16.25.

For example, the ORM model in Figure 16.27(a) contains four constraints $C_1..C_4$. While the uniqueness constraints are easily expressed in UML as multiplicity constraints, the subset and exclusion constraints have no graphic counterpart in UML. The UML metamodel fragment shown in Figure 16.27(b) extends the standard UML metamodel with constraintNr, constraintType, and elementNr attributes, and SetCompConstraint as a subtype along with the argLength attribute.

The full UML metamodel is vast, so we've included only the fragment relevant to the example. The attribute constraintType stores the type of constraint (subset, exclusion, mandatory, etc.). SetCompConstraint denotes set comparison constraint (subset, equality, or exclusion), and argLength is the argument length or number of roles (association ends) at each end of the constraint.

This metamodel fragment is probably easiest to understand in ORM. Figure 16.27(c) shows an ORM metamodel fragment with sample population based on Figure 16.27(a).

The four ORM constraints may now be stored as the object-relation shown in Table 16.5. The subset (SS) and exclusion (X) constraints have their argument length recorded. The actual arguments of these two constraints may now be derived by "dividing" the role lists by this number.

Figure 16.27 Modeling ORM set-comparison constraints in ORM and extended UML.

Thus the arguments of the subset constraint are the simple roles r_4 and r_2, whereas the arguments of the exclusion constraint are the role pairs (r_1, r_2) and (r_3, r_4). The constraint type may now be used to determine the appropriate semantics.

Although this simple example illustrates the basic idea, transforming the complete ORM metamodel into an extended version of UML is complex. For example, as the UML metamodel fragment indicates, UML associations must have at least two roles (association ends), so artificial constructs must be introduced to deal with unaries.

Table 16.5 Meta-table for storing ORM constraints.

***Constraint*:**

constraintNr	*constraintType*	*roles*	*argLength*
C_1	UC	r_1, r_2	
C_2	UC	r_4	
C_3	SS	r_4, r_2	1
C_4	X	r_1, r_2, r_3, r_4	2

The following exercise includes several questions to hone your metamodeling skills. A taste of metamodeling can really whet one's appetite, but unfortunately this is as far as we go in this book. The authors hope that you have gained some insights into the science and art of conceptual modeling by reading this book and that you share their belief that modeling the real world is one of the most challenging, important, and satisfying human activities.

Exercise 16.8

1. Devise a conceptual metaschema (in ORM, UML, or ER) to store simple SQL schemas where each table has at most one primary key (possibly none). A primary key may be composite (multi-column), but each foreign key references a simple (single column) primary key. Tables are identified by simple names. Columns are ordered by position, and may be optional or mandatory. Each column in a primary key is mandatory, but it is possible that all columns in a keyless table are optional. Ignore domains and all other constraints (e.g., uniqueness constraints on column sets other than primary keys). For example, your metaschema should be able to store the following schema:

2. Map your answer to Question 1 to a relational metaschema.

3. A sample UML class diagram is shown.

Specify a metaschema in ORM notation for simplified UML class diagrams that are restricted to the following features. Classes have attributes only (e.g., no operations). Attributes have visibility (+ = public, # = protected, - = private, ~ = package) and multiplicity (* = 0 or more, 0..1 = 0 or 1, 1..* = 1 or more, 1 = exactly 1). Associations may be binary or *n*-ary. No association classes or qualified associations are allowed. Association ends have multiplicity (* = 0 or more, 0..1 = 0 or 1, 1..* = 1 or more, 1 = exactly 1). Role names are mandatory (whether or not they are displayed). The only constraints that may be added to a

subclassing scheme are disjoint and complete. No derived attributes or derived associations are allowed. No notes are allowed. No aggregation is allowed.

No other graphic constraints (e.g., {ordered}, {xor}, {subset}) or textual constraints are allowed. Ignore our extensions (e.g., {P}, {U1}). Ignore data types. Ignore instance populations and presentation aspects (e.g., layout, or whether the display of a feature is suppressed). Your metaschema should be able to store a single class diagram like the one shown.

4. Specify a metaschema in ORM notation for Barker ER (as discussed in Section 8.2). Your metaschema should be able to store a single Barker ER diagram like the one shown.

5. Specify a metaschema in ORM for *deterministic, finite-state automata*. Two examples of a deterministic automaton are depicted in the transition graphs shown here. Each state appears as a named circle. The starting state has a no-input (unlabeled) arrow attached to it. Each automaton has one or more accepting states, depicted as named double-circles. Transitions between states are depicted as labeled arrows from a state to the next state, where the label names the action (or disjunction of actions) causing the state transition.

A sequence of transitions is acceptable (well formed) only if it ends in an accepting state. An accepting state cannot be the start state. The examples have only one accepting state, but your metaschema should allow for multiple accepting states. A finite-state automaton is deterministic if and only if for each action applied to a state there is only one possible resulting state. Your metaschema should be able to store the information content of single transition graphs like the examples shown.

6. In a *nondeterministic* finite-state automaton, a given action performed on a given state may result in one of many different transitions (we cannot know in advance which transition option will occur for any specific occurrence of the action being applied to the state). For instance, in the following example the action *a* performed on state s_0 might cause a transition to state s_1 at one time and to state s'_1 at another time. Modify your answer to (a) to cater for nondeterministic automata.

16.9 Summary

Online transaction processing (*OLTP*) systems are used for typical database applications involving frequent updates and queries. A *data warehouse* is updated only on a scheduled basis from online sources and is used in read-only mode for enterprise-wide analysis of trends by management. *Data marts* are departmental subsets derived from the data warehouse but focused on a single subject area. Data marts are often denormalized for performance reasons and are often based on a "*star schema*" structure with one central "fact" table, referenced by dimension tables. Data warehouses and data marts are used for Online Analytical Processing (*OLAP*) and data mining. Often a star schema is used to construct a multidimensional structure, or *cube*, containing base and aggregated measures that can be accessed across various dimensions, including time.

Just as information systems can be modeled at different levels, so can they be queried at different levels. Conceptual query languages allow queries to be formulated either in natural language (e.g., English Query) or by selecting and constraining paths on a conceptual schema. (e.g., ConQuer). Conceptual query tools automatically transform these high level queries into queries in a lower level language such as SQL. Conceptual queries based on an attribute-free model such as ORM are far more stable when the schema evolves.

With large, complex schemas there is a need for *abstraction mechanisms* to hide details that are not of immediate relevance. Modularization divides the global schema

into conveniently sized subschemas. Constraint layers and textual rules may be toggled off and on as desired. We may zoom in on, or out from, a selected object type, specifying how much of its neighborhood is to be displayed.

Major object types may be identified by their importance in the global schema (e.g., due to having an explicit mandatory role), and the display suppressed for "minor" fact types (those without two roles played by major object types). This view may be refined further by focusing on its major fact types, and so on. Such bottom-up abstraction may be reversed to give top-down refinement.

Minor fact types may also be displayed in terms of attributes of major object types. In this case, ER diagrams and UML class diagrams may be generated as abstractions of ORM diagrams. While such abstractions are good for compact overviews, detailed ORM diagrams are best for developing, transforming, and evolving conceptual schemas.

Although the data model is the foundation of any information systems application, there is also a need to specify *process models* to indicate the permissible *operations* on the data and to design the *external interface*. For such purposes, various diagrams and notations may be used to supplement data model diagrams.

In early stages, informal diagrams such as cartoons can help clarify the UoD, and *function trees* and UML *use case diagrams* may be used to provide overviews of the system requirements. *Data flow diagrams* may also be used to picture information flows among processes, agents, and information stores. However at some stage a formal connection needs to be made between processes and data. For this purpose, *data use cases* may be used to determine the data model to which processes may be bound (e.g., by defining *operations on the major object types*). This can also be used to generate the content (operations) of default screen forms for the external interface. Some default decisions can be made about the format and use of these screens.

Design of the *human-computer interface* (HCI) typically requires careful planning by humans. Access rights for different user groups need to be controlled. Users may also differ in their ability level. To make the *user interface* (UI) easy to learn and use, the interface should be consistent, simple, structured, efficient and adapted to the user. In most cases a graphical user interface (GUI) is preferable. The *menu* design needs to provide efficient navigation, and various levels of online and offline help are required.

In information modeling the term "*ontology*" is used for a conceptual model that is specifically designed for sharing with other models. The term "*semantic web*" refers to an enhanced web for the future where documents on the Internet are structured in such a way as to enable automated agents to extract their semantics. Semantic web applications are typically built on top of *OWL* (Web Ontology Language), which itself is built on top of RDF (Resource Description Framework) and RDF Schema, using XML Schema for the tagged textual representation and data types.

RDF breaks down information into atomic units that can be represented as simple (subject, predicate, object) triples denoting a binary relationship, using Uniform Resource Identifer references (*URIrefs*) to provide global identifiers for the resources that instantiate the subject, predicates, and object nodes in a triple. An object node may also be instantiated by a literal value (either untyped or typed).

RDF and RDFS are weakly expressive (very few constraints can be captured) but also too relaxed in allowing problematic assertions (e.g., a class may be an instance of itself). OWL Lite and OWL DL are based on description logics that lie in a decidable fragment of first order logic that is efficient to compute. OWL Full has the same looseness as RDFS. OWL DL can be used to capture constraints such as multiplicity constraints on binary associations, but is not rich enough to capture many ORM constraints (e.g., acyclicity, and uniqueness constraints spanning multiple roles). OWL also lacks a graphic notation. To fully exploit OWL, it seems best to capture the ontology first in a conceptual language such as ORM and then map to OWL or a more appropriate description logic, noting those constraints that are lost in the mapping.

Relational DBMSs are suitable for about 90% of business applications, but may prove inefficient for structurally complex applications. Such applications might best be implemented using either an *object-relational database* (ORDB) or an *object-database* (ODB). In the long-term future, *deductive databases* might prove a viable option, at least for specific niches.

Apart from the usual DBMS features, ODB systems provide direct support for complex objects, object identity, encapsulation, subtypes, overriding, and late binding and are computationally complete and extensible. Object identifiers are basically rigid, system-generated surrogates that are hidden from the user. They avoid many of the problems associated with changing identifiers, but it is at least debatable whether they should be used when a stable primary key is available.

Object-oriented database schemas include a mix of conceptual, logical, and external levels. Elementary facts may be redundantly specified to provide two-way navigation between objects. Like ER and UML schemas, OO schemas are awkward to populate and do not facilitate the expression of many constraints. Hence they are best developed after an ORM schema has already been constructed. Object-relational systems support some object-oriented features, but provide these on top of the basic relational features. To facilitate mapping to ODBs or ORDBs, it is sometimes argued that we should model complex objects in conceptual schemas using constructors for collections (e.g., set, bag, sequence, schema). However, it may well be better to keep the conceptual schema flat and later annotate it where necessary to provide mapping to collections.

Instead of being centralized, a database might be *distributed* or *federated*. This raises additional design and optimization problems (e.g., communication overhead, redundancy control, and translation). Further design problems may arise in the modeling of *temporal* or *spatial* data and in controlling the *evolution* of schemas.

By treating conceptual schemas as sample database states, the CSDP may be used to develop a conceptual *metaschema*. This may be used by a CASE tool to ensure that only valid conceptual schemas are entered. The activity of *metamodeling*, or making models of models, also helps to clarify and translate between different formalisms.

Chapter Notes

The literature on data warehousing and OLAP is substantial. Date (2000, Chapter 21) provides a clear overview of decision support. Inmon (1993), Kimball (1996) and Silverston et al. (1997)

provide practical advice on data warehousing. Jukik (2006) provides a useful overview of alternative strategies for modeling data warehouses. The gendercode example of data cleansing was suggested by Scot Becker—thanks Scot! Thomsen et al. (1999) provide a thorough discussion of OLAP for SQL Server, including a detailed coverage of MDX.

A brief review of conceptual query prototypes, such as ERQL, Super, Hybris, and CBQL, is given in Bloesch and Halpin (1997). Some other graphical database query languages are surveyed in Ozsoyoglu and Wang (1993). For further details on the English Query product, see *www.microsoft.com/sql/eq*.

The RIDL language was originally used in conjunction with Control Data's IAST tool for NIAM, which was the first true CASE tool for information modeling (Meersman and Van Assche 1983). A full description of the RIDL language is given in Meersman et al. (1984), and further discussions of RIDL may be found in Meersman (1982a, 1982b) and Verheijen and van Bekkum (1982). For further discussion of ConQuer, see Bloesch and Halpin (1996, 1997) and Halpin (1998a).

For more details about schema abstraction within ORM, see Campbell and Halpin (1994a) and Campbell et al. (1996) and their various references. A procedure for generating default external forms from major object types is discussed in Campbell and Halpin (1993).

McLeod (2000) offers useful suggestions for extending use case and activity diagrams in UML. For practical advice on traditional ways to model the process aspects of information systems, see Barker and Longman (1992) and Barker (1990). Some promising ways to integrate data and process are discussed by Shoval and Kabeli (2000), Halpin and Wagner (2003), Dietz and Halpin (2003), and Morgan (2007).

Extensive discussions of HCI design are provided by Shneiderman (1992) and Dix et al. (1993). Microsoft Windows user interface guidelines can be accessed at the following URL: *http://msdn.microsoft.com/library/books/winguide/welcome.htm*.

Specifications for RDF, RDFS, and OWL, as well as a helpful primer on RDF, may be found online at *www.w3.org*. A standard reference on description logics is Baader et al. (2003). For discussion of mapping ORM to OWL and description logics, see Jarrar (2007) and Keet (2007). For an example of an ORM-based ontology tool, see Trog et al. (2006).

The OODB "manifesto" is stated in Atkinson et al. (1989), and an alternative, extended-relational "manifesto" is proposed by Stonebraker et al. (1990). Two good papers on object identity are provided by Khoshaflan and Copeland (1990) and the ever lucid Kent (1991). One of the best books on object-oriented databases is Cattell (1991): this includes an excellent, annotated bibliography (pp. 273–310). Cattell & Barry (2000) provide a thorough coverage of the ODMG 3.0 standard. Date (2000, Chapters 24 and 25) provides a critique of object databases and object-relational databases. An alternative foundation for object-relational databases is provided by the "Third Manifesto" of Date and Darwen (1998).

Date (2000, Chapters 20, 22, and 23) gives a clear overview of distributed, temporal, and deductive databases. There is a vast literature on temporal databases and the impact of time in information systems. A classic survey of the area is given by Snodgrass (1990). For a survey of temporal issues within the context of fact-based modeling, see Petrounias and Loucopoulos (1994). An advanced treatment of evolutionary aspects of information systems is given by Proper (1994). A thorough treatment of temporal databases in SQL is provided by Snodgrass (2000). The latest official news on XML standards is accessible from the Web site *www.w3.org*.

Metamodels for ORM and Barker ER are discussed in Halpin (2000d,e). NIST (1993) includes a metamodel for IDEF1X. A metamodel for UML 2 is contained in the online UML specifications at *www.uml.org/*. The UML metamodel is large and currently has several errors. As a final exercise, see if you can find them!

An overly optimistic (or pessimistic, depending on which way you look at it) account of the future of artificial intelligence can be found in Kurzweil (1999, 2005). A more conservative account of the expected future of computing over the next 40 years is given in Denning and Metcalfe (1997).

ORM Glossary

This glossary lists key terms and symbols used in Object-Role Modeling (ORM), and briefly explains their meaning. A concise explanation of other technical terms may be found in the chapter summaries. Further details on technical terms may be accessed by using the Index.

Alethic constraint: Constraint that holds necessarily for all states of the model.

Arity: Number of roles in a relationship (unary =1, binary = 2, ternary = 3, etc.).

Asserted fact: Fact that is simply asserted, rather than being derived from others; also called a primitive fact or base fact.

Asserted subtype: Subtype that is simply asserted (not defined by a subtype definition).

Association: Relationship type, usually involving at least two roles.

Atomic fact: Either an elementary fact or an existential fact.

Base fact: Fact that is primitive (not derived from others). Also called an asserted fact.

Compound fact type: Fact type that is equivalent to a conjunction of smaller fact types

Conceptual schema: Conceptual model of the UoD structure; design that specifies what states and transitions are possible; declaration of fact types, constraints, and derivation rules

Conceptual schema design procedure (CSDP):

0 Divide the UoD into manageable sub-sections
1 Transform familiar examples into elementary facts, and apply quality checks
2 Draw the fact types, and apply a population check
3 Check for entity types that should be combined, and note arithmetic derivations
4 Add uniqueness constraints, and check arity of fact types
5 Add mandatory role constraints, and check for logical derivations
6 Add value, set comparison (subset, equality, exclusion) and subtype constraints
7 Add other constraints and perform final checks
8 Integrate the subschemas into a global conceptual schema

Constraint: Restriction on possible or permissible states (static constraint) or transitions (dynamic constraint).

Compositely identified object type: Either a coreferenced or a nested object type.

Coreferenced object: Object that is identified by means of two or more reference types in combination; hence its identification scheme involves an external uniqueness constraint.

Database: Variable set of related fact instances.

Deontic rule: An obligation, i.e. a rule that ought to be obeyed (but possibly may be violated).

Derivation rule: Rule that declares how one fact type may be derived from others.

Derived fact: Fact that is derived from other fact types using a derivation rule

Derived subtype: Subtype that is derived from other object types using a subtype definition.

Elementary fact: Assertion that an object has a property, or that one or more objects participate in a relationship, where the fact cannot be split into simpler facts with the same object types without information loss. Application of an atomic predicate to a sequence of objects.

Entity: Object that is referenced by relating it to other objects (e.g., the Country that has CountryCode 'AU'); not a value; typically, an entity may undergo changes over time; an entity is either atomic or nested (i.e. an objectified relationship); at the top level, entities are partitioned into primitive entity types, from which subtypes may be defined.

Existential fact: Assertion that an object exists (e.g., there is a Country that has CountryCode 'AU'); also called a reference.

Fact: Proposition that is taken to be true by the relevant business community, where the proposition is elementary or existential (rather than being a constraint or derivation rule).

Fact role: Role in an elementary fact type.

Fact type: Kind of fact, including object terms and either a predicate or existential quantifier.

Flatten: Restate without nesting.

Functional fact type: Fact type with a functional role.

Functional role: Role with a simple uniqueness constraint.

Generalization: Forming a more general case from one or more specific cases; the inverse of specialization.

Independent object: Object that may exist without participating in any elementary fact; the disjunction of fact roles played by an independent object type is optional.

Instance: An individual occurrence (one specific member of a type).

Mandatory role: Role that must be played by all instances in the population of the object type playing the role; also called a total role.

Modality: Mode in which a proposition is expressed. In ORM 2, modalities are either alethic (expressing necessities or possibilites) or deontic (expressing obligations or permissions).

Nested object: Relationship that plays some role (also called an objectified relationship).

Object: Thing of interest; an object may be an entity or a value.

Objectification: Treating a relationship as an object; also called nesting. Strictly speaking, objectification in ORM 2 distinguishes the object formed by the objectification from the original relationship, and hence involves situational rather than propositional nominalization.

Object-Role Modeling (ORM): Conceptual modeling method that pictures a business domain in terms of objects playing roles; it provides graphical and textual languages for verbalizing and querying information as well as various design and transformation procedures.

Population: Set of instances present in a particular state of the database.

Predicate: Proposition with object-holes in it, e.g. "… works for …".

Reference: Relationship used as the preferred way to reference or identify an object (or to provide part of the identification).

Reference mode: Mode or manner in which a single value references an entity; used to abbreviate simple reference schemes, e.g. (.code), (kg:).

Reference role: Role in a reference (existential fact).

Relationship: Property or association involving one or more objects.

Rigid subtype: Subtype whose instances must remain in that type for their whole lifetime (e.g, Person).

Rmap: Relational mapping procedure.

Role: Part played by an object in a relationship (unary, binary, ternary, etc.).

Role subtype: Subtype whose instances may leave that type during their lifetime (e.g., Child).

Semiderived fact type: Fact type. some of whose instances may be derived from others, while some other instances may be simply asserted.

Semiderived subtype: Subtype, some of whose instances may be derived using a derivation rule while some other instances may be simply asserted.

Subtype: Object type that is properly contained in another object type (e.g., Woman is a subtype of Person).

Type: Set of possible instances.

Uniqueness constraint (UC): Repetition is not allowed in the role or role sequence spanned by the constraint; a uniqueness constraint on a single predicate is an internal UC, and a uniqueness constraint over roles from different predicates is an external UC.

Universe of Discourse (UoD): Business domain (the aspects of the world that we want to talk about).

Value: Unchangeable object that is identified by a constant; in this book a value is either a character string or a number; sometimes called a label.

The following symbol glossary covers the main graphical symbols in ORM 2 (as supported by the NORMA tool) and the corresponding symbols in ORM 1 (as supported in Microsoft Visio for Enterprise Architects).

ORM 2	ORM 1
Object Types	
Entity type A (first shape is the default) A or A or A From now on, we show only the default shape.	A
A identified by *reference mode Ref* A (Ref)	A (Ref)
Value type A A	A
Independent entity type A A !	A !
Independent value type A A !	Not supported.
Duplicated object types A B	Not supported.
External object type A A^ To be finalized	A
Predicates	Predicates are basically the same, except that role boxes are larger
Unary: R	
Binary: R ◂ S R / S Forward reading Inverse reading Both readings	
Ternary: R	
Quaternary: R	
etc. (*n* role-boxes for *n*-ary predicate)	
Role names [role1] [role2]	Role names may be entered but are not displayed

ORM 2	ORM 1
Internal Uniqueness Constraints *Unary:* or *Binary:* *n:1* *1:n* *1:1* *m:n* *UC on role pair 1-2* *UC on role pair 1, 3* Many UC combinations are possible *Preferred uniqueness:* ORM 2 enables display of preferred uniqueness constraints on *n*-aries to be toggled on/off.	or P P
Role Connection A Role is played only by *A*	A
External Uniqueness Constraints A B_1 B_n Each B_1,…, B_n combination ($n > 1$) relates to only one instance of *A* *Preferred uniqueness:* A B_1 B_n	A B_1 B_n U A B_1 B_n P

ORM 2	ORM 1
Mandatory Role Constraints *Simple:* Role is mandatory for population of *A* *Disjunctive (inclusive-or constraint):* Each instance in the population of *A* plays at least one of the *n* attached roles (*n* > 1). Role numbers are not displayed.	
Objectification etc. Fact type is objectified as object type *A*. ORM 2 allows any fact type to be objectified.	ORM 1 does not support objectified unaries. It allows objectification only if for a spanning UC or a 1:1 pattern.
Object Value Constraints *Enumeration* *Range* *Semibounded discrete range* { a.. } { ..a } *Bounded continuous range* {[a_1 .. a_2]} *includes both end values* {(a_1 .. a_2)} *excludes both end values* {[a_1 .. a_2)} *includes first value* {(a_1 .. a_2]} *includes last value* Combinations are allowed.	 { a.. } { ..a } {a_1..a_2} ORM1 does not support exclusion of any end values Combinations are allowed.
Role Value Constraints Same patterns as above	Not supported

ORM 2	ORM 1
Subset Constraints *Simple:* Each object that plays role 1 also plays role 2	Same
Contiguous Role-pair: Each object pair that plays the role sequence 1.1, 1.2 also plays the role sequence 2.1, 2.2	Same
Other cases: Each object tuple that plays the role sequence 1.1, 1.2, 1.3 also plays the role sequence 2.1, 2.2, 2.3	Same
ORM 2 also displays subset constraints over join paths	ORM 1 does not display subset constraints over join paths
Equality Constraints *2 role-sequences (of 1 or more roles):* Populations of role-sequences must be equal	Same
3 or more role-sequences: e.g. 	Not supported (instead use multiple 2-sequence constraints)
Exclusion Constraints Populations of 2 or more role-sequences must be mutually exclusive	Same

ORM 2	ORM 1
Exclusive-Or Constraints Each instance in *A*'s population plays exactly one of the *n* attached roles (*n* > 1)	Same
Subtyping *B* is a proper subtype of *A and C. A (but not C) provides a path to the preferred identifier for A.* *Exclusive* *Total* *Partition*	*B* is a proper subtype of *A* (its primary supertype) and *C* (a secondary supertype) ORM 1 does not display constraints over subtyping connections
Frequency Constraints Each instance that plays role 1 does so *f* times Each instance pair that plays roles 1, 2 does so *f* times Each instance pair that plays roles 1, 2 does so *f* times The frequency specification *f* may be any of the following *n* exactly *n* (a positive integer) ≥ *n* at least *n* ≤ *n* at most *n* *n..m* at least *n* and at most *m*	Same Same *n* >=*n* <=*n* n..m

ORM 2	ORM 1
Ring Constraints A *Irreflexive* *Asymmetric* *Intransitive* *Antisymmetric* *Acyclic* *Asymmetric + Intransitive* *Acyclic + Intransitive* *Symmetric* *Symmetric + Irreflexive* *Symmetric + Intransitive* *Purely Reflexive*	A °ir °as °it °ans °ac °(as,it) °(ac,it) °sym °(ir,sym) °(it,sym) Not supported
Value-comparison Constraints	Not supported
Derived Fact Types * = derived, ** = derived and stored + = semi-derived	Same for first two options. 3rd option not supported.
Deontic Constraints Colored blue rather than violet. Most include "o" for "obligatory". Deontic ring constraints instead use dashed lines. *Uniqueness* *Mandatory* *Subset, Equality, Exclusion* *Frequency* °f *Irreflexive* *Acyclic* *Asymmetric* *Asym-Intrans* *Intransitive* *Acyclic-Intrans* *Antisymmetric* *Symmetric* *Purely Reflexive* *etc.*	No deontic constraints are supported

	ORM 2	ORM 1
Object Cardinality Constraints		
# = *n* A	Each population of *A* includes exactly *n* instances	
# ≤ *n* A	Each population of *A* includes at most *n* instances	Not supported
Role Cardinality Constraints		
A # = *n* R	Each population of *R* includes exactly *n* instances	
A # ≤ *n* R	Each population of *R* includes at most *n* instances	Not supported

Textual Constraints (ORM 2 example)

[1] **For each** Employee, birthdate < hiredate.
[2] **Each** Employee **who** has Rank 'NonExec' uses **at most one** CompanyCar.
[3] **Each** Employee **who** has Rank 'Exec' uses **some** CompanyCar.

Constraint Verbalization (ORM 2 example)

C_1: **Each** Person works for **at most one** Company.
C_2: **Each** Person works for **some** Company.
C_3: **Each** Person heads **at most one** Company.
C_4: **Each** Company is headed by **at most one** Person.
C_5: **Each** Person **who** heads **some** Company **also** works for **that** Company.

The absence of a UC on the top righthand role verbalizes as

It is possible that more than one Person works for **the same** Company.

ER Glossary

This glossary lists the key symbols used in the Barker ER notation, the Information Engineering notation for ER, and the IDEF1X notation, as discussed in Chapter 8.

Barker ER notation

Information Engineering notation

IDEF1X notation

UML Glossary

This glossary lists key terms and symbols used in UML data models (*class diagrams*), as discussed in Chapter 9.

Aggregate: Class that is the "whole" in a whole-part relationship.

Association: Relationship type, involving two or more association roles.

Association class: An association that is also a class; an objectified association in ORM.

Association end: Corresponds to a role in ORM.

Attribute: Property of an object; either single-valued or multivalued.

Changeability: How an attribute or role value may change: unrestricted, readOnly, addOnly, removeOnly.

Class: Type of object (cf. ORM entity type).

Collection property string: Annotation to an attribute or association role indicating the kind of collection to be used. For example, {ordered}, {nonunique}, and {ordered, nonunique} specify an ordered set, bag, and sequence, respectively.

Composition (*composite aggregation*): Each part belongs to at most one whole.

Data value: Scalar constant; data type instance; self-identifying (cf. ORM value).

Derived element: An attribute or association that is computed or inferred from others.

Generalization: Forming a more general class (a superclass) from one or more classes.

Link: Relationship instance—a fact in ORM.

Multiplicity: Number of instances to which the same object may relate in an attribute or association.

Object: A thing that may typically change its state; a class instance; identified by an oid (cf. ORM entity instance).

Oid: Object identifier; system generated constant that references an object.

Qualified association: Association where one class relates to another via a qualifier.

Shared aggregation: A part may belong to more than one whole.

Specialization: Forming one or more specific classes (subclasses) from a more general class.

State: The properties of an object at a point in time.

Subset constraint: Restricts the population of one association to be a subset of another.

Xor constraint: Exclusive-or constraint between roles (association ends).

Multiplicities:

*	0 or more
0..1	0 or 1
1..*	1 or more
1	exactly 1
n..m	at least *n* and at most *m*

Each *A* plays with exactly one *B*.
Each *B* plays with 0 or more *A*'s

default attribute multiplicity = 1

default association role multiplicity = *

Ternary association

Association class

Qualified association

Exclusive-or constraint

Each *A* plays with *B or C* but not both

Subset constraint

pop(S) is a subset of pop(R)

Subtyping

Authors are *ordered*

This page intentionally left blank

Useful Web Sites

www.mkp.com/imrd/. Morgan Kaufmann Web site for this book: includes online appendices.

Fact-Oriented Modeling (General)

www.ormfoundation.org. ORM Foundation site.
www.orm.net. Terry Halpin's ORM site.
www.erikproper.eu/publications/. Erik Proper's publications site.
www.inconcept.com/JCM/index.html. InConcept's Journal of Conceptual Modeling.
www.ormcentral.com. John Miller's ORM site: includes COM API details for Visio ORM.
www.pna-group.com. PNA Group: Dutch consultancy for fact-oriented modeling.
www.starlab.vub.ac.be/website/. STARLab group at Free University of Brussels.

Fact-Oriented Modeling Tools

NORMA (ORM 2 tool). www.ormfoundation.org or http://sourceforge.net/projects/orm).
Microsoft VisioModeler (free ORM 1 tool). www.microsoft.com/downloads/.
Microsoft Visio E.A. (ORM 1 tool). http://msdn2.microsoft.com/en-us/vstudio/aa718657.aspx.
CaseTalk (FCO-IM tool). www.casetalk.com/php/.
Infagon (free FCO-IM tool). www.mattic.com/.
Doctool and CogNIAM (CogNIAM tools). www.pna-group.com.
ActiveFacts (ORM 2 tools). http://dataconstellation.com/ActiveFacts/.
DogmaStudio (ORM Ontology tool). www.starlab.vub.ac.be/website/tools.
Orthogonal Toolbox (free XML add-on to database modeling COM API for Microsoft's ORM solution). www.orthogonalsoftware.com/products.html.

Business Rules and Architecture Frameworks

www.brcommunity.com/index.php. Business Rules Journal.
www.zifa.com/. Zachman Framework.
www.opengroup.org/togaf/. The Open Group Architecture Framework.
www.liacs.nl/~verrynst/frisco.html. The FRISCO Report.

SQL, XML, RDF, and OWL

www.wiscorp.com/SQLStandards.html. Free downloads of various SQL Standards etc.
http://sqlzoo.net. Basic tutorials on SQL, XML etc.
www.w3.org/RDF/. W3C Web site on RDF (Resource Description Framework).
www.w3.org/2004/OWL/. W3C Web site on OWL (Web Ontology Language).
www.iso.org. The latest version of the ISO SQL standard (ISO/IEC 9075).

UML, OCL, MDA, and SBVR

www.uml.org/. The OMG's Web site on UML.
www-st.inf.tu-dresden.de/ocl/. Portal on the Object Constraint Language.
www.omg.org/mda/. Model Driven Architecture (MDA).
http://omg.org/technology/documents/bms_spec_catalog.htm#SBVR OMG Web site on SBVR.
http://en.wikipedia.org/wiki/Semantics_of_Business_Vocabulary_and_Rules. Wikipedia article on SBVR.

Workflow Modeling

www.yawl-system.com/. Free YAWL tool and documents on workflow modeling.
www.workflowpatterns.com. A compendium of resources on the well-known workflow patterns.
www.pi-workflow.org. Workflow patterns from a pi-calculus viewpoint.
www.oasis-open.org. The home of the BPEL standard.
www.omg.org. The latest information on the BPMN and BPDM standards can be found here.

Information Modeling Organizations

www.dama.org/. DAMA International.
http://home.dei.polimi.it/pernici/ifip81/team.html. IFIP WG 8.1.
www.emmsad.org/index.htm. EMMSAD workshop site.

Bibliography

Entries marked with an asterisk "*" are accessible online at *www.orm.net*.

Abrial, J. 1974, 'Data Semantics', *Data Base Management*, eds J. Klimbie and K. Koffeman, North-Holland, Amsterdam, pp. 1-60.

Aho, A., Beeri, C. and Ullman, J. 1979, 'The Theory of Joins in Relational Databases', *ACM Trans. on Database Systems*, vol. 4, no. 3, pp. 297–314.

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrick, K., Maier, D. and Zdonik, S., 'The Object-Oriented Database System Manifesto', *Proc. DOOD-89: First Int. Conf. on Deductive and Object-Oriented Databases*, eds, W. Kim, J-M. Nicolas and S. Nishio, Elsevier, pp. 40–57.

Baader, F., Calvanese, D., McGuinness, D., Nardi, D. and Patel-Schneider, P. (eds) 2003, *The Description Logic Handbook*, Cambridge University Press.

Bakema, G., Zwart, J. and van der Lek, H. 1994, 'Fully Communication Oriented NIAM', *NIAM-ISDM 1994 Conf. Working Papers*, eds G.M. Nijssen and J. Sharp, Albuquerque, NM, pp. L1–35.

Balsters, H., Carver, A., Halpin, T. and Morgan, T. 2006, 'Modeling Dynamic Rules in ORM', *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, eds. R. Meersman, Z. Tari, P. Herrero et al., Montpellier. Springer LNCS 4278, pp. 1201–10.

Barbier, F., Henderson-Sellers, B., Opdahl, A. and Gogolla, M. 2000, 'The Whole-Part Relationship in UML: A New Approach', *Unified Modeling Language: Systems Analysis, Design and Development Issues*, eds K. Siau and T. Halpin, Idea Group, Hershey, PA.

Barker, R. 1990a, *CASE*Method: Entity Relationship Modelling*, Addison-Wesley, Wokingham.

Barker, R. 1990b, *CASE*Method: Tasks and Deliverables*, Addison-Wesley, Wokingham.

Barker, R. and Longman, C. 1992, *CASE*Method: Function and Process Modelling*, Addison-Wesley, Wokingham.

Batini, C., Ceri, S. and Navathe, S. 1992, *Conceptual Database Design: An Entity-Relationship Approach*, Benjamin Cummings, Redwood City, CA.

Bennett, S., McRobb, S. and Farmer, R. 2006, *Object-Oriented Systems Analysis and Design, Third Edtion*, McGraw Hill, Berkshire.

Bentley, J. 1998, 'Little Languages', *More Programming Pearls*, Addison-Wesley, Reading MA.

Berners-Lee, T., Hendler, J. and Lassila, O. 2001, 'The Semantic Web', *Scientific American*, May 2001.

Bird, L., Goodchild, A. and Halpin, T. 2000, 'Object-Role Modelling and XML Schema', *Conceptual Modeling – ER2000*, Proc. 19th ER Conf., Springer LNCS vol. 1920, pp. 309–22.

Booch, G., Rumbaugh, J. and Jacobson, I. 1999, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading MA.

Blaha, M. and Premerlani, W. 1998, *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, New Jersey.

Bloesch, A. and Halpin, T. 1996, 'ConQuer: A Conceptual Query Language', *Proc. ER'96: 15th Int. Conf. on Conceptual Modeling*, Springer LNCS, no. 1157, pp. 121–33.*

Bloesch, A. and Halpin, T. 1997, 'Conceptual Queries Using ConQuer-II', *Proc. ER'97: 16th Int. Conf. on Conceptual Modeling*, Springer LNCS, no. 1331, pp. 113–26.*

Bruce, T. 1992, *Designing Quality Databases: Practical Information Management and IDEF1X*, Dorset House, New York.

Bubenko, J. 2007, 'From Information Algebra to Enterprise Modelling and Ontologies—a Historical Perspective on Modelling for Information Systems', in *Conceptual Modelling in Information Systems Engineering*, eds J. Krogstie, A, Opdahl and S. Brinkkemper, Springer, Berlin, pp. 1–18.

Burlton, R. 2001, *Business Process Management: Profiting From Process*, Sams Publishing.

Campbell, L. and Halpin, T. 1993, 'Automated Support for Conceptual to External Mapping', *Proc. 4th Workshop on Next Generation CASE Tools*, eds S. Brinkkemper and F. Harmsen, Univ. Twente Memoranda Informatica 93–32, pp. 35–51.

Campbell, L. and Halpin, T. 1994a, 'Abstraction Techniques for Conceptual Schemas', *Proc. 5th Australasian Database Conf.*, Christchurch (Jan.), World Scientific, Singapore.

Campbell, L. and Halpin, T. 1994b, 'The Reverse Engineering of Relational Databases', *Proc. 5th Workshop on Next Generation CASE Tools*, Utrecht (June).

Campbell, L., Halpin, T. and Proper, H.1996 'Conceptual Schemas with Abstractions: Making Flat Conceptual Schemas More Comprehensible', *Data and Knowledge Engineering,* vol. 20, no. 1, pp. 39–85.*

Carver, A. and Halpin, T. 2008, 'Semantic Normalization', Neumont University Technical Report.

Cattell, R. 1991, *Object Data Management*, Addison-Wesley, Reading MA.

Cattell, R. and Barry, D. (eds) 2000, *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann, San Francisco.

Celko, J. 1999, *Data and Databases: Concepts in Practice*, Morgan Kaufmann, San Francisco.

Celko, J. 2000, *Joe Celko's SQL for Smarties: advanced SQL programming*, 2nd edn, Morgan Kaufmann, San Francisco.

Celko, J. 2004, *Joe Celko's Trees and Hierarchies in SQL for Smarties*, Morgan Kaufmann, San Francisco.

Celko, J. 2005, *Joe Celko's SQL Programming Style*, Morgan Kaufmann, San Francisco.

Chamberlin, D. 1998, *A Complete Guide to DB2 Universal Database*, Morgan Kaufmann, San Francisco.

Chamberlin, D. and Boyce, R. 1974, 'SEQUEL: A Structured English Query Language', *Proc. 1974 ACM SIGMOD Workshop on Data Description, Access and Control*, Ann Arbor, Michigan.

Chawdhry, P. 1992, 'NIAMEX—A NIAM Compiler and EXPRESS Pre-processor', *Proc. EXPRESS Users Group Conf.,* Oct. 1992.

Chen, P. 1976, 'The Entity-Relationship Model—Toward a Unified View of Data', *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9–36.

Chong, R., Wang, X., Dang, M. and Snow D. 2008, *Understanding DB2: Learning Visually with Examples*, 2nd edn, IBM Press.

Choobineh, J., Mannino, M. and Tseng, V. 1992, 'A Form-based Approach for Database Analysis and Design', *CACM*, vol. 35, no. 2, pp. 108–20.

Codd, E. 1969, 'Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks', *IBM Research Report* RJ599, August 19, 1969.

Codd, E. 1970, 'A Relational Model of Data for Large Shared Data Banks', *CACM*, vol. 13, no. 6, pp. 377–87.

Codd, E. 1990, *The Relational Model for Database Management: Version 2*, Addison-Wesley, Reading MA.

Codd, E., Codd, S. and Salley, C. 1993, 'Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate', Arbor Software Corporation. Accessible online at *www.arborsoft.com/essbase/wht_ppr/coddToc.html*.

Connolly, T., Begg, C. and Strachan, A. 1999, *Database Systems*, 2nd edn, Addison-Wesley.

Control Data 1982, *IAST: Information Analysis Support Tools*, Reference Manual, Control Data Publication no. 60484610.

Creasy, P. 1989, 'ENIAM—A More Complete Conceptual Schema Language', *Proc. 15th VLDB Conf.*, Amsterdam.

Cuyler, D. and Halpin, T. 2003, 'Metamodels for Object-Role Modeling', *Proc. EMMSAD'03: 8th IFIP WG8.1 Int. Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*, Velden, Austria (June).

Czejdo, B., Elmasri, R., Rusinkiewicz, M. and Embley, D. 1990, 'A Graphical Data Manipulation Language for an Extended Entity-Relationship Model', *IEEE Computer*, March 1990, pp. 26–37.

D'Atri, A. and Sacca, D. 1984, 'Equivalence and Mapping of Database Schemas', *Proc. 10th Int. conf. on Very Large Databases*, VLDB, Singapore, pp. 187–195.

Date, C. 1998, 'The Birth of the Relational Model: Parts 1–3', *Intelligent Enterprise*, vol. 1, Oct-Dec issues, Miller Freeman, Inc. San Mateo CA.

Date, C. 2000, *An Introduction to Database Systems*, vol. 1, 7th edn, Addison-Wesley, Reading MA.

Date, C. and Darwen, H. 1997, *A Guide to the SQL Standard*, 4th edn, Addison-Wesley, Reading MA.

Date, C. and Darwen, H. 1998, *Foundation for Object/Relational Databases: the Third Manifesto*, Addison-Wesley, Reading MA.

Date, C. and Fagin, R. 1992, 'Simple Conditions for Guaranteeing Higher Normal Forms in Relational Databases', *ACM TODS*, vol. 17, no. 3, pp. 465–476.

Delaney, K. 2000, 'Introducing Indexed Views', *SQL Server Magazine*, no. 15, Duke Communications, www.sqlmag.com. pp. 32–37.

Denning, P. and Metcalfe, R. (eds) 1997, *Beyond Calculation: The Next Fifty Years of Computing*, Springer-Verlag, New York.

De Troyer, O. 1991, 'The OO-Binary Relationship Model: A Truly Object-Oriented Conceptual Model', *Advanced Information Systems Engineering: Proc. CAiSE-91*, Springer LNCS, no. 498, Trondheim.

De Troyer, O. 1993, *On Data Schema Transformation*, PhD thesis, Katholieke Universiteit of Brabant, Tilburg.

De Troyer, O., Meersman, R. and Verlinden, P. 1988, 'RIDL* on the CRIS Case: A Workbench for NIAM', *Computerized Assistance during the Information Systems Life Cycle: Proc. CRIS-88*, eds T. Olle, A. Verrijn-Stuart and L. Bhabuta, North-Holland, Amsterdam.

De Troyer, O. and Meersman, R. 1995, 'A Logic Framework for a Semantics of Object Oriented Data Modeling', *Proc. OOER'95: Object-Oriented and Entity-Relationship Modeling*, Springer LNCS, no. 1021, Gold Coast Australia, pp. 238-49.

Dietz, J. and Halpin, T. 2003, 'Combining DEMO and ORM—An Investigation of Mutual Benefits', *Proc. EMMSAD'03*: 8th IFIP WG8.1 Int. Workshop on Evaluation of Modeling Methods in Systems Analysis and Design, Velden, Austria (June).

Dix, A., Finlay, J., Abowd, G. and Beale, R. 1993, *Human-Computer Interaction*, Prentice Hall, New York.

DuBois, P. 2006, *MySQL Cookbook*, 2nd edn, O'Reilly Media, Inc.

Elmasri, R. and Navathe, S. 1994, *Fundamentals of Database Systems*, 2nd edn, Benjamin/Cummings, Redwood City, CA.

Embley, D., Kurtz, B. and Woodfield, S. 1992, *Object-Oriented Systems Analysis*, Prentice Hall, Englewood Cliffs, NJ.

Embley, D. 1998, *Object Database Management*, Addison-Wesley, Reading, MA.

Embley, D. Wu, H., Pinkston, J. and Czejdo, B. 1996, 'OSM-QL: A Calculus-based Graphical Query Language', Tech. Report, Dept of Comp. Science, Brigham Young Univ., Utah.

Eriksson, H. and Penker, M. 2000, *Business Modeling with UML*, John Wiley.

Evans, C. 1980, *The Mighty Micro*, Coronet Books, London.

Everest, G. 1976, 'Basic Data Structure Models Explained with a Common Example', *Proc. Fifth Texas Conference on Computing Systems*, Austin TX, IEEE Computer Society Publications, Long Beach, CA, pp. 39-45.

Everest, G. 1994, 'Experiences Teaching NIAM/OR Modeling', *Proc. 2nd NIAM-ISDM Conf.*, Albuquerque, NM.

Fagin, R. 1977, 'Multivalued Dependencies and a New Normal Form for Relational Databases', *ACM Trans. on Database Systems*, vol. 2, no. 3.

Fagin, R. 1979, 'Normal Forms and Relational Database Operators', *Proc. 1979 ACM SIGMOD int. Conf. on Management of Data*, Boston.

Fagin, R. 1981, 'A Normal Form for Relational Databases that is based on Domains and Keys', *ACM Trans. on Database Systems*, vol. 6, no. 3, pp. 387–415.

Falkenberg, E. 1976, 'Concepts for Modelling Information', *Modelling in Database Management Systems*, ed. G. Nijssen, North-Holland Publishing, Amsterdam.

Falkenberg, E. 1993, 'DETERM: Deterministic Event-tuned Entity-Relationship Modeling', *Entity-Relationship Approach – ER'93*, Springer LNCS no. 823, pp. 230–41.

Falkenberg, E. and Oei, J. 1994, 'Meta-model Hierarchies from an Object-Role Modeling Perspective', *Proc. First Int. Conf. On Object-Role Modeling (ORM-1)*, eds T. Halpin and R. Meersman, Magnetic Island, Australia, pp. 218–227.

Falkenberg, E. Hesse, W., Lindgreen, P., Nilson, B., Oei, J., Rolland, C., Stamper, R., Van Assche, F., Verrijn-Stuart, A. and Voss, K. 1998, *A Framework of Information System Concepts: the FRISCO Report* (Web edition), IFIP, Available online by anonymous *ftp://ftp.leidenuniv.nl/pub/rul/fri-full.zip*.

Fillmore, C. 1968, "The Case for Case", in *Universals in Linguistic Theory*, eds E. Bach and R. Harms, Holt, Rinehart and Winston, New York, pp. 1–88.

Finkelstein, C. 1989, *Introduction to Information Engineering*, Addison-Wesley, Reading MA.

Finkelstein, C. 1992, *Information Engineering: Strategic Systems Development*, Addison-Wesley, Reading MA.

Finkelstein, C. 1998, 'Information Engineering Methodology', *Handbook on Architectures of Information Systems*, eds. P. Bernus, K. Mertins and G. Schmidt, Springer-Verlag, Berlin, Germany, pp. 405–27.

Finkelstein, C. and Aiken, P. 2000, *Building Corporate Portals with XML*, McGraw-Hill, New York.

Fitting, M. 2000, 'Databases and Higher Types'. In John Lloyd et. al. (ed.), *Computational Logic–CL2000*, (LNAI 1861, pp 41-52). Springer-Verlag.

Fowler, M. with Scott, K. 1997, *UML Distilled*, Addison-Wesley.

Gorman, M. 1998, 'Great News, The Relational Data Model is Dead!', URL: *http://www.wiscorp.com/SQLStandards.html#sql1999readings*.

Greenwald, R., Stackowiak, R. and Stern, J. 2007, *Oracle Essentials: Oracle Database 11g*, O'Reilly Media, Inc.

Guarino, N. and Welty, C. 2002, 'Evaluating Ontological Decisions with OntoClean', *CACM*, vol. 45, no. 2, pp. 61–65.

Guizzardi, G., Wagner, G., Guarino, N. and van Sinderen, N. 2004, 'An Ontologically Well-Founded Profile for UML Conceptual Models, *Proc. 16th Int. Conf. on Advanced Inf. Sys. Engineering, CAiSE2004*, eds. A. Persson and J. Stirna. Springer LNCS 3084, pp. 112–26.

Guizzardi, G. 2005, Ontological Foundations for Structural Conceptual Models, CTIT PhD Thesis Series, No. 05-74, Enschede, The Netherlands.

Haack, S. 1978, *Philosophy of Logics*, Cambridge University Press, London.

Habrias, H. 1993, 'Normalized Object Oriented Method', in *Encyclopedia of Microcomputers*, vol. 12, Marcel Dekker, New York, pp. 271–85.

Halpin, T. 1989a, 'Venn Diagrams and SQL Queries', *The Australian Computer Journal*, vol. 21, no. 1, pp. 27-32.

Halpin, T. 1989b, 'A Logical Analysis of Information Systems: Static Aspects of the Data-Oriented Perspective', PhD thesis, University of Queensland.

Halpin, T. 1989c, 'Contextual Equivalence of Conceptual Schemas', *Proc. Advanced Database Systems Symposium*, Inform. Proc. Society of Japan, Kyoto, pp. 47-54.

Halpin, T. 1990a, 'Conceptual Schemas and Relational Databases', *Databases in the 1990s: Proc. 1st Australian Database Conf.*, eds B. Srinivasan and J. Zeleznikov, World Scientific, Singapore, pp. 45–56.

Halpin, T. 1990b, 'Conceptual Schema Optimization', *Proc. 13th Australian Computer Science Conf.*, Monash University, Melbourne.

Halpin, T. 1991a, 'Optimizing Global Conceptual Schemas', *Databases in the 1990s:2–Proc. 2nd Australian Database Conf.*, eds B. Srinivasan and J. Zeleznikov, World Scientific, Singapore.

Halpin, T. 1991b, 'A Fact-Oriented Approach to Schema Transformation', *Proc. MFDBS-91 Int. Conf. on Mathematical Fundamentals of Database and Knowledge Base Systems*, Spinger LNCS, no. 495, Rostock.

Halpin, T. 1991c, 'WISE: a Workbench for Information System Engineering', *Proc. 2nd European Workshop on Next Generation of CASE Tools*, Trondheim. Revised and reprinted in *Next Generation CASE Tools*, IOS Press (1992).

Halpin, T. 1993, 'What is an Elementary Fact?', *Proc. 1st NIAM-ISDM Conf.*, Utrecht.*

Halpin. T. 1995, 'Subtyping: Conceptual and Logical Issues', *Database Newsletter*, ed. R. Ross, Database Research Group Inc., vol. 23, no. 6, pp. 3–9.*

Halpin, T. 1996, 'Business Rules and Object-Role Modeling', *Database Programming and Design*, vol. 9, no. 10 (Oct. 1996), pp. 66–72.*

Halpin, T. 1998a, 'Conceptual Queries', *Database Newsletter*, vol. 26, no. 2, ed. R. Ross, Database Research Group, Inc., Boston (March/April 1998).*

Halpin, T. 1998–99, 'UML Data Models from an ORM Perspective: Parts 1–10', *Journal of Conceptual Modeling*, InConcept, Minneapolis USA.*

Halpin, T. 1999a, 'Object Role Modeling: an overview', *www.orm.net*.*

Halpin, T. 1999b, 'Data Modeling in UML and ORM Revisited', *Proc. EMMSAD'99: 4th IFIP WG8.1 Int. Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*, Heidelberg, Germany (June).*

Halpin, T. 2000a, 'Integrating Fact-Oriented Modeling with Object-Oriented Modeling', *Information Modeling for the new Millenium*, eds K. Siau and M. Rossi, Idea Group, Hershey, PA.

Halpin, T. 2000b, 'Modeling Collections in UML and ORM', *Proc. EMMSAD'00: 5th Int. Workshop on Evaluation of Modeling Methods in Sys. Analysis and Design*, Kista, Sweden.*

Halpin, T.2000c, 'Supplementing UML with Concepts from ORM', *Unified Modeling Language: Systems Analysis, Design and Development Issues*, eds K. Siau and T. Halpin, Idea Group, Hershey, PA.

Halpin, T. 2000d, 'An ORM Metamodel', *Journal of Conceptual Modeling*, no. 16, InConcept, Minneapolis, MN.*

Halpin, T. 2000e, 'An ORM Metamodel of Barker ER', *Journal of Conceptual Modeling*, no. 17, InConcept, Minneapolis, MN.*

Halpin, T. 2002a, 'Join Constraints', *Proc. Seventh CAiSE/IFIP-WG8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*, eds. T. Halpin, J. Krogstie and K. Siau, Toronto, Canada, pp. 121–31.

Halpin, T. and Wagner, G. 2003, 'Modeling Reactive Behavior in ORM'. *Conceptual Modeling – ER2003*, Proc. 22nd ER Conference, Chicago, October 2003, Springer LNCS.

Halpin, T. 2004a, 'Information Modeling and Higher-Order Types', *Proc. CAiSE'04 Workshops*, vol. 1, (eds J. Grundspenkis and M. Kirkova), Riga Tech. University, pp. 233–48.

Halpin, T. 2004b, 'Comparing Metamodels for ER, ORM and UML Data Models', *Advanced Topics in Database Research, vol. 3*, ed. K. Siau, Idea Group, Hershey PA, pp. 23–44.

Halpin, T. 2005a. 'Objectification', *Proc. CAiSE'05 Workshops*, vol. 1, eds J. Castro and E. Teniente, FEUP, pp. 519–532.

Halpin, T. 2005b, 'ORM 2', *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops*, eds. R. Meersman, Z. Tari, P. Herrero et al., Cyprus, Springer LNCS 3762, pp. 676–87.

Halpin, T. 2005c, 'Constraints on Conceptual Join Paths', *Information Modeling Methods and Methodologies,* eds J. Krogstie, T. Halpin, T.A. and K. Siau, Idea Group, Hershey PA, pp. 258–77.

Halpin, T. 2005d, 'Higher-Order Types and Information Modeling', *Advanced Topics in Database Research, vol. 4*, ed. K. Siau, Idea Publishing Group, Hershey PA, USA, Ch. X (pp. 218–37).

Halpin, T. 2005e, 'Information Modeling in UML and ORM: A Comparison', *Encyclopedia of Information Science and Technology*, vol. 3, ed. M. Khosrow-Pour, Idea Group, Hershey PA, pp. 1471–75.

Halpin, T. 2006a, 'Business Rule Modality', Proc. CAiSE'06 Workshops, eds T. Latour and M. Petit, Namur University Press, pp. 383–94.

Halpin, T. 2006b, 'Object-Role Modeling (ORM/NIAM)', *Handbook on Architectures of Information Systems*, *2nd edition*, Springer, Heidelberg, pp. 81–103.

Halpin, T. 2007a, 'Modality of Business Rules', *Research Issues in Systems Analysis and Design, Databases and Software Development*, ed. K. Siau, IGI Global, Hershey, PA, pp. 206–26.

Halpin, T. 2007b, 'Subtyping Revisited', *Proc. CAiSE'07 Workshops*, vol. 1, eds. B. Pernici and J. Gulla, Tapir Academic Press, pp. 131–141.

Halpin, T., Carver, A. and Owen, K. 2007, 'Reduction Transformations in ORM', *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, eds. R. Meersman, Z. Tari, P. Herrero et al., Vilamoura, Springer LNCS 4805, pp. 699–708.

Halpin, T. 2008, 'Objectification of Relationships', *Advanced Topics in Database Research,* ed. K. Siau, IGI Global, Hershey, PA, (in press).

Halpin, T. and Bloesch, A. 1999, 'Data Modeling in UML and ORM: A Comparison', *Journal of Database Management*, vol. 10, no. 4, Idea Group, Hershey, PA, pp. 4–13.*

Halpin, T., Evans, K, Hallock, P. and MacLean, W. 2003, *Database Modeling with Microsoft® Visio for Enterprise Architects*, Morgan Kaufmann, San Francisco.

Halpin, T. and Girle, R. 1981, *Deductive Logic*, 2nd edn, Logiqpress, Brisbane.

Halpin, T. and Proper, H. 1995a, 'Database Schema Transformation and Optimization', *Proc. OOER'95*, ed. M. Papazoglou, Springer LNCS, no. 1021, pp. 191–203.*

Halpin, T. and Proper, H. 1995b, 'Subtyping and Polymorphism in Object-Role Modeling', *Data and Knowledge Engineering*, vol. 15, pp. 251–81.*

Halpin, T. and Ritson, P. 1992, 'Fact-Oriented Modelling and Null Values', *Proc. 3rd Australian Database Conf.*, eds. B. Srinivasan and J. Zeleznikov, World Scientific, Singapore.

Halpin, T. and Ritson, P. 1996, 'Entity Integrity Revisited', *The Australian Computer Journal*, vol. 28, no. 3, August 1996, pp. 73–80.

Halpin, T. and Vermeir, D. 1997, 'Default Reasoning in Information Systems', *Database Applications Semantics*, Chapman and Hall, London, pp. 423–41.

Hammer, M. 1997, *Beyond Reengineering: How the Process-Centered Organization is Changing Our Work and Our Lives*, Collins.

Hammer, M. and McLeod, D. 1981, 'Database Description with SDM: A Semantic Database Model', *ACM Transactions on Database Systems*, vol. 6, pp. 351–86.

Harel, D. and Politi, M. 1998, *Modeling Reactive Systems With Statecharts: The Statemate Approach*, McGraw-Hill.

Harris, R. 1996, *Information Graphics: A Comprehensive Illustrated Reference*, Management Graphics, Atlanta, GA.

Havey, M. 2005, *Essential Business Process Modeling*, O'Reilly Media, Inc.

Hay, D. 1996, *Data Model Patterns: Conventions of Thought*, Dorset House, New York.

Hay, D. 1999a, 'There Is No Object-Oriented Analysis', *DataToKnowledge Newsletter*, vol. 27, no. 1, Business Rule Solutions, Inc., Houston TX.

Hay, D. 1999b, 'Object Orientation and Information Engineering: UML', *The Data Administration Newsletter*, no. 9, (June 1999), ed. R. Reiner, available online at *www.tdan.com*.

Hay, D. 2003, *Requirements Analysis*, Pearson Education, New Jersey.

Hay, D. 2006, *Data Model Patterns: A Metadata Map*, Morgan Kaufmann.

Heath, I. 1971, 'Unacceptable File Operations in a Relational Database', *Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control*, San Diego, CA.

Hohenstein, U. and Engels, G. 1991, 'Formal Semantics of an Entity-Relationship-Based Query Language', *Proc. 9th ER Conf.*, ed. H. Kangassalo, Elsevier Science, Amsterdam.

IEEE 1999, *IEEE standard for conceptual modeling language syntax and semantics for* $IDEF1X_{97}$ *(*$IDEF_{object}$*)*, IEEE Std 1320.2–1998, IEEE, New York.

Inmon, W. 1993, *Building the Data Warehouse*, John Wiley and Sons, New York.

Intellibase 1990, *RIDL-M User's Guide*, Intellibase N.V., Belgium.

Jacobson, I., Booch, G. and Rumbaugh, J. 1999, *The Unified Software Development Process*, Addison-Wesley, Reading, MA.

Jarke M., Mylopoulos, J., Schmidt, J. and Vassiliou, Y. 1992, 'DAIDA: An Environment for Evolving Information Systems', *ACM Trans. on Inform. Systems*, vol. 10, no. 1, pp. 1–50.

Jarrar, M. 2007, 'Mapping ORM into the SHOIN/OWL Description Logic', *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, eds. R. Meersman, Z. Tari, P. Herrero et al., Vilamoura, Springer LNCS 4805.

Jeyaraj, A. and Sauter, V. 2007, 'An Empirical Investigation of the Effectiveness of Systems Modeling and Verification Tools', *CACM*, vol. 50, no. 6, pp. 63–67.

Jukik, N. 2006, 'Modeling Strategies and Alternatives for Data Warehousing Projects', *CACM*, vol. 49, no. 4, pp. 83–88.

Keet, C. 2006, 'Part-Whole Relations in Object-Role Models', *Proc. OTM Workshops 2006*, Springer LNCS 4278, pp. 1118–27.

Keet, C. 2007, 'Mapping the Object-Role Modeling language ORM2 into Description Logic Language DLR_{ifd}', KRDB Research Centre TR KRDB07–2, Faculty of Computer Science, Free University of Bozen-Bolzano, Italy.

Kendall, K. and Kendall, J. 1988, *Systems Analysis and Design*, Prentice Hall, Englewood Cliffs, NJ.

Kennedy, P. 1993, *Preparing for the Twenty-first Century*, Harper Collins, London.

Kent, W. 1977, 'Entities and Relationships in Information', *Proc. 1977 IFIP Working Conf. on Modelling in Data Base Management Systems*, ed. G. Nijssen, Nice, France, North-Holland Publishing, pp. 67–91.

Kent, W. 1983, 'A Simple Guide to Five Normal Forms in Relational Database Theory', *CACM*, vol. 26, no. 2, pp 120–5.

Kent, W. 1991, 'A Rigorous Model of Object Reference, Identity, and Existence', *Jnl of Object-Oriented Programming*, June 1991, pp. 28–36.

Kent, W. 2000, *Data and Reality, 2nd Edition*, The 1stBooks Library.

Khoshaflan, S. and Copeland, G. 1990, 'Object Identity', in *Readings in Object-Oriented Database Systems*, eds S. Zdonik and D. Maier, Morgan Kaufmann, San Mateo CA, pp. 37–46.

Kobayashi, I. 1986, 'Losslessness and Semantic Correctness of Database Schema Transformation: Another Look at Schema Equivalence', *Information Systems*, vol. 11, no. 1, Pergamon Press, pp. 41–59.

Kobayashi, I. 1990, 'Transformation and Equivalence among Predicate Systems', *Proc. French-Japanese Seminar on Deductive Databases and Artificial Intelligence*, INRIN, Sophia-Antipolis.

Kobryn, C. 1999, 'UML 2001: A Standardization Odyssey', *CACM*, vol. 42, no. 10, pp. 29–37.

Kimball. R. 1996, *The Data Warehouse Toolkit*, John Wiley and Sons, New York.

Kulkarni, K. 2006, 'Overview of SQL:2003', URL: *www.wiscorp.com/SQL2003Features.pdf*.

Kurzweil, R. 1999, *The Age of Spiritual Machines*, Phoenix, London.

Kurzweil, R. 2005, *The Singularity is Near: When Humans Transcend Biology*, Penguin Group, New York.

Lacy, L. 2005, *OWL: Representing Information Using the Web Ontology Language*, Trafford Publishing, Victoria, BC.

Lassen K. and van der Aalst W. 2006, 'WorkflowNet2BPEL4WS: A Tool for Translating Unstructured Workflow Processes to Readable BPEL', BETA Working Paper Series, WP 167, Eindhoven University of Technology, Eindhoven.

Levesque, H. 1984, 'A Fundamental Trade-off in Knowledge Representation and Reasoning', *Proc. CSCSI-84*, London, Ontario, pp. 141–52.

Ling, T., Tompa, F. and Kameda, T. 1981, 'An Improved Third Normal Form for Relational Databases', *ACM Trans. on Database Systems*, vol. 6, no. 2, pp. 329–46.

Loney, K. and Koch, G. 2000, *Oracle8i: The Complete Reference*, Osborne/McGraw-Hill, Berkeley, CA.

Mark, L. 1988, 'The Binary Relationship Model', Tech. Report UMIACS–TR–88–67, University of Maryland.

Martin, J. 1993, *Principles of Object Oriented Analysis and Design*, Prentice Hall, Englewood Cliffs, NJ.

Martin, J. and Odell, J. 1998, *Object-Oriented Methods: A Foundation, UML edn*, Prentice Hall, Upper Saddle River, NJ.

Matthews, R. and McGee, W. 1990, 'Data Modeling for Software Development', *IBM Systems Journal*, vol. 29, no. 2, pp. 228–35.

McLeod, G. 2000, 'Beyond Use Cases', *Proc. EMMSAD'00: 5th IFIP WG8.1 Int. Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*, Kista, Sweden.

Meersman, R. 1982a, 'The RIDL Conceptual Language', Research Report, Int. Centre for Information Analysis Services, Control Data Belgium, Brussels.

Meersman, R. 1982b, 'The High-level End User', *Infotech State of the Art Report*, series 10, no. 7, Pergamon Press, pp. 535–53.

Meersman, R., De Troyer, O. and Ponsaert, F. 1984, *The RIDL User Guide*, Control Data Corp., Brussels.

Meersman, R. and Van Assche, F. 1983, 'Modeling and Manipulating Production Databases in terms of Semantic Nets', *Proc. 8th IJCAI Conference*, Morgan Kaufmann.

Melton, J. and Simon, A. 2002, *SQL:1999 Understanding Relational Language Components*, Morgan Kaufmann.

Melton, J. and Buxton, S. 2006, *Querying XML: XQuery, XPath, and SQL/XML in Context*, Morgan Kaufmann.

Menzel, C. and Mayer, R. 1998, 'The IDEF Family of Languages', *Handbook on Architectures of Information Systems*, eds. P. Bernus, K. Mertins and G. Schmidt, Springer-Verlag, Berlin, Germany, pp. 209–41.

Morgan, T. 2006, 'Some Features of State Machines in ORM', *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, eds. R. Meersman, Z. Tari, P. Herrero et al., Montpellier, Springer LNCS 4278.

Morgan, T. 2007, 'Business Process Modeling and ORM', *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, eds. R. Meersman, Z. Tari, P. Herrero et al., Vilamoura, Springer LNCS 4805.

Mok, W. and Embley, D. 1996, 'Transforming Conceptual Model to Object-Oriented Database Designs: Practicalities, Properties and Peculiarities', *Proc. ER'96: 15th Int. Conf. on conceptual modeling*, Springer LNCS, vol. 1157, pp. 309–24.

Moody, D. and Kortink, M. 2000, 'From Enterprise Models to Dimensional Models: A Methodology for Data Warehouse and Data Mart Design', *Proc. DMDW'00: Int. Workshop on Design and Management of Data Warehouses*, Report 25, Swiss Life IT RandD, Zurich.

Muller, R.. 1999, *Database Design for Smarties*, Morgan Kaufmann, San Francisco.

Mylopoulos, J. 1998, 'Information Modelling in the Time of the Revolution', *Information Systems*, vol. 23, no. 3/4, Pergamon, Exeter, pp. 127–55.

Nijssen, G. 1976, 'A Gross Architecture for the Next Generation Database Management Systems', *Proc. 1976 IFIP Working Conf. on Modelling in Data Base Management Systems*, ed. G. Nijssen, Freudenstadt, Germany, North-Holland Publishing, pp. 1–24.

Nijssen, G. 1977, 'Current Issues in Conceptual Schema Concepts', *Proc. 1977 IFIP Working Conf. on Modelling in Data Base Management Systems*, ed. G. Nijssen, Nice, France, North-Holland Publishing, pp. 31–66.

Nijssen, G. 1994, 'A General Analysis Procedure: Recent Advances in Universal Informatics', *Proc. 2nd NIAM-ISDM Conf.*, Albuquerque, NM.

NIST 1993, *Integration definition for information modeling (IDEF1X)*, FIPS Publication 184, National Institute of Standards and Technology. *www.sdct.itl.nist.gov/~ftp/idef1x.trf*.

Odell, J. 1998, *Advanced Object-Oriented Analysis and Design using UML*, Cambridge University Press, and SIGS Books, New York.

Olle, T., Hagelstein, J., Macdonald, I., Rolland, C., Sol, H., van Assche, F. and Verrijn-Stuart, A. 1991, *Information Systems Methodologies: a framework for understanding*, 2nd edn, Addison-Wesley,Wokingham, England.

OMG 2007, *Semantics of Business Vocabulary and Business rules (SBVR) Specification*. Available online at *http://omg.org/technology/documents/bms_spec_catalog.htm#SBVR*.

Orlowska, M. and Zhang, Y. 1992, 'Understanding the fifth normal form (5NF)', *Australian Computer Science Communications*, vol. 14, pp. 631–9.

Ouyang, C., van der Aalst, W., Dumas, M. and ter Hofstede, A. 2006 'From BPMN Process Models to BPEL Web Services'. In *Pro 4th International Conference on Web Services (ICWS)*, Chicago, IEEE Computer Society.

Ozsoyoglu, G. and Wang, H. 1993, 'Example-Based Graphical Database Query Languages', *Computer*, vol. 26, no. 5, pp. 25–38.

Pardi, W. 1999, *XML in Action*, Microsoft Press, Redmond WA.

Petrounias, I. and Loucopoulos, P. 1994, 'Time Dimension in a Fact-based Model', *Proc. ORM-1 Conference*, Dept of Computer Science, University of Queensland.

Pohl, K. 1994, 'The Three Dimensions of Requirements Engineering: A Framework and its Applications', *Information Systems*, Pergamon, Exeter, pp. 243–58.

Proper, H. 1994, 'A Theory for Conceptual Modelling of Evolving Application Domains', PhD thesis, University of Nijmegen.

Rissanen, J. 1977, 'Independent Components of Relations', *ACM Trans. on Database Systems*, vol. 2, no. 2, pp 317–25.

Ritson, P. and Halpin, T. 1993a, 'Mapping One-to-One Predicates to a Relational Schema', *Advances in Database Research: Proc. 4th Australian Database Conf.*, eds M. Orlowska and M. Papazoglou, World Scientific, Singapore, pp. 68–84.

Ritson, P. and Halpin, T. 1993b, 'Mapping Integrity Constraints to a Relational Schema', *Proc. 4th ACIS*, Brisbane (Sep.), pp. 381–400.

Rochfeld, A., Morejon, J. and Negros, P. 1991, 'Inter-Relationship Links in E-R Model', *Proc. 9th ER Conf.*, ed. H. Kangassalo, Elsevier Science, Amsterdam.

Ross, R. 1998, *Business Rule Concepts*, Business Rule Solutions, Inc., Houston TX.

Rumbaugh, J., Jacobson, I. and Booch, G. 1999, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading MA.

Rummler, G. and Brache, A. 1995, *Improving Performance: How to Manage the White Space in the Organization Chart*, 2nd edn, Jossey-Bass Publishing.

Seidman, C. 2000, 'Creating Horizontally Partitioned Views', *SQL Server Magazine*, no. 14, Duke Communications, pp. 51–3.

Senko, M. 1975, 'Information Systems: Records, Relations, Sets, Entities and Things', *Information Systems*, vol. 1, no. 1, Jan. 1995, Pergamon Press, pp. 3–13.

Shneiderman, B. 1992, *Designing the User Interface*, 2nd edn, Addison-Wesley, Reading MA.

Shoval, P. Gudes, E. and Goldstein, M. 1988, 'GISD: A Graphical Interactive System for Conceptual Database Ddesign', *Information Systems*, vol. 13, no. 1, pp. 81–95.

Shoval, P. and Kabeli, J. 2000, 'Functional and Object-Oriented Analysis and Design of Information Systems—An Integrated Methodology', *Proc. EMMSAD'00: 5th IFIP WG8.1 Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*, Kista, Sweden.

Shoval, P. and Shreiber, N. 1993, 'Database Reverse Engineering: from the Relational to the Binary Relational Model', *Data and Knowledge Engineering,* vol. 10, pp. 293–315.

Siau, K. and Halpin, T. (eds) 2000, *Unified Modeling Language: Systems Analysis, Design and Development Issues*, Idea Group, Hershey, PA.

Silverston, L. 2001a, *The Data Model Resource Book: Vol. 1*, John Wiley, New York.

Silverston, L. 2001b, *The Data Model Resource Book: Vol. 2*, John Wiley, New York.

Simsion, G. and Witt, G. 2005, *Data Modeling Essentials*, 3rd edn., Morgan Kaufmann.

Snodgrass, R. 1990, 'Temporal Databases: Status and Research Directions', *SIGMOD Record*, vol. 19, no. 4, pp. 83–89.

Snodgrass, R. 2000, *Developing Time-Oriented Database Applications in SQL*, Morgan Kaufmann, San Francisco.

Stonebraker, M., Rowe, L., Lindsay, B., Gray, J., Carey, M., Brodie, M., Bernstein, P. and Beech, D. 1990, 'Third Generation Database System Manifesto', *ACM SIGMOD Record*, vol. 19, no. 3.

Sundblad, S. and Sundblad, P. 2002, *Design Patterns for Scalable Microsoft .NET Applications*, Sundblad and Sundblad ADB-Arkitektur AB, Uppsala, Sweden. Electronic version available online at *www.2xsundblad.com/*.

ter Hofstede, A., Proper, H. and van der Weide, T. 1992, 'Data Modeling in Complex Application Domains', *Proc. CAiSE'92: Fourth Int. Conference on Advanced Information Systems Engineering*, ed. P. Loucopoulos, Springer LNCS, no. 593, pp. 364–77.

ter Hofstede, A. 1993, *Information Modeling in Data Intensive Domains*, PhD thesis, University of Nijmegen.

ter Hofstede, A., Proper, H. and van der Weide T. 1993, 'Formal Definition of a Conceptual Language for the Description and Manipulation of Information Models', *Information Systems*, vol. 18, no. 7, pp. 489–523.

ter Hofstede, A. and van der Weide, T. 1994, 'Fact Orientation in Complex Object Role Modeling Techniques', *Proc. 1st Int. Conf. on Object-Role Modeling (ORM-1)*, eds T. Halpin and R. Meersman, Magnetic Island, Australia, pp. 45–59.

ter Hofstede, A., Proper, H. and van der Weide, T. 1996, 'Query Formulation as an Information Retrieval Problem', *The Computer Journal*, vol. 39, no. 4, pp. 255–74.

Thalheim, B. 1984, 'Deductive Basis of Relations', *Proc. MFSSS-84*, Berlin, Springer LNCS, no. 215, Heildelberg, pp. 226–30.

Thalheim, B. 1994, *Fundamentals of Entity-Relationship Modelling*, Springer, Heidelberg.

Thomsen, E., Spofford, G. and Chase, D. 1999, *Microsoft OLAP Solutions*, John Wiley, New York.

Trog, D., Vereecken, J., De Leenheer, P. and Meersman, R. 2006, 'T-Lex: A Role-based Ontology Engineering Tool', *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, eds. R. Meersman, Z. Tari, P. Herrero et al., Montpellier. Springer LNCS 4278.

van Bommel, P. and van der Weide, T. 1992, 'Reducing the Search Space for Conceptual Schema Transformations', *Data and Knowledge Engineering*, vol. 8, pp. 269–92.

van der Aalst, W. and van Hee, K. 2004, *Workflow Management: Models, Methods, and Systems,* MIT Press.

van Griethuysen, J. (ed.) 1982, *Concepts and Terminology for the Conceptual Schema and the Information Base*, ISO TC97/SC5/WG3, Eindhoven.

Verheijen, G. and van Bekkum, J. 1982, 'NIAM: An Information Analysis Method', *Information systems Design Methodologies: a comparative review, Proc. IFIP WG8.1 Working Conf.*, Noordwijkerhout, The Netherlands, North Holland Publishing, pp. 537–90.

Vermeir, D. 1983, 'Semantic Hierarchies and Abstractions in Conceptual Schemata', *Information systems*, vol. 8, no. 2, pp. 117–24.

Vieira R. 2007, *Professional SQL Server 2005 Programming*, Wiley Publishing, Indianapolis.

Vincent, M. 1994, 'Semantic Justification of Normal Forms in Relational Database Design', PhD thesis, Monash University.

Vincent, M. and Srinivasan, B. 1993, 'Redundancy and the Justification for Fourth Normal Form in Relational Databases', *Int. Journal of Foundations of Computer Science*, vol. 4, no. 4, pp. 355–65.

Vincent, M. and Srinivasan, B. 1994, 'A Note on Relation Schemes which are in 3NF but not in BCNF', *Information Processing Letters*, vol. 48, pp. 281–3.

Walmsley P. 2007, *XQuery*, O'Reilly Media, Inc.

Warmer, J. and Kleppe, A. 2003, *The Object Constraint Language Second Edition*, Addison-Wesley, Boston, MA.

Wintraecken, J. 1990, *The NIAM Information Analysis Method: Theory and Practice*, Kluwer, Deventer, The Netherlands.

Zachman, J. 1987, 'A Framework for Information Systems Architecture', *IBM Systems Journal*, vol. 26, no. 3, pp. 276–92.

Zaniolo, C. 1982, 'A New Normal Form for the Design of Relational Database Schemas', *ACM Trans. on Database Systems*, vol. 7, no. 3, pp. 489–99.

Index

Symbols and Numbers

* (asterisk)
  for attributes in Barker notation, 309
  for derived fact types, 98, 100
  for derived subtypes, 250
  in Euler diagrams, 215
  as multiplication symbol, 100
" (double quotes)
  for delimited identifiers in SQL, 559
  for objectified association names, 88
… (ellipsis), for value type definitions, 217
! (exclamation mark), for independent object types, 219, 220–221
- (hyphen)
  in predicates, 115
[] (italic square brackets), for projection, 150
+ (plus superscript)
  for semiderived fact types, 204
  for semiderived subtypes, 250
– (minus sign), for nulls, 227
% (percent sign), as **like** operator wildcard, 585
? (question mark), for nulls, 162
_ (underscore), as **like** operator wildcard, 585
1:1 associations, 11, 115
  relational mapping for, 487
1:1 into mapping (injection), 174, 185
1:1 relation, 115
1:*n* relation, 114
m:*n* relation, 116
n:1 relation, 114
n–1 rule, 138
1NF (first normal form), 737
2NF (second normal form), 739
3GL (third generation language), 406
3NF (third normal form), 741–742
4GL (fourth generation language), 18
4NF (fourth normal form), 745
5GIS (fifth generation information system), 28
5GL (fifth generation language), 20
5NF (fifth normal form), 747
100% Principle, 60

A

Abox (assertion box), 867
absorption approach for relational mapping of subtype constraints, 498
abstract classes in UML, 374
abstract syntax, 60
abstraction mechanisms
  of a language, 60
  schema abstraction mechanisms, 835, 852
access rights
  of end users, 31, 52
  in SQL, 474, 664–666
  tables for user groups, 862
activity diagram, 803–806
acyclic relation, 286
aggregation in UML, 379–382, 396
Aho, A., 737, 772
AI (artificial intelligence) systems, 47
algebra, relational, 528–553
aliases
  as clause for, 580, 613
  for tables, 532
alethic constraints, 410, 893
all option
  before select-list, 563
  count bag function with, 606
  quantified comparison subqueries with, 600
  subqueries with, 595
  union operator with, 591
alter table statement, 644–645, 691
alter view statement, 647
alternate key constraints
  in IDEF1X, 325, 330
  in relational schemas, 476
antisymmetric relation, 282
antirigid property, 427
any option
  quantified comparison subqueries with, 600
  subqueries with, 595
arithmetic operators in SQL, 602–605
arity
  key length check, 137–149
  names for, 92
  of predicates, 10, 68, 88

projection-join check, 109, 137, 151–156, 747
array types, 646, 757
artificial intelligence (AI) systems, 47, 71, 872, 880, 892
as clause, 580, 613
asc option for specifying order, 567
as-is model, 78–79
asserted fact type, 99, 113, 118, 204
asserted subtype, 249–250, 893–895
assignment operation, 537
association in UML, 357–363
aggregation and, 379–382, 396
association classes, 362, 389, 392
attributes vs., 355–357
binary, 357
derived, 380
multiplicity constraints for, 358–360
n-ary, 358
qualified, 363–364
association redefinition in UML, 387
associative entity types in IDEF1X, 331
associative laws in relational algebra, 633
associative operations, 531
asterisk (*)
for attributes in Barker notation, 309
for derived fact types, 98, 100
for derived subtypes, 250
in Euler diagrams, 215
as multiplication symbol, 100
asymmetric relation, 281, 284
atomic fact, 484, 893
attribute multiplicity constraints in UML, 352
attribute style for derivation rules, 16, 36, 98, 106
attributes
Barker notation, 309
changeability in UML, 384, 396, 428
ER vs. ORM, 10
hiding, 856
in IDEF1X, 325
key, 343, 739, 742
mandatory, 309
minor fact types viewed as, 856
multivalued, in UML, 354–357
nonkey, 343, 739
in relational schemas, 475
single-valued, in UML, 352
UML vs ORM, 353–357
avg bag function, 605–607
group by clause with, 615

B

bag functions in SQL, 602–611
avg, 605–607
case, 609
cast, 603
count, 606
distinct option with, 606
group by clause with, 613–618
lower, 608
max, 601, 606
min, 606
substring, 608
sum, 605–606
upper, 608
bags, 213, 226, 432, 437, 470, 536,877
Barker ER notation, 308–318
attributes in, 309
changeability constraints in, 317
entity types in, 309
exclusion constraints in, 314–315
frequency constraints in, 316
identification bars in, 312–313
illegal ring associations in, 312
mandatory constraints in, 309–311
naming discipline for relationships, 310
nontransferable relationships in, 317
optional roles in, 309–311
ORM vs., 314
Other subtype in, 316–317
relationships in, 309
subtyping in, 316–317
Barker, Richard, 308
base fact type, *See* asserted fact type
base table, 46, 684
BCNF (Boyce-Codd normal form), 744
Beeri, C., 737
behavior-oriented perspective for conceptual design, 51–52
Bentley, J., 106
…**between**…**and**… operator, 584–585
biconditional derivation rule, 99
binary associations in UML, 357
binary predicates
in conceptual schema diagrams, 83
defined, 67
uniqueness constraints on, 113–121
binary-unary contraction, 719
Boyce, Raymond, 556, 635, 736, 744
Boyce–Codd normal form (BCNF), 744, 770
BPEL (Business Process Execution Language), 813–814, 818, 833
BPDM (Business Process Definition Metamodel), 813–817, 832–833
BPMN (Business Process Modeling Notation), 783, 813–819, 823–826, 831–833, 857
business domain. *See* universe of discourse (UoD)

C

Campbell, L., 85, 862, 891
candidate identifiers, 180, 189, 208
candidate keys in relational schemas, 5–6, 476, 525, 537, 756
Cartesian product (cross joins)
as associative, 531
overview, 527–531
in SQL, 571, 575, 618, 631
cascade option, in SQL, 644-646
case
setting for strings, 565
SQL bag functions and, 609
in SQL queries, 563
case bag function, 609
CASE (computer-assisted software engineering) tools, 3, 24, 39, 84, 498, 524, 891
cast function
bag function, 603, 608, 615, 674
for domains, 646
categories (subtypes) in IDEF1X, 332–333
categorization relationships in IDEF1X, 332
categorization type, 456, 462–463, 472
cells in relational databases, 4
centralized database systems, 879
Chamberlin, Don, 556, 635
changeability constraints in Barker ER, 317
changeability in UML, 384–385
char data type, 560–562, 585
character strings
case setting for, 565
concatenation operator, 604–605
ordering, 567

check clause
for constraints, 47, 290, 407,494
restrictions, 639
subqueries in, 595
check constraint, 433, 641
See also check clause
Chen, Peter, 8, 306–307, 342–343
Chen's original notation, 306–308
CIP (conceptual information processor), 30–39, 56, 167, 176
clarity
of a language, 60
semantic, 722–733, 847, 850
classes in UML, 348–351
Closed World Assumption (CWA), 65, 218, 450–451, 464, 471, 511
CODASYL (Conference on Data Systems and Languages) Database Task Group, 21
Codd, E. , 17, 21, 25, 46, 57, 81, 528, 532, 555, 736–737, 741, 744,772
collection types, 432–438
denormalization and, 759
column-list joins in SQL, 572, 574
columns
choosing in SQL, 562–564
mandatory, 46
meaningful names for, 485
order of, 46
ordering in SQL, 568
in relational databases, 45
relational mapping and derived columns, 497
combining entity types, 95–102
See also generalization of object types
comments
in ORM, 70, 341
in SQL, 581
commit statement, 566
commutative laws in relational algebra, 530, 633
commutative operations, 530
compact disc retailer case study
CSDP for, 193–199
query strategies, 549–550
relational mapping for, 500–502
comparison operators
join operation with, 538–539
order of evaluation for, 566
overview, 528
selection operation with, 533
comparison subqueries in SQL, 595, 600, 623
completeness
final check in ORM, 298
of requirements specifications, 50
complete subtyping constraint in UML, 373
complex objects, 871–873, 876–877, 890
complex types, 432–438
denormalization and, 759
composite key, 477, 723–724, 739, 743
compositely identified object type, 417, 499–501, 524, 894
compound fact types, 129, 137, 139, 151, 893
compound occurrence frequency, 273
compound reference scheme, 177–178, 180, 190, 195, 208
candidate identifiers, 180, 189, 208
minimizing number of components, 181
rigid identifier, 181
semantic names, 182
compound transaction, 37, 293
computational completeness, 873, 877
computer-assisted software engineering (CASE) tools. 3, 24, 39, 84, 498, 524, 891
concatenated string operator in SQL, 604–605
conceptual database
access rights, 31
overview, 29
conceptual design, 51
conceptual information processor (CIP), 30–39, 56, 167, 176
conceptualization stage of reengineering, 760
conceptual join in ORM, 129, 151, 203, 307, 378, 400, 402, 404, 469
conceptual level, 32–39
and other levels, 26–32
conceptual modeling language criteria, 60–62
conceptual query languages, 843–851
ConQuer, 19, 107, 356, 846–852
English Query, 843–845
RIDL (Reference and Idea Language), 106, 845–846
conceptual schema
access rights, 31
completeness check, 298
components, 33
conceptual database and, 29
conceptual query languages for, 843–851
constraints section, 33
derivation rules section, 33
equivalence, 688, 769
external consistency check, 295
fact types section, 33
implementation overview, 43–48, 474–475
in information systems life cycle, 50–51
internal consistency check, 295
for large applications, 62–63
main sections, 33
mapping to logical schema, 45, 52
mapping to relational schema, 43–48, *See also* relational mapping
metaschema for, 881–885
optimization, 724–735
overview, 32–39
as recorded-world schema, 120
redundancy check, 297–298
reengineering, 759–765
reference schemes section, 35
See also conceptual schema optimization; conceptual schema transformations; relational mapping
conceptual schema design procedure (CSDP)
compact disc retailer case study, 193–199
overview, 62–63
step 1 (from examples to elementary facts), 63–80
step 2 (draw fact types, and populate), 81–94
step 3 (trim schema, note basic derivations), 95–103
step 4 (uniqueness constraints), 110–158
step 5 (mandatory roles), 159–210
step 6 (value, set-comparison, and subtype constraints), 211–270
step 7 (other constraints and final checks), 271–304
conceptual schema diagrams
derived fact types in, 98–100
draw fact types, 81–92
flattened (unnested) versions, 87–88
instance diagrams vs., 82
nesting, 88
populating, 84–88

symbol glossary, 896–902
verbalization of, 89–91
conceptual schema equivalence, 688, 769
See also equivalence of conceptual schemas
conceptual schema optimization, 724–735
data migration, 766–767
defined, 724
denormalization and low-level optimization, 753–758
main factors, 722
reducing number of mapped tables, 723–726
simplifying individual tables, 727–732
step 1: unifying compatible, composite keys, 723–724, 727
step 2: object types with both functional and nonfunctional role, 723–725
step 3: table width guideline (TWG), 727–729
steps 4 and 5: exclusive roles, 730–732
target system, 733
conceptual schema transformations, 688–721
binary-unary contraction, 719–720
defined, 689
denormalization and low-level optimization, 753–758
entity type elimination/introduction, 720
nesting, coreferencing, and flattening, 702–714
for optimization, 524–735
predicate specialization and generalization, 692–700
reengineering the conceptual schema, 761–767
unary-binary expansion, 719–7201
value concatenation/separation, 719
See also specialization and generalization of predicates
conceptual-schema-based queries, 845
concrete classes in UML, 374
concrete syntax, 47, 60
concurrency, 662–664
deadlock, 664
dirty read, 662, 664
isolation level, 664
lock, 662
nonrepeatable read, 662, 664
phantom, 662, 664
conditional joins in SQL, 572–573
Conference on Data Systems and Languages (CODASYL) Database Task Group, 21
connection relationships in IDEF1X, 324–326, 334, 342
ConQuer, 19, 846–852
consistency
CIP enforcement of, 38–39
external, 295–296
final checks in ORM, 295–296
of interface, 863
internal, 295
of requirements specifications, 50
subtype definitions and, 245
constraint implication, 234
constraint option, 639
constraint
acyclic, 285
alethic, 108, 408–410, 428, 469–470, 747, 893–894
antisymmetric, 282
asymmetric, 281
check clause for, 47, 290
conceptual example, 35
counterexamples in ORM, 11, 120, 234
deontic, 108, 284, 408–410, 427, 469–470
dynamic, 33, 56, 110, 399, 424–428, 470, 894
equality, 226–227, 232
exclusion, 228–230, 233–234
exclusive-or, 229, 248, 314, 370–371
external frequency, 275
external uniqueness, 128–136
frequency, 272–275
functional dependency (FD), 140–146, 156, 203, 234, 735, 739
inclusive-or, 165–169
in relational schemas, 46, 475–482
internal uniqueness, 111–127
intransitive, 284–285
irreflexive, 280
join, 377–378, 399–406, 469
mandatory role, 162–171
multiplicity in UML, 352, 358
object cardinality, 289
ORM vs. ER, 338
ORM vs. UML, 396
purely reflexive, 287, 901
restricted mandatory role, 291, 304
restricted uniqueness, 291, 853
ring, 277–287
role cardinality, 289, 303
set-comparison, 224–234
SQL constraint option, 639
static vs. dynamic, 33, 56, 111, 428, 894
subset, 225–226, 229–234, 366–368
subtyping, 238–257, 372–376
symmetric, 279
textual, 18, 338, 370, 376–377
transition, 426, 431
transitive, 279
uniqueness, 111–136
value, 216–220
value-comparison, 290
context diagrams, 50, 860–861
context-dependent reference, 265–266, 322, 376, 518
contextual equivalence of conceptual schemas, 689
controlled redundancy, 112, 297, 484, 740, 755, 771
conversion rules, 190–191, 208
coreference/flatten (CR/F) equivalence theorems, 707
coreferenced object type, 335–336, 503, 720, 769
See also compound reference scheme
correlated subqueries in SQL, 595, 619–625, 836
corresponding option, **union** operator with, 592
counterexample, 11, 15–16, 120, 133, 201
cost/benefit analysis, 50
count bag function, 606
group by clause with, 611–618
CR/F (coreference/flatten) equivalence theorems, 707
create assertion statement, 639–640, 755–756
create database statement, 638
create domain statement, 384, 481, 639, 645
create schema statement, 638, 640
create table statement, 481, 638–642, 684
create trigger statement, 652–654
create type statement, 639, 646

create view statement, 646–651
cross joins. *See* Cartesian product (cross joins)
CSDP. *See* conceptual schema design procedure (CSDP)
cursor, in SQL, 660–664
CWA. *See* Closed World Assumption.

D

DAG (directed acyclic graph), 316, 321, 373
data
  information vs., 1–2
  usage in this book, 32
data control language (DCL), 558
data definition language (DDL), 47, 558, 638
  alter table statement, 644–645, 691
  alter view statement, 647
  check clause, 47, 290, 407, 433, 494, 595, 639, 641
  check constraint, 433, 641
  columns, 638–639
  constraint option, 639
  create assertion statement, 639–640, 755–756
  create database statement, 638
  create domain statement, 384, 481, 639, 645
  create schema statement, 638, 640
  create table statement, 481, 638–642, 684
  create type statement, 639, 646
  create view statement, 646–651
  deciding where to declare constraints, 641
  default clause, 641
  deontic rule, 108, 284, 408–410, 427, 469–470
  distinct types, 646
  drop schema statement, 638
  drop table statement, 631, 645
  drop trigger statement, 653
  drop view statement, 558, 646
  foreign key, 6, 18, 324, 639
  indexes, 659–660
  match operator, 643–644
  references clause, 643–644
  referential actions, 643
  referential integrity, 526, 643–644
  structured types, 646
data flow diagrams, 860–861, 889
data maintenance, 48, 482
data manipulation language (DML), 558, 638, 873
data marts, 837–838, 841–842, 888
data migration
  and query transformation, 766–767
  defined, 691
  reengineering stage, 760
data mining, 841, 888
data presentation, 48, 482
data procedural language (DPL), 558
data retrieval, 48, 482
data types
  in SQL, 560–562
  in UML, 351
  syntactic, 219
data use cases
  for CSDP, 104, 858–859, 889
  defined, 7–9
data values in UML, 354
data warehouses, 836–843, 888, 891
database design, importance of, 2–6
database management system (DBMS)
  advantages of, 3
  defined, 2
  history of, 19–23
  relational, 554–556
  postrelational, 871–880
database schemas, 638
databases (defined), 2
data-oriented perspective for conceptual design, 51
DBMS. *See* database management system (DBMS)
DCL (data control language), 558
DDL. *See* data definition language (DDL)
deadlock, 664
decimal data type, 560–561
declarative language, 20
decomposition approach to normalization, 735–736
deconceptualization, 734, 751, 770
deductive databases, 878
default clause, 384, 641
default values, 189, 382–384, 396, 522
definiendum, 99
definite description, 9, 66, 104, 177
definitions for derivation rules, 99
delayed projection laws in relational algebra, 633
delete operation, 48
delete statement, 631
delete-action for references clause, 643
delimited identifiers in SQL, 599
De Morgan's laws, 536
denormalization and low-level optimization, 753–758
  complex types, 757
  for data mart and data warehouse performance, 840
  fragmentation, 758
  indexes, 753, 758
  introducing controlled redundancy, 643–645
  materialized views, 754
  redundancy between tables, 755
  redundancy within a table, 756
  working tables, 754
derivation rules
  asterisk notation for, 98, 100
  attribute style for, 16, 36, 98, 106
  biconditional, 99
  definitions for, 99
  derived fact types, 33
  Horn clauses, 35
  redundancy in, 297
  relational style for, 99
  in UML, 380, 386
derived associations in UML, 380, 387–388
derived columns, relational mapping and, 497
derived fact types
  asserted fact types vs., 99
  in conceptual schema diagrams, 98–102
  eager evaluation (derived-on-update), 100, 236, 297, 521
  lazy evaluation (derived-on-query), 100, 236, 521
  logical derivation check, 200–206
  semiderived, 99, 102, 204
  relational mapping and, 497
  See also logical derivation check
derived redundancy, 297
derived tables. *See* views
derived subtypes, 245–250, 269, 333
derived units for conversion rules, 190
derived-on-query (lazy evaluation) of derived fact types, 100, 236, 521
derived-on-update (eager evaluation) of derived fact types, 100, 236, 297, 521
desc option for specifying order, 567–569, 616, 618

description logic (DL), 867–870, 890
development frameworks, 27, 48–57
difference operation
  in SQL, 593–594, 598
  set difference, 215
dimension table, 840–841, 888
directed acyclic graph (DAG), 316, 321, 373
dirty read, 662, 664
discriminator
  in IDEF1X, 332–334
  in UML, 375, 396
disjoint sets, 215, 371
disjoint subtyping constraint in UML, 373
disjunctive mandatory roles, 168–169
  exclusion constraints with, 229
  subset constraints and, 229
  See also inclusive-or constraint
disjunctive reference, 187–189, 208–209, 402, 510, 522, 526
distfix (a.k.a. mixfix) form, 68
distinct option
  bag functions with, 606
  before select-list, 563–564
distinct types, 646
distributed database system, 880
distributive laws in relational algebra, 633
division operation, 545, 603
DKNF (domain key normal form), 749–750, 771–772
DML (data manipulation language), 558, 638, 873
documentation
  in information systems life cycle, 53
  requirements specifications, 50
domain constraints. *See* value constraints in ORM
domain dependencies, 749
domain experts
  defined, 6
  mandatory roles and, 163
  translating examples into elementary facts and, 74
domain key normal form (DKNF), 749–750, 771–772
domains
  in DDL, 639, 645
  in relational schemas, 475
double quotes (“),
  for delimited identifiers in SQL, 559, 634, 672, 674
  for objectified associations, 88
DPL (data procedural language), 558
drop schema statement, 638
drop table statement, 631, 645
drop trigger statement, 653
drop view statement, 558, 646
duplicated object type, 171, 896
dynamic behavior, 773–775, 780, 790, 799, 811
dynamic constraint, 33, 56, 110, 399, 424–428, 470, 894

E

eager evaluation of derived fact types, 100, 236, 297, 521
EBNF (Extended Backus Naur Form) notation, 564
efficiency
  of interface, 691
  in relational schemas, 412
EKNF (elementary key normal form), 564, 569
elementary facts
  defined, 64–65
  irreducibility of, 70
  in relational database tables, 45
elementary key normal form (EKNF), 743–744, 770
elementary update, 36
elements of sets, 212
ellipsis (…), for value type definitions, 217
encapsulation, 873–877, 890
end users
  access rights, 31, 52
  defined, 30
enterprise architecture, 54, 837
Enterprise Resource Planning (ERP), 51
entities
  defined, 35, 66
  definite descriptions for, 66–67, 177, 350
  entity type, 66
  in IDEF1X, 324
  values vs., 95
entity integrity rule, 479, 522, 525
entity type elimination/introduction, 720
entity types, 66
  Barker notation, 309
  classes in UML, 351
  combining, 96–98
  in conceptual schema diagrams, 82
  defined, 66
  elimination/introduction of, 720
  in FCO-IM, 221
  in IDEF1X, 324
  in IE notation, 318
  population of, 163
  primitive, 95, 164
  reference scheme choices for, 185–191
  weak, 307
  See also object types
Entity-Relationship modeling (ER), 306–344
  Barker notation, 308–317
  Chen’s original notation, 306–308
  glossary, 903–906
  historical overview, 306–308
  IDEF1X, 322–333
  Information Engineering (IE) notation, 318–321
  mapping from ORM, 334–341
  overview, 7–8, 314–315, 346–347
  weak entity types, 307
  See also Barker notation; IDEF1X; Information Engineering (IE) notation
equality constraints in ORM, 226–227, 232
equality constraints in UML, 369
equality of sets, 212
equijoin operation, 540–541, 575
equivalence of conceptual schemas, 688, 769
  contextual, 689
  coreference/flatten (CR/F) equivalence theorems, 707
  nest/coreference (N/CR) equivalence theorem, 705, 769
  nest/flatten into many fact types (N/Fm) equivalence, 709
  nest/flatten (N/F) equivalence theorem, 706
  overview, 688–691
  in UML, 690
ER. See Entity-Relationship modeling (ER)
ER glossary, 903–906
ERmap procedure, 335
ERP (Enterprise Resource Planning), 51
escape character, **like** operator with, 588

Euler diagram
overview, 214–216
for ring constraints, 286
for subtypes, 240–241, 316, 321, 342
Euler, Leonhard, 214
Everest, Gordon, 309, 319
example sets, 76
except operation. *See* difference operation
except operator in SQL, 593–594
exclamation mark (!), for independent object types, 219, 220–221
exclusion constraints in ER
Barker notation, 314–315
IE notation, 321
exclusion constraints in ORM, 228–230, 233–234
pair-exclusion constraints, 268
predicate specialization / generalization and, 694–695
subtypes and, 248–249, 251
exclusion constraints in UML, 371
exclusive-or constraints
in Barker ER, 314
in IE, 322
in ORM, 229, 248
in UML, 370–371
executable UML, 811–812
exhaustion constraints in ORM, 249
existential fact, 83, 112, 117, 161, 219, 484, 894
existential subqueries in SQL, 595, 623–625
exists quantifier, 595, 599, 623, 634
expert systems, 28
exportation laws in relational algebra, 633
expressibility of a language
defined, 60
expressibility-tractability trade-off, 61
Extended Backus Naur Form (EBNF) notation, 564
eXtensible Markup Language (XML), 23, 666–683
extensional definition of sets, 213
extensional uniqueness constraint, 433–434, 471
external consistency, 295–296
external frequency constraint, 275
external interface, 754, 759, 843, 862–864,
external object type, 171, 208, 896
external schema, 31, 474
external uniqueness constraints in ORM, 128–136
qualified associations in UML vs., 363–364
relational mapping for, 492–494

F

fact, 441
immutable, 397, 426
fact roles, 161, 207. *See also* roles
fact tables, 11, 45, 84
fact types
asserted, 99, 113, 118, 204
binary, 11, 114
changeable, 418–419
compound, 129, 137, 893
defined, 9
definitional, 413–414
derived, 99. *See also* derived fact types
heterogeneous, 86
historical, 111, 421, 838
homogeneous, 86
major, 855
minor fact types viewed as attributes, 353, 856, 889
once-only, 415, 470
primitive, 99
repeatable, 414–415, 470
ring, 86–87, 277–287, 292, 371
semiderived, 99, 204–205, 209, 895
snapshot, 111, 419
static constraints on, 33, 56
ternary, 87, 123
unary, 86, 111
fact-oriented modeling, 7, 9. *See also* Object-Role Modeling (ORM)
Fagin, R., 737, 745–747, 749, 772
FCO-IM (Fully Communication Oriented Information Modeling), 221, 270, 445
FDs. *See* functional dependencies (FDs)
feasibility study, 50, 56
federated database, 853, 880
fields. *See* columns
fifth normal form (5NF), 747
fifth generation information system (5GIS), 28
fifth generation language (5GL), 20
final checks in ORM, 271–304
final option of **create type** statement, 646
Finkelstein, Clive, 318–319, 322, 343
first normal form (1NF), 737
first-order logic, 456–458, 461–463
flattened (unnested) ORM diagrams, 88
flattening, conceptual schema transformations and, 702–714
foreign keys, 6
in DDL, 641
in IDEF1X, 324–325
indexes for, 659
Formal ORM Language (FORML), 35, 852
forms, input. *See* input forms
forward engineering the conceptual schema, 760
fourth normal form (4NF), 745
fourth generation language (4GL), 18
fragmentation, 758
Framework of Information System Concepts (FRISCO), 55–57
frequency constraints, 272–275
FRISCO (Framework of Information System Concepts), 55–57
from clause
basics, 562–565
for joins, 570–581
subqueries in, 595
full outer joins
in ORM, 130
in SQL, 572
function trees, 889
functional dependencies (FDs)
2NF and, 739
3NF and, 632
BCNF and, 744
defined, 140, 735
EKNF and, 743
embedded, 143
uniqueness constraints and, 143–144
functional role, 200
functions in SQL, 582, 606

G

Garbage In Garbage Out (GIGO), 24, 39
garbage, factual vs logical, 39
generalization of object types in ORM, 260–266

context-dependent reference scheme and, 265–266
generalization procedure (GP), 266
generalization of object types in UML, 372
generalization of predicates. *See* specialization and generalization of predicates
generated columns, 646
generic entities (supertypes) in IDEF1X, 332–333
GIGO (Garbage In Garbage Out), 24, 39
glossaries
ER, 903–9063
ORM, 893–902
UML, 907–909
grant statement, 665
graphical user interface (GUI), 863
Grelling's paradox, 461
group by clause, 611–618
guarded transition diagrams, 426
GUI (graphical user interface), 863

H

Halpin, T., 25, 60, 107
Harris, J., 397
Harris, R., 78
having clause, 615
Hay, David, 316, 350
HCI (human-computer interface) design, 863
HEDs (Hypothetical Euler Diagrams), 215
heterogeneous fact type, 86
hierarchic data model, 20
higher-order logic, 456–462, 468
higher-order types, 456–468
historical fact types, 111, 421, 838
HOLAP (hybrid OLAP), 841
homogenous fact type, 86
horizontal normal form, 737
Horn clauses, 35
human-computer interface (HCI) design, 863
hybrid OLAP (HOLAP), 841
hyphen binding, 115
Hypothetical Euler Diagrams (HEDs), 215

I

IDEF1X, 322–334
alternate keys in, 325
attributes in, 325
categories (subtypes) in, 332–333
connection relationships in, 325
entities in, 325
foreign keys in, 325
generic entities (supertypes) in, 332–333
identifier-dependent entity types in, 324–325
identifier-independent entity types in, 324–325
identifying relationships in, 326
intersection entity types in, 331
mapping from ORM, 334–335
nonidentifying relationships in, 328, 331
primary keys in, 325
relationships in, 325–332
ring associations in, 333
role names in, 333
views, 324
identification bars in Barker notation, 312–313
identifier-dependent entity types in IDEF1X, 324–325
identifier-independent entity types in IDEF1X, 324–325
identifiers in SQL, 558–560
delimited, 559
regular, 558–559
reserved words and, 559
identifying relationships in IDEF1X, 326
identity column, 641–642, 658
identity of sets, 212
IE notation. See Information Engineering (IE) notation
immutable facts, 397, 426
implementation
of conceptual schemas, 43–48, 474–475
in information systems life cycle, 53
implied constraints
equality constraints, 227
exclusion constraints, 234
mandatory roles, 164
subset constraints, 226, 234
subtype exclusion or exhaustion constraints, 247
in operator in SQL, 582–584
incomplete subtyping constraint in UML, 373
inclusive-or constraint, 165–169
independent object types
defined, 219
FCO-IM and, 221
nested object types as, 220
notation, 219–220
relational mapping for, 495
value constraints and, 216–221
indexed views, 754, 758
indexes, 659–660, 753–754
indirect subtypes, 242–243, 269
information, data vs., 2
Information Engineering (IE) notation, 318–322
entity types in, 318
exclusive-or constraints in, 322
cardinality constraints in, 319–320
intersection entity types in, 321
mandatory constraints in, 319
relationships in, 319
subtyping in, 322
information levels, 28–32
conceptual, 29–30
external, 31
logical, 31
physical, 31
information modeling
approaches compared, 6–19
overview, 2–6
See also Entity-Relationship modeling (ER); Object-Role Modeling (ORM); Unified Modeling Language (UML)
information system
in the broad sense, 49
life cycle, 49–53
in the narrow sense, 49
information systems engineering, 3
information systems life cycle, 49–53
inheritance of subtypes
in ORM, 243
in UML, 372
initial values in UML, 382
injection, 174, 185
Inmon, Bill, 837
input forms
as fact examples source, 76–79
for defining subtypes, 255
insert operation, 48
insert statement
for adding rows to table, 566
for adding rows to views, 649–651
for updating table populations, 630
instance diagrams, 82

instances
in ORM, 66, 350
in UML, 354
integer data type, 561
integrity, 33
entity, 479, 522, 525
referential, 46, 479, 526
intelligence, 28
intensional definition of sets, 213
interface. *See* external interface
internal consistency, 295
internal uniqueness constraints in ORM, 111–127
interpredicate uniqueness constraints in ORM. *See* external uniqueness constraints in ORM
intersect operator, 215, 529, 593
intersection entity types
in IDEF1X, 331–332
in IE notation, 321
intersection operation
in relational algebra, 529–531
in SQL, 593–594
intransitive relationship, 284–285
intrapredicate uniqueness constraints in ORM, 111–127
inverse predicates, 10, 69, 82, 87
irreducibility of elementary facts, 70
irreflexive relationships, 280–281
is null operator, 588–590
isolation level, 664

J

JD (join dependency), 747
join condition, 129, 540
join constraints
in ORM, 377–378, 399–406, 469
in UML, 404
join dependency (JD), 747
join operations in ORM
conceptual inner join, 129, 400
conceptual outer join, 401
projection-join check, 151–154
self-join, 546–549, 580, 622, 649
join operations in relational algebra, 530–532, 539–544
comparison operators with, 540–544
cross joins (Cartesian product), 530–532
equijoin, 540–541
join condition, 540
left outer join, 544
natural inner join, 542
joins in SQL, 570–581
column-list, 572, 574
conditional, 572
cross (Cartesian product), 571
natural inner, 572, 575
outer, 572, 578–579
qualified, 571
self-joins, 579–580
syntax for, 580
table summarizing, 572
union, 572
join-subset constraint, 353, 378, 404

K

Kent, W., 25, 745
key attributes, 325, 739
key dependency, 749
key length check, 137–148
keys
alternate, 476
candidate, 5, 6, 476
composite, 477, 723
defined, 47, 138
in IDEF1X, 325
in normal form definitions, 739
primary, 6, 45, 476
proper superkeys, 138
simple, 138
unifying compatible, composite keys, 724
Kipling, Rudyard, 53
knowledge base, 30, 39
Kurzweil, Ray, 28, 880, 892

L

Law of Extensionality, 212
lazy evaluation of derived fact types, 100, 236, 521
leaf subtypes in UML, 374
left outer joins
in ORM, 130, 401
in SQL, 579
life cycles
information systems, 50
software, 49
life-buoy symbol, 228
like operator, 585–588
links in UML, 351
locally homogeneous relationship, 458
lock, 662–664
exclusive, 663–664
shared, 663–664
logical data models, 6
logical derivation check, 200–206
logical garbage, 39
logical operators, 528, 566
logical predicates. *See* predicates
lower scalar function, 565
low-level optimization. *See* denormalization and low-level optimization

M

machine code, 19
maintenance, 48, 50, 53, 56
major object type, 855
major fact type, 855
mandatory attributes
in Barker notation, 309
in IDEF1X, 325
in UML, 352
mandatory columns, 46
mandatory constraints in Barker ER, 309–311
mandatory constraints in UML, 352, 360
mandatory relationships in ER
Barker notation, 309–311
in IDEF1X, 329
mandatory roles in ORM, 162–171
defined, 162–165
derived fact types and, 200–206
disjunctive, 168–169
real world and, 165
reference schemes, 174–191
relational mapping and, 488
restraint recommended for, 166
restricted, 291, 304
subset constraints and, 226
subtypes and, 248
mandatory roles in relational schemas, 477
mandatory symbol for subtype links, 242
many-to-many (*m*:*n*) relation, 116
many-to-one (*n*:1) relation, 114
mapping
conceptual schema to relational schema, 45–47, 473–524. *See also* relational mapping
from ORM to ER or IDEF1X, 334–339
from ORM to UML, 388–393
reengineering stage, 760
Martin, James, 318, 343

match operator, 644
materialized views, 650, 754, 771
max bag function, 605, 606, 616
maximum frequency constraints, 274–275
MDX (Multidimensional Expression) language, 842
Mealy machine, 785–786, 808
membership subqueries in SQL, 595, 599
metadata, 190, 426, 664
metalanguage, 32, 817
metamodeling, 881–886
metaschema, 32, 881–886
Milner, Robin, 800
min bag function, 605, 606, 614
minimum frequency constraints, 274–275
minus sign (–), for nulls, 227
mixfix form, 68
modality, 408–409, 894
modeling process
  as-is vs. to-be model, 78
  defined, 60
modify operation, 48
modularization, 852, 888
MOLAP (multidimensional OLAP), 841
Moore machine, 785–787, 807–808
Multidimensional Expression (MDX) language, 842
multidimensional OLAP (MOLAP), 841
multiple inheritance of subtypes, 243, 317, 322, 372, 445, 516
multiplicity constraints in UML, 352, 358
multisets, 437, 471, 757, 877
multivalued attributes in UML, 354–357
multivalued dependencies (MVDs), 745
mutually exclusive roles, 228
mutually exclusive sets, 215
mutually independent attributes, (sec 12.6, 3NF discussion)
MVDs (multivalued dependencies), 745

N

n–1 rule, 138
N/CR (nest/coreference) equivalence theorem, 705
N/F (nest/flatten) equivalence theorem, 706
N/Fm (nest/flatten into many fact types) equivalence, 709
n-ary associations in UML, 358
n-ary predicates, 68
natural inner joins
  in relational algebra, 575
  in SQL, 572, 575
nchar data type, 560–561
negated subqueries in SQL, 597
Negation
  Classical, 452
  Semipositive, 452, 455, 471
  As Failure, 450, 455, 471
nest/coreference (N/CR) equivalence theorem, 705
nest/flatten into many fact types (N/Fm) equivalence, 709
nest/flatten (N/F) equivalence theorem, 706
nested normal form, 737
nested object types
  defined, 88
  relational mapping and, 495–497
  UML association classes, 362, 389, 392
  See also objectified associations
nested subqueries in SQL, 597
nesting
  in conceptual schema diagrams, 88
  conceptual schema transformations and, 702–714
network data model, 21
NFNF or NF^2 (non-first normal form), 456, 737
NIAM (Natural Information Analysis Method), 107
no action option, for **references** clause delete-action, 642–643
nominalization,
  propositional, 440–449, 471, 895
  situational, 440–449, 471
non-first normal form (NFNF or NF^2), 456, 737
nonidentifying relationships in IDEF1X, 328, 331
nonkey attributes, 343, 739
nonrepeatable read, 662, 664
nonrigid property, 427
nontransferable relationships in Barker ER, 317
normalization, 734–752
  Boyce-Codd normal form (BCNF), 744–745
  decomposition approach, 627, 735–736
  domain key normal form (DKNF), 749–750, 771–772
  elementary key normal form (EKNF), 564, 569
  fifth normal form (5NF), 747
  first normal form (1NF), 737
  fourth normal form (4NF), 745
  main normal forms, 737
  non-first normal form (NFNF or NF^2), 456, 737
  project-join normal form (PJNF), 747
  relevant constraints not captured by 5NF or DKNF, 749–752
  second normal form (2NF), 739
  synthesis approach, 735–736
  third normal form (3NF), 741–742
not operator, 584
  …between…and… operator with, 584
  exists quantifier with, 623
  in operator with, 584
  is null operator with, 589
  like operator with, 585
  subqueries with, 597
n-role constraints, 124
null set, 160, 207, 214
nulls
  is null operator, 588–590
  minus sign for, 227
  question mark for, 162
  relational mapping and percentage of, 490
  SQL three-valued logic, 567
numeric data type, 560–561
NXS (No eXclusion with a Subset constraint), 233, 295

O

object cardinality constraints in ORM, 289
Object Constraint Language (OCL), 14, 347, 397, 403
object database (ODB) or object-oriented database (OODB), 872–879, 890
object diagrams in UML, 354
object identity, 872–873
object join in ORM, 129

object language, 32
Object Management Group (OMG), 12, 55, 107, 347, 813, 833
Object Modeling Technique (OMT), 347
Object Query Language (OQL), 356, 844
object relativity, 689
object terms in predicates, 10, 67
object type absorption, 692
object type constraints. *See* value constraints in ORM
object type extraction, 692
object type generalization. *See* generalization of object types
object type population, 162–163
object type specialization. *See* specialization of object types in ORM
object type zoom, 854
object types
  duplicated, 171, 896
  generalization of, 260–266, 372
  imported from external model, 171, 208, 896
  independent, 219–221, 495
  refinement levels based on major object types, 855
objectification, 88
objectified associations
  defined, 88
  relational mapping and, 495–497
  UML association classes, 362, 389, 392
  See also nested object types
objectified relationship types. *See* objectified associations
object-oriented database, 872–879, 890
object-oriented modeling, 7, 25, 52, 397. *See also* Unified Modeling Language (UML)
object-oriented programming paradigm, 249, 395, 739, 872
Object-Oriented Software Engineering (OOSE), 347
object-relational database (ORDB) systems, 877–890
Object-Role Modeling (ORM)
  abstraction mechanisms, 835, 852
  arity, 10, 68, 88, 92
  attribute-free approach, 10, 353
  counterexamples for constraints, 11, 15–16, 120, 133, 201
  ER and UML vs., 6–19
  fact types, 9. *See also* fact types
  glossary, 893–902
  join constraints in, 377–378, 399–406, 469
  mapping to ER or IDEF1X, 335
  mapping to UML, 388–393
  metamodeling in, 881–885
  overview, 9–12
  roles, 9–10
  semantic stability, 12, 353, 847
  UML correspondences, 396
  UML multiplicities and, 360
  validation of models, 10–18, 309
objects
  complex, 871–873, 876–877, 890
  entities, 35, 66
  nested, 88
  in UML, 349
  values, 35, 65
  See also entities; values
occurrence frequency constraints in ORM, 272–275
  compound, 273
  defined, 272
  external, 275
  as local constraints, 272
  specifying frequency ranges, 274
OCL (Object Constraint Language), 14, 347, 397, 403
ODB (object database), 872–879, 890
of clause, 639
oids (object identifiers), UML reliance upon, 349, 351
OLAP (online analytical processing), 841
OLTP (On-Line Transaction Processing), 556, 836
OMG (Object Management Group), 12, 55, 107, 347, 813, 833
OMT (Object Modeling Technique), 347
on clause, for joins, 572–578
one-to-one relation, 115
ONF (Optimal Normal Form), 484
online analytical processing (OLAP), 841
Online Transaction Processing (OLTP), 556, 836
Ontology, 427, 864–871, 889–891
OODB. *See* object database (ODB) or object-oriented database (OODB)
"OODBMS Manifesto, The," 872–873
OOSE (Object-Oriented Software Engineering), 347
Open World Assumption (OWA), 451–452, 471, 867
Open World with Negation (OWN) assumption, 452–453, 471, 511, 524
Optimal Normal Form (ONF), 484
optimization. *See* conceptual schema optimization; denormalization and low-level optimization
optional roles in ER
  Barker notation, 310–311
  IE notation, 319–320
optional roles in ORM, 162–171
optional roles in UML, 352, 360
OQL (Object Query Language), 356, 844
Or
  inclusive, 165–169
  exclusive, 229, 248, 314, 322, 370–371
ORDB (object-relational database) systems, 877–890
order
  of columns and rows, 46
  of elements in sets, 213
order by clause, 567–569
ordered sets in UML, 386, 437
ORM. *See* Object-Role Modeling (ORM)
ORM glossary, 893–902
orthogonality
  of a language, 61, 315
  of SQL, 595
Other subtype in Barker ER, 316–317
outer joins
  in ORM, 401–402
  in relational algebra, 544
  in SQL, 572, 578–580
output reports, 7, 9, 64, 78
overlap algorithm, 712–714, 724
overlap of sets, 215
overlapping
  entity types, 102
  sets, 215
  subtypes in ORM, 252, 264
  subtypes in UML, 373
OWA (Open World Assumption), 451–452, 471, 867
OWL (Web Ontology Language), 865–870
OWN (Open World with Negation) assumption, 452–453, 471, 511, 524

P

pair-equality constraints, 232, 711
pair-exclusion constraints, 18, 232–233, 290, 315, 371, 711
pair-subset constraints, 231, 234, 297, 330, 578, 709, 711, 719, 720, 725, 750, 752, 755, 757, 769
parsimony of a language, 61–62, 106
partition, 95, 229, 242, 247, 316, 375
patterns
  decreasing disjunctions, 429–431, 470
  role playing, 429–431, 470
pattern matching, **like** operator for, 585–588
percent sign (%), as **like** operator wildcard, 585
performance. *See* conceptual schema optimization; denormalization and low-level optimization
period events, 416–417
Petri, Carl Adam, 796
Petri net, 796–800, 833
phantom, 662
physical optimization techniques, 758
π−calculus (pi calculus), 800, 833
PJNF (project-join normal form), 747
point events, 416
polymorphism, 372
popular reference mode, 84. 105
population
  of conceptual schema diagrams, 84–88
  of object types, 162–163
  as ORM validation method, 10, 104
  of roles, 161
  UML vs. ORM, 354
  See also significant population
postrelational databases, 871–880
  deductive databases, 878
  distributed database systems, 880
  eXtensible Markup Language (XML) and, 880
  federated databases, 880
  object database (ODB) or object-oriented database (OODB), 872–876
  object-relational database (ORDB) systems, 877–878
  spatial databases, 878–879
  temporal database systems, 879
powertypes in UML, 456–461, 465, 472
pragmatics of a language, 60
predefined types in SQL, 561
Predicate Specialization / Generalization theorem 1 (PSG1), 693–697, 729
Predicate Specialization / Generalization theorem 2 (PSG2), 698, 714, 718, 724, 730
Predicate Specialization / Generalization theorem 3 (PSG3), 700, 723
predicates
  acyclic relationships, 286
  antisymmetric relationships, 282
  asymmetric relationships, 281, 284
  binary, 67, 83, 113–121
  defined, 67
  intransitive relationships, 284–285
  inverse, 10, 69, 82, 87
  irreducibility and, 70
  irreflexive relationships, 280–281
  mixfix form for, 68
  n-ary, 68
  object terms in, 10, 67
  objectified, 88
  quaternary, 89, 92, 126, 143, 149
  reflexive relationships, 278
  ring, 86–87, 277–287, 292, 371
  specialization and generalization, 692–700
  surface vs. deep structure, 69
  symmetric relationships, 279
  ternary, 87, 123
  transitive relationships, 279
  unary, 86, 111
  See also specialization and generalization of predicates
Predicator Set Model (PSM), 107, 432
preferred identifier, 175
  compound reference scheme, 177
  disjunctive reference scheme, 187–189
  guidelines for choosing, 181–185
  simple reference scheme, 174–175
preferred reference. *See* preferred identifier.
preferred identifier. *See* preferred identifier.
primary keys
  in DDL, 639
  defined, 46, 476
  in IDEF1X, 325
  in relational schemas, 476
primitive entity types, 95–96, 105
primitive object types, 262
priority of scalar operators in SQL, 605
privileges. *See* access rights
"problem of induction", 253
procedural languages, 20, 558, 871
process modeling, 774–832
  activity diagrams, 803–806
  BPEL (Business Process Execution Language), 813–814, 818, 833
  BPDM (Business Process Definition Metamodel), 813–817, 832–833
  BPMN (Business Process Modeling Notation), 783, 813–819, 823–826, 831–833, 857
  context diagrams, 861
  data flow diagrams, 860–861
  dynamic behavior, 774–777
  executable UML, 811–812
  foundations, 795–800
  function trees, 860
  Petri net, 796–800, 833
  standard process patterns, 819–829
  state machine diagrams, 806–811
  use case diagrams, 801–803
  workflow, 777–785
process-oriented perspective for conceptual design, 51
projection operation, 150, 536
projection–join check, 152
project–join normal form (PJNF), 747
Prolog, 35, 47, 65, 300
proper names, 66, 106
proper overlap of sets, 215
proper subsets, 214, 268
proper subtypes, 372, 646
proper superkeys, 138
proper supersets, 214
proper supertypes, 241
proposition-forming operators, 214
  relational algebra logical operators, 528
  set operators, 214
propositions vs. sentences, 34
prototyping, 50, 52, 56, 858
pseudostate, 807–809
PSG1 (Predicate Specialization / Generalization theorem 1), 693–697, 729
PSG2 (Predicate Specialization / Generalization theorem 2), 698, 714, 718, 724, 730

PSG3 (Predicate Specialization / Generalization theorem 3), 700, 723
PSM (Predicator Set Model), 107, 432

Q

QBE (Query-By-Example), 23, 474, 528, 554, 844
QBF (Query-By-Form), 843
qualified associations in UML, 363–364
qualified joins in SQL, 571
qualified names, 532
quantified comparison subqueries in SQL, 595
quaternary fact types, 126
queries
  conceptual query languages, 843–852
  examples for five generations of languages, 19
  point-to-point, 847
  recursive, 47, 626–629
  relational algebra query strategies, 546–550
  semantically optimized, 850
  SQL guidelines, 579–581
  subqueries in SQL, 594–601, 619–625
  See also conceptual schema optimization; select statement; SQL; subqueries in SQL
query optimizer, 548, 754
Query-By-Example (QBE), 23, 474, 528, 554, 844
Query-By-Form (QBF), 843
question mark (?), for nulls, 162

R

RDF (Resource Description Framework), 865–867
RDFS (Resource Description Framework Schema), 867
real world
  mandatory roles and, 166
  uniqueness constraints and, 120
  vs. recorded-world, 120, 166
recursion. *See* recursion.
recursive queries, 626–629
redefinition of associations in UML, 387
redundancy
  avoided in relational mapping, 484
  in bad database design, 5
  bags vs. sets and, 112
  between tables, 756
  controlled, 297, 755–757
  derived, 297
  final checks in ORM, 297–298
reengineering the conceptual schema, 59, 761–767
  conceptualization stage, 760
  data migration stage, 691, 766–767
  forward engineering, 760
  mapping stage, 760
  reverse engineering, 59, 760
  steps in, 760
  transformation stage, 760
Reference and Idea Language (RIDL), 106, 845–846
reference modes
  of entities, 67
  general, 85
  popular, 84–85
  unit-based, 85
reference schemes, 174–191
  candidate identifiers, 180, 189, 208
  compound, 177–178
  context-dependent reference, 265–266, 322, 376, 518
  disjunctive reference, 187–189, 208–209, 402, 510, 522, 526
  guidelines for choosing, 181–185
  mandatory 1:1 reference types, 174
  parenthesized reference modes and, 181
  rigid identifier, 181, 208
  simple 1:1 scheme, 174–175
  synonyms, 189, 208
reference tables for value constraints, 219
reference types
  in ORM, 84, 175
  in SQL, 561
references clause, 639–644
  delete-action, 643–644
  match operator with, 643–644
  update-action, 643
referential actions, 643
referential integrity constraints
  in relational schemas, 479
  See also subset constraints
referential integrity in SQL, 642–646
referential integrity rule, 479
referential roles, 161
refinement levels based on major object types, 855–857
reflexive relationships, 278
reification, 88, 363, 439
regular identifiers in SQL, 558–559
relational algebra, 528–550
  assignment operation, 537–538
  basic table operations, 632
  Cartesian product (unrestricted join), 530–532
  comparison operators, 528
  difference operation, 530
  division operation, 545–550
  intersection operation, 529
  join operation, 539–544
  laws for relational operators, 633
  logical operators, 528
  projection operation, 536–539
  query strategies, 546–550
  selection operation, 532–536
  self joins, 548
  union operation, 529
relational data model, 21–22, 554–556
relational databases, 45–47
relational DBMS, 554–556
relational division in SQL, 617
relational mapping, 483–524
  1:1 associations, 487–492
  absorption of functional fact types, 497–499, 513
  advanced aspects, 511–524
  advanced subtyping, 513–520
  basic grouping, 484–485
  basic subtyping, 497–499
  derived facts, 497, 521
  disjunctive reference, 522–524
  equality constraints, 486
  external uniqueness constraints, 492–494
  independent object types, 495–496
  mandatory roles, 477, 485–486
  meaningful table and column names, 485
  objectified associations, 494–497
  open/closed world aspects, 510–512
  partition of functional fact types, 513–515
  percentage of nulls, 490
  procedure, 499, 510
  reference predicates, 512
  separation of functional fact types, 513–515
  step 0 refinements, 510
  subtyping, 497–499, 513–520
  total table procedure, 518

relational OLAP (ROLAP), 841
relational schemas, 475–482
attributes, 475
constraints in, 476
defined, 45, 475
derivation rules in, 47
domains, 475
entity integrity rule, 479, 522, 525
external uniqueness constraints in, 492–494
horizontal layout notation, 475–477
mandatory roles and, 477
mapping conceptual schema to, 483–524
referential integrity constraints in, 479, 455
referential integrity rule, 479
table scheme, 475
uniqueness constraints in, 476
vertical layouts for, 480
See also relational mapping
relational style for derivation rules, 99
relationships
in Barker notation, 317
in IDEF1X, 325–332
in IE notation, 319
in ORM. *See* fact types; predicates
objectification of, 88, 362
in UML, 357–363
relative complement operation. *See* difference operation
requirements analysis, 50–51
requirements specifications, 51
reserved words in SQL, 558–559
Resource Description Framework (RDF), 865–867
Resource Description Framework Schema (RDFS), 867
restrict option
of **alter table** statement, 644
of **drop table** statement, 645
of **drop view** statement, 646–647
restricted mandatory roles, 291, 304
restricted Turing Tests, 28
restricted uniqueness constraints, 291, 853
restriction operation. *See* selection operation
reuse of design patterns, 51, 56
reverse engineering, 59, 760
revoke statement, 665
RIDL (Reference and Idea Language), 106, 845–846
right outer joins in SQL, 572, 579
rigid identifier, 181, 208
rigid property, 427
rigid subtype, 427–428
ring associations,
illegal associations in Barker notation, 312
in IDEF1X, 333
in ORM, 86–87, 277–287, 292, 371
in UML, 377
ring constraints in ORM, 277–287
acyclic, 285
antisymmetric, 282
asymmetric, 281
Euler diagram for, 286
intransitive, 284–285
irreflexive, 280
notation, 901
purely reflexive, 287, 901
reflexive, 278
symmetric, 279
transitive, 279
ring constraints in UML, 377
ring fact types, 86–87, 277–287, 292, 371
Ritson, Peter, 188, 526
Rmap procedure. *See* relational mapping
ROLAP (relational OLAP), 841
role
in conceptual schema diagrams, 83
defined, 9, 83
fact, 161, 207
functional, 200
mandatory, 162–171
optional, 162–171
population of, 161
referential, 161
names in Chen ER, 306
names in IDEF1X, 333
names in ORM, 98, 105, 357
names in UML, 357
subtype, 427–428
See also mandatory roles
role cardinality constraint, 289, 303, 724
role names
in Chen ER, 306
in IDEF1X, 333
in ORM, 98, 105, 357
in UML, 357
role subtype, 427–428
role type, 427
rollback statement, 630
root node of subtype graphs, 243, 513
root supertype in UML, 374
Ross, Ron, 309
routines in SQL, 655–657
row types, 646
rows
choosing in SQL, 567
deleting in SQL, 631
grouping in SQL, 611–618
inserting in SQL, 629
ordering in SQL, 567–569
updating in SQL, 631
Russell's paradox, 461, 469, 867

S

SBVR (Semantics of Business Vocabulary and Business Rules), 107, 408, 410, 449, 871
scalar operators in SQL, 602–605
for concatenating strings, 604
priority, 605
schema abstraction mechanisms, 835, 852–857
display toggles, 853
layering, 853
modularization, 852–853
object type zoom, 854
refinement levels based on major object types, 855–857
schema integration, 853
schema transformations. *See* conceptual schema transformations
search conditions
…between…and… operator, 584–585
defined, 564
in operator, 582–584
is null operator, 588–590
like operator, 585–588
second normal form (2NF), 739
secondary keys in relational schemas, 476
security
external schema and, 31
in SQL, 474, 664–666
See also access rights
select statement
bag functions with, 602–611
BNF syntax for, 634
choosing columns, 562–564
choosing rows, 564–567
cross joins, 572–573

grouping rows, 611–618
natural inner joins, 572, 575
ordering columns, 498
ordering rows, 567–569
outer joins, 572, 578–580
recursive queries, 626–629
scalar operators with, 602–605
search conditions, 564
self-joins, 579–580
subqueries,
correlated, 595, 619–625, 836
existential, 595, 623–635
membership, 595, 599
union operator, 591–593
See also SQL
selection operation in relational algebra, 532–536
self-joins
in relational algebra, 548
in SQL, 579–580
semantic clarity, 847
semantic independence, 848
semantic names, 182
semantic relevance of a language, 61
semantic stability
with ConQuer, 848–849
of a language, 61
of ORM, 12
semantic web, 865
semantically optimized SQL queries, 850
semantics of a language, 60
Semantics of Business Vocabulary and Business Rules (SBVR), 107, 408, 410, 449, 871
Semiclosed World Assumption (SWA),
semiderived fact type, 99, 204–205, 209, 895
semiderived subtype, 250, 895
sentences
propositions vs., 34
surface vs. deep structure, 69
separation approach for relational mapping of subtypes, 513–515
SEQUEL (Structured English Query Language), 556
sequences
defined, 213
in SQL, 658–659
set comparison constraints between, 230–234
set default option for **references** clause delete-action, 643–644
set null option for **references** clause delete-action, 643–644
set oriented languages, 20
set theory, 212–216
bags or multisets, 213
defining sets, 212–213
elements or members of a set, 212
equality or identity of sets, 212
Euler diagrams, 214–215
extension of a set, 212
extensional definition, 213
Hypothetical Euler Diagrams (HEDs), 215
intensional definition, 213–214
Law of Extensionality, 212–213
mutually exclusive or disjoint sets, 215
null set, 214
order of elements and, 213
overlap of sets, 215
proper overlap of sets, 215
proper subsets, 215
proposition-forming operators, 214
sequences, 213
set-forming operations, 215–216
subsets, 214
supersets, 214
Venn diagrams, 216
set union operator, 162, 215–216
set-comparison constraints, 224–234
constraint implication theorems for, 234
in UML, 364–372
See also subset constraints, equality constraints, exclusion constraints
set-forming operations, 215–216
sets. *See* set theory.
shared aggregation in UML, 379
significant example sets, 76
significant population
for binary fact types, 114
for subtypes, 253–254
simple 1:1 reference scheme, 174–175
simple key, 138
simple option of **match** operator, 643
simple uniqueness constraints, 124
simplicity
of interface, 863
of a language, 60–61
single inheritance of subtypes, 243
single-valued attributes in UML, 351–353
smallint data type, 560–562
snapshot fact types, 111, 419
some quantifier, 595
spanning rule, 145
spatial databases, 871–872, 878
specialization and generalization of predicates, 692–700
PSG1 (Predicate Specialization / Generalization theorem 1), 693–697, 729
PSG2 (Predicate Specialization / Generalization theorem 2), 698, 714, 718, 724, 730
PSG3 (Predicate Specialization / Generalization theorem 3), 700, 723
specialization of object types in ORM
generalization vs., 260
procedure, 251
See also subtypes
specialization of object types in UML, 372
splitting
n–1 rule and, 143
nesting vs., 88, 146
projection-join check for, 152
redundancy check for splittability, 145
sufficient conditions for, 150
SQL, 556–686
as 4GL, 19
aliases, 580, 532, 613
arithmetic operators, 602–605
bag functions, 602–611
…between…and… operator, 584–585
case in, 563
choosing columns, 562–564
choosing rows, 564–567
comments, 581
correlated subqueries, 595, 619–625, 836
create table statement, 481, 638–642, 684
cursor, 660–664
data definition language (DDL), 47, 558, 638
data types, 560–562
except operator, 593–594
existential subqueries, 595, 623–625
functions, 602–611
generated columns, 646
grouping, 611–618
identifiers, 558–560

identity column, 641–642, 658
in operator, 582–584
intersect operator, 529, 593
is null operator, 588–590
joins, 570–581
like operator, 585–588
metadata, 190, 426, 664
nulls, 567, 588–590,
ordering columns, 564
ordering rows, 567–569
orthogonality of, 595
recursive queries, 626–629
regular identifiers, 588–560
relational division, 617
reserved words, 558–559
routines, 655–657
scalar operators, 602–605
search conditions, 564, 582–590
security, 474, 664–666
sequence, 658–659
standards, 556–567
stored procedure, 656–657
string constants, 559
sublanguages, 557–558
subqueries,
correlated, 595, 619–625, 836
existential, 595, 623–635
comparison, 595
joins replacing, 596–597
membership, 595, 599
nested, 597
quantified comparison, 595
three-valued logic, 567
triggers, 652–654
union operator, 591–593
updating table populations, 629–631
user defined function (UDF), 655–657
with clause, 626–629
See also **select** statement
SQL/XML, 666–675
standard units for conversion rules, 190, 208
star schema, 840–841
state machine
diagram in UML, 586–811
Mealy machine, 785–786, 808
Moore machine, 785–787, 807–808
state model, 785–795
state transition diagrams, 425, 788
statecharts, 426, 803, 806
static constraints, 33, 56, 111
Stonebraker, M., 873–874
string constants in SQL, 559
Structured English Query Language (SEQUEL), 556
structured types, 646
subject world, 48
subqueries in SQL,
correlated, 595, 619–625, 836
existential, 595, 623–635
comparison, 595
joins replacing, 596–597
membership, 595, 599
nested, 597
quantified comparison, 595
subschemas
conceptual, 31, 51, 853
integration of, 51, 62, 104
large applications and, 62
subset constraints in DDL, 639–641
subset constraints in ORM, 225–226, 229–234, 366–368
See also set theory
subset constraints in UML, 364–368
subsets
defined, 214–215
proper, 214–215
substring scalar function, 608–609
subtables, **under** clause defining, 639
subtypes,
asserted subtypes, 249–250, 893–895
basic ORM theory, 238–266, 428–430
Barker ER notation, 316–317
categories in IDEF1X, 332–333
context-dependent reference and, 265–266
definitions of subtypes, 245
derived subtypes, 245
Euler diagrams for, 247, 252
exclusion constraints and, 242
exhaustion constraints and, 242
generalization procedure, 266
IE (Information Engineering) notation, 322
indirect, 242–243, 269
migration between subtypes, 429–430
multiple inheritance, 243, 317, 322, 372, 445, 516
overlapping subtypes in ORM, 252, 264
overlapping subtypes in UML, 373
“problem of induction” and, 253
proper, 241
reasons for, 238–239
relational mapping for, subtyping, 497–499, 513–520
rigid, 427–428
role, 427–428
semiderived, 250, 895
set-comparison constraints replaced by, 246
specialization procedure, 251
UML subtyping, 372–376
root node, 243, 374, 513
top-down vs. bottom-up approach, 260
successive states, dynamic constraints and, 425, 470
sum bag function, 605–606
superkeys, proper, 138
supersets, 214
supertypes
generic entities in IDEF1X, 332–333
proper, 241
See also generalization of object types
surface structures of sentences, 69
surrogate identifiers, 406, 424
symmetric 1:1 cases, relational mapping and, 487–492
symmetric relationships, 279
synonyms
for standard terms, 77
transitivity, 283–284
syntax of a language, 60
synthesis approach to normalization, 735–736
system tables in SQL, 665
system world, 48

T

table scheme, 401, 475
table width guideline (TWG), 727–729
tables
aliases for, 532
alter table statement, 644–645, 691
base tables, 46, 684
conceptual fact tables, 11, 45, 84
create table statement, 481, 638–642, 684
drop table statement, 631, 645
meaningful names for, 414

reducing number of mapped tables, 723–726
in relational databases, 45–46
simplifying individual tables, 727–7325
system tables in SQL, 665
updating table populations, 629–631
views, 46, 646–651
working tables for queries, 646
taxonomies, 239, 246, 249, 375, 869
Tbox (terminology box), 867
telephone heuristic, 68, 89
temporal database systems, 879
temporality, 411–430
temporal granularity, 412–423
ter Hofstede, Arthur, 107, 819
terminology box (Tbox), 867
ternary fact types, 87, 123
testing, 53
textual constraints in ORM, 290–293
textual constraints in UML, 349, 355
"Third Generation DBMS Manifesto", 873
third generation language (3GL), 406
third normal form (3NF), 741–742
Thomsen, E., 843
three-valued logic in SQL, 567
time
transaction, 412, 419, 470, 879
valid, 412, 419, 470, 489
time deictic facts, 418–419, 430, 461, 470
to-be model, 78–79
total roles. *See* mandatory roles
total table procedure for relational mapping, 518
traceability of requirements specifications, 50
tractability of a language, 61, 870
transactions, 36
compound, 37
simple, 36
transaction time, 412, 419, 470, 879
transformations. *See* conceptual schema transformations
transition constraints in ORM, 426, 431
fact tables for, 426
guarded transition diagrams, 426
state transition diagrams, 425
transitive relationships
in predicates, 203, 279
in subtyping, 242
triggers, 652–654
tuples, 46
tuple-subset constraints, 232, 268, 771
Turing Test, 28
TWG (table width guideline), 727–729
types, 212, 456

U

UDF (user defined function), 655–657
UDT (user defined type) in SQL, 560–561
Ullman, J., 737
UML. *See* Unified Modeling Language (UML)
UML glossary, 907–909
UMLmap procedure, 388–389
unary fact types, 86, 111
unary predicates, 86, 111
unary relationships
defined, 86
UML and, 357
unary-binary expansion, 719–720
under clause defining subtables, 639
underscore (_), as **like** operator wildcard, 585
Unified Modeling Language (UML), 346–397, 800–812
activity diagrams, 803–806
aggregation in, 379–382, 396
association classes in, 362, 389, 392
associations in, 357–363
association redefinition, 387
changeability, 384–385
classes in, 348–351
composition (composite aggregation), 379
constraints for subtyping, 372–376
derivation rules in, 380, 391–392
diagram types, 347, 801
discriminator label in, 375, 396
enumerations in, 375
exclusive-or constraints in, 370–371
executable UML, 811–812
glossary, 907–909
initial values in, 382
instances in, 354
links in, 351
mapping from ORM, 388–389
multiplicity constraints in, 352, 358
multivalued attributes in, 354–357
nulls in, 354
object diagrams, 354
objects in, 351
oids relied upon by, 351
{ordered} constraint in, 386, 437
ordered sets in, 386, 437
ORM correspondences, 396
overview, 346–348
set-comparison constraints, 364–372
shared aggregation in, 379–380
single-valued attributes in, 351–354
state machine diagram, 586–811
subset constraints in, 364–368
subtyping in, 372–376
textual constraints in, 349, 355
use case diagrams, 801, 860, 889
value constraints in, 375
Uniform Resource Identifier (URI), 669, 674, 865
Uniform Resource Locator (URL), 865
Uniform Resource Name (URN), 865–866
union joins in SQL, 572
union of two sets, 215
union operation
as associative, 530
as commutative, 530
on sets, 215
on tables, 529
union operator in SQL, 591–593
unique constraints in SQL, 639
unique option of **match** operator, 644
uniqueness constraints in ORM, 111–136
CSDP step 4 overview, 110–111
external, 128–136
functional dependencies (FDs) and, 143–144
internal, 111–127
key length check, 137–148
local nature of, 163
longer fact types and, 122–127
mandatory roles with, 164
real world and, 121
relational mapping for external uniqueness constraints, 492–494
restricted, 291, 853
simple, 124
spanning rule, 145
See also external uniqueness constraints in ORM

uniqueness constraints in relational schemas, 476
unit-based reference modes, 85
universe of discourse (UoD), 6, 23, 64
unrestricted join. *See* Cartesian product (cross joins)
UoD. *See* universe of discourse (UoD)
update, usage in this book, 48
update statement, 631
update-action for **references** clause, 643
updating
  table populations, 629–631
upper scalar function, 565
URI (Uniform Resource Identifier), 669, 674, 865
URIref (Uniform Resource Identifier Reference), 866
URL (Uniform Resource Locator), 865
URN (Uniform Resource Name), 865–866
usage world, 48
use case diagrams, 801, 860, 889
user interface. *See* external interface
user defined function (UDF), 655–657
user defined type (UDT), 560– 561
using clause, 571, 574, 577

V

validation of ORM models, 10–11
validation rules. *See* constraints
valid time, 412, 419, 470, 489
value-comparison constraints, 290
value concatenation/separation, 719
value constraints
  in ORM, 216–220
  in UML, 375
value types, 10, 82
values
  default, 189, 382–384, 396, 522
  defined, 65, 95
  entities vs., 65–66, 95
  initial values in UML, 382
Van der Aalst, Wil, 819, 832–833
varchar data type, 560–561
Venn diagrams, 215–216
verbalization, 9–11, 44
views
  alter view statement, 647
  create view statement, 646–651
  in SQL, 646–651
  defined, 46
  drop view statement, 558, 646
  in IDEF1X, 324
  indexed, 754, 758
  materialized, 650, 754, 771

W

weak entity types, defined, 307
Web Ontology Language (OWL), 865–870
where clause, 564, 634
with clause, 626–629
with time zone option in SQL, 561
working tables for queries, 646, 754,
workflow, 778–785, 798, 813, 827

X

XML (eXtensible Markup Language), 666–675
XML schema, 669–674, 866, 880
XQuery, 675–683

Y

"yama" (Yet Another Modeling Approach) syndrome, 346
YAWL (Yet Another Workflow Langauge), 798

Z

Zachman framework for enterprise architecture, 53–57
Zachman, John, 53, 57

About the Authors

Dr. Terry Halpin, BSc, DipEd, BA, MLitStud, PhD, is a Distinguished Professor in Computer Science at Neumont University. His industry experience includes several years in data modeling technology at Asymetrix Corporation, InfoModelers Inc., Visio Corporation, and Microsoft Corporation. His doctoral thesis formalized Object-Role Modeling (ORM/NIAM), and his current research focuses on conceptual modeling and conceptual query technology. He has authored over 150 technical publications and five books, and has co-edited three books on information systems modeling research. He is an editor or reviewer for several academic journals, is a regular columnist for the *Business Rules Journal*, and has presented at many international conferences. Dr. Halpin is the recipient of the DAMA International Achievement Award for Education (2002) and the IFIP Outstanding Service Award (2006).

Dr. Tony Morgan is a Distinguished Professor in Computer Science at Neumont University, where he is involved in teaching at undergraduate and postgraduate levels and in research into future information systems. He has extensive experience of the practical issues involved in creating and delivering software systems, gained in companies such as EDS and Unisys. His main technical interest is in the use of model driven development to build information systems that deliver real business value.