

国外优秀信息科学与技术系列教学用书

# 数据库系统

## ——面向应用的方法

(第二版 影印版)

**DATABASE SYSTEMS**  
**An Application-Oriented Approach**  
**(Second Edition)**

■ Michael Kifer  
Arthur Bernstein  
Philip M. Lewis



高等 教育 出 版 社  
Higher Education Press

# 一流的品质 优惠的价格

## Database Systems: An Application-Oriented Approach (Second Edition)

Designed for students learning databases for the first time, *Database Systems: An Application-Oriented Approach, Introductory Version*, Second Edition, presents the principles underlying the design and implementation of databases and their applications. The book consists of nine core chapters, including separate chapters on triggers (chapter 7) and using SQL in an application (chapter 8) that recognize the growing importance of application development in building database systems. Additional chapters (chapters 11–17) cover database tuning, transaction processing, query processing, object-oriented databases, and XML databases and provide a variety of ways to enrich students' introduction to databases.

### Features of the Second Edition

- An application-oriented introduction to database concepts
- SQL updated to the latest standard
- Coverage of both Entity-Relationship modeling and the Unified Modeling Language
- Discussions of software-engineering issues related to implementing transaction-processing applications
- Detailed case studies providing hands-on experience in application design and programming
- In-depth coverage of XML, object-oriented databases, and database tuning

### About the Authors

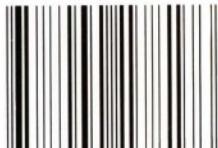
**Michael Kifer** is a professor in the Department of Computer Science at the State University of New York at Stony Brook. His interests include database systems, knowledge representation, and Web information systems. He has published and edited several books and many articles in these areas, including award-winning works on F-logic and object-oriented database languages.

**Arthur Bernstein** is also a professor in the Department of Computer Science at the State University of New York at Stony Brook. His research focuses on transaction processing, Web services, and concurrency, and he has published numerous articles in these areas. This is Professor Bernstein's second textbook.

**Philip Lewis** is a leading professor in the Department of Computer Science at the State University of New York at Stony Brook. With interests in database systems, transaction processing, and concurrency, he has published four textbooks and many articles in these areas. Professor Lewis is also the founding editor of the SIAM Journal of Computing.

For more information, please visit [www.PearsonEd.com](http://www.PearsonEd.com).

ISBN 7-04-017817-6



9 787040 178173 >

定价 49.50 元



PEARSON  
Addison  
Wesley

For sale and distribution in the People's Republic of China exclusively (except Taiwan, Hong Kong SAR and Macao SAR).

仅限于中华人民共和国境内(不包括中国香港、  
澳门特别行政区和中国台湾地区)销售发行。

国外优秀信息科学与技术系列教学用书

# 数据库系统 ——面向应用的方法

(第二版 影印版)

## DATABASE SYSTEMS An Application-Oriented Approach (Second Edition)

Michael Kifer

Arthur Bernstein

Philip M. Lewis



高等教育出版社

# 图字 :01-2005-0965 号

**Database Systems: An Application-Oriented Approach, Introductory Version, Second Edition**

Michael Kifer, Arthur Bernstein, Philip M. Lewis

本书封面贴有 Pearson Education(培生教育出版集团)激光防伪标签。无标签者不得销售。

English reprint edition copyright © 2005 by PEARSON EDUCATION ASIA LIMITED and HIGHER EDUCATION PRESS. ( Database Systems: An Application-Oriented Approach, Introductory Version, 2e from Pearson Education's edition of the Work )

**Database Systems: An Application-Oriented Approach, Introductory Version, 2e** by Michael Kifer, Arthur Bernstein, Philip M. Lewis, Copyright © 2005.

All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Pearson Education, Inc.

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Regions of Hong Kong and Macau).

原版 ISBN: 0-321-22838-3

**For sale and distribution in the People's Republic of China exclusively (except Taiwan, Hong Kong SAR and Macao SAR).**

仅限于中华人民共和国境内(但不允许在中国香港、澳门特别行政区和中国台湾地区)销售发行。

## 图书在版编目(CIP)数据

数据库系统:面向应用的方法 = Database Systems:  
An Application - Oriented Approach : 第 2 版 / ( 美 )  
基弗 (Kifer, M.) , (美)伯恩斯坦 (Bernstein, A.) ,  
(美)刘易斯 (Lewis, P. M.) . —影印本. —北京:  
高等教育出版社, 2005. 12  
ISBN 7 - 04 - 017817 - 6

I. 数... II. ①基... ②伯... ③刘... III. 数据  
库系统 - 高等学校 - 教材 - 英文 IV. TP311. 13

中国版本图书馆 CIP 数据核字 (2005) 第 134986 号

出版发行	高等教育出版社	购书热线	010 - 58581118
社 址	北京市西城区德外大街 4 号	免费咨询	800 - 810 - 0598
邮 政 编 码	100011	网 址	<a href="http://www.hep.edu.cn">http://www.hep.edu.cn</a>
总 机	010 - 58581000	网上订购	<a href="http://www.landraco.com">http://www.landraco.com</a>
经 销	蓝色畅想图书发行有限公司	畅想教育	<a href="http://www.widedu.com">http://www.widedu.com</a>
印 刷	北京民族印刷厂		
开 本	787 × 1092 1/16	版 次	2005 年 12 月第 1 版
印 张	43.25	印 次	2005 年 12 月第 1 次印刷
字 数	830 000	定 价	49.50 元

本书如有缺页、倒页、脱页等质量问题,请到所购图书销售部门联系调换。

**版权所有 侵权必究**

**物料号 17817 - 00**

# 出版说明

20世纪末,以计算机和通信技术为代表的信息科学和技术对世界经济、科技、军事、教育和文化等产生了深刻影响。信息科学技术的迅速普及和应用,带动了世界范围信息产业的蓬勃发展,为许多国家带来了丰厚的回报。

进入21世纪,尤其随着我国加入WTO,信息产业的国际竞争将更加激烈。我国信息产业虽然在20世纪末取得了迅猛发展,但与发达国家相比,甚至与印度、爱尔兰等国家相比,还有很大差距。国家信息化的发展速度和信息产业的国际竞争能力,最终都将取决于信息科学技术人才的质量和数量。引进国外信息科学与技术优秀教材,在有条件的学校推动开展英语授课或双语教学,是教育部为加快培养大批高质量的信息技术人才采取的一项重要举措。

为此,教育部要求由高等教育出版社首先开展信息科学与技术教材的引进试点工作。同时提出了两点要求,一是要高水平,二是要低价格。在高等教育出版社和信息科学技术引进教材专家组的努力下,经过比较短的时间,第一批由教育部高等教育司推荐的20多种引进教材已经陆续出版。这套教材出版后受到了广泛的好评,其中有不少是世界信息科学技术领域著名专家、教授的经典之作和反映信息科学技术最新进展的优秀作品,代表了目前世界信息科学技术教育的一流水平,而且价格也是最优惠的,与国内同类自编教材相当。这套教材基本覆盖了计算机科学与技术专业的课程体系,体现了权威性、系统性、先进性和经济性等特点。

在引进教材的同时,我们还应做好消化吸收,注意学习国外先进的教学思想和教学方法,提高自编教材的水平,使我们的教学和教材在内容体系上,在理论与实践的结合上,在培养学生的动手能力上能有较大的突破和创新。

希望这些教学用书的引进出版,对于提高我国高等学校信息科学技术的教学水平,缩小与国际先进水平的差距,加快培养一大批具有国际竞争力的高质量信息技术人才,起到积极的推动作用。同时也欢迎广大教师和专家们对教材引进工作提出宝贵的意见和建议。联系方式:hep.cs@263.net。

高等教育出版社  
二〇〇四年十月

# 前　　言

此教材的第二版将会发行两个版本：

- 本书的版本，主要是入门性内容，适合本科生或研究生的数据库初级课程。

- 另一个版本则是完全版，适合以下三方面的课程。

- 数据库入门课程，面向本科生或研究生。

- 事务处理方面的课程，面向学习过数据库入门性课程的本科生或研究生。

- 关于数据库的本科生高级课程或者研究生初级课程，面向学习过数据库入门性课程的学生。

我们的目标之一就是减少入门版本的篇幅，使学生更容易接受。另一个目标是在使用本书第一版所获经验的基础上，尽量使本版本更加完善。

本书的章节不仅仅是完全版的子集。我们认为讲授数据库入门课程的教师应该选择增加对象数据库和 XML(完全版中许多章节详细叙述的主题)，以丰富这门课程的内容。因此在本书中，我们新加了两章，第 16 章“对象数据库绪论”和第 17 章“XML 与 Web 数据绪论”，都是从本书完全版中经过适当挑选出来的内容。

为了使本书能与迅速发展的技术保持同步，我们在本书两个版本的许多章节中加入了大量关于 UML 的内容，并且新加了一章关于数据库优化的内容(第 12 章)。

正如第一版，我们主要关注如何利用数据库来创建应用程序，而不是如何创建数据库管理系统本身。我们相信，开发应用程序的学生比创建 DBMS 的学生要多得多。因此，本书加入了大量关于通过事务访问数据库的语言(如嵌入式 SQL、ODBC 和 JDBC)和 API 的内容。

尽管本书涵盖了许多关于数据库和事务处理应用程序的实用内容，但是我们更关心这些主题的基本概念，而不是特定的商用系统或应用程序的细节。因此，我们把重点集中在关系和对象数据模型的概念上。即使在 SQL 过时很久之后，这些概念仍然是数据库处理的基础。

为了加强学生对技术内容的理解，我们增加了一个事务处理应用程序的案例，即贯彻全书的“学生注册系统”。尽管一个学生注册系统可能并不吸引人，但它有一个独特的优势，即所有学生都曾作为用户和这样的系统进行过交互。更重要的是，“麻雀虽小，五脏俱全”，因此我们可以用它来阐述在数据库设计、查询处理和事务处理等方面的诸多问题。

本书的一个重要特点就是将“学生注册系统”作为例子，介绍了用来实现事务处理应用程序所需的软件工程概念。由于许多信息系统的实现因为项目管理不善和软件工程的欠缺而最终夭折，因此我们认为这些内容应成为学生教育中一个重要组成部分。由于许多学生还需要在这方面另修专门的课程，因此本书对软件工程问题的处理是简明扼要的。然而，我们相信，如果学生们看到这些内容在信息系统实现中会涉及，他们能更好地理解并应用它。由于在 Stony Brook 使用本教材的课程不是软件工程课程，因此我们上课时不涉及这些内容。事实上，我们要求学生课后阅读并在他们的课程项目中很好地利用软件工程实践。我们在课堂上会介绍“学生注册系统”的特点，以说明关于数据库和事务处理的重要问题。

## 第二版中的变化

数据库和事务处理系统的基本技术变化如此之快,使我们对本书第一版的内容进行了大量的更改和增补。一种快速发展的技术就是统一建模语言(UML)。在第4章“数据库设计”原有的E-R图的基础上,增加了大量关于UML的内容。同时还在第2章、第14章和第15章关于软件工程的内容中添加了UML的内容。

由于在实际应用中,大量投资被用在日益增加的数据库和事务处理应用程序的吞吐量方面,我们增加了第12章关于“数据库优化”的内容。

除此之外,几乎所有章节都进行了增补和更新。一个重要的例子就是融入了SQL/XML和RAID技术。

本书虽然没有包括Web业务,但它仍是一个很重要的内容。因为这是一个快速发展的、面向应用的有意义的技术,我们对本书的完全版做了大量修改以加入这方面的内容。除了通过更新关于XML和Web数据的章节并加入关于SQL/XML的小节来加强本书在XML技术方面的内容之外,我们也加入了新的关于Web业务的章节,包括SOAP、WSDL、BPEL、UDDI和利用WS-Coordination与WS-Transaction处理基于XML的事务等方面的内容。在“安全与网络商务”这一章,加入了基于XML的加密、利用XML加密、XML签名、WS安全和SAML等方面的内容。而在“事务处理系统体系结构”这一章,加入了关于网络应用程序服务器和J2EE方面的内容,它们被用来实现对许多Web业务的支持。

## 本书的组织

第1章~第7章应该以它们在本书中出现的顺序来进行讲授。第8章包含了许多信息,学生需要利用这些信息将前几章所学到的知识运用到实践中。但是,后续的章节并不是很依赖于第8章。第3部分中的第9章~第12章也应当按照顺序进行讲授。第4部分中软件工程方面的章节利用了第2部分和第3部分中的章节内容,但是关于软件工程的章节可以和数据库方面的内容并行讲授。本书高级主题中的第16章和第17章依赖于第2部分中的前7章。

最后,需要指出的是,书中打了“\*”的小节是可选的,因此如果教师愿意,可以将其省略。目录中前面有C<sub>s</sub>图标的小节是关于案例研究的。另外,带有星号的习题会比其他题目难一些,而带有两个星号的题目更难。

## 补充材料

除了本教材以外,下列补充材料可以协助教师教学:

- 所有章节的在线PowerPoint讲稿。
- 所有图表的在线PowerPoint。
- 一份在线的习题解答手册,包括所有练习的答案。
- 附加的参考文献、笔记、勘误表,作业和测试题。

请访问本书原版的伴随网站([www.aw-bc.com/kifer](http://www.aw-bc.com/kifer))来取得关于如何获取这些补充材料的详细信息。答案手册和PowerPoint讲稿只能通过Addison-Wesley公司的销售代表向教师提供。请访问[www.aw.com](http://www.aw.com)来联系你所在区域的销售代表。

## 致谢

我们要感谢下列审阅者,他们的意见和建议极大地促进了本书第二版的修改:

Tran Cao Son, New Mexico State University

Frantisek Franek, McMaster University

Junping Sun, Nova Southeastern University

Philip Cannata, Sun Microsystems

Dehu Qi, Lamar University

Nematollah Shiri, Concordia University

Jian Pei, State University of New York at Buffalo

Jack Wileden, University of Massachusetts Amherst

Sibel Adali, Rensselaer Polytechnic Institute

Roger King, University of Colorado at Boulder

Markus Schneider, University of Florida

Yaron Y. Goland, BEA

Dennis Shasha, New York University

Christelle Scharff, Pace University

Zhiwei Wang, Graduate Programs in Software Engineering, IT, and IS University of St. Thomas

我们还要感谢本书第一版的审阅者:

Suad Alagic, Wichita University

Catriel Beeri, The Hebrew University

Rick Cattell, Sun Microsystems

Jan Chomicki, SUNY Buffalo

Henry A. Etlinger, Rochester Institute of Technology

Leonidas Fegaras, University of Texas at Arlington

Alan Fekete, University of Sidney

Johannes Gehrke, Cornell University

Hershel Gottesman, Consultant

Jiawei Han, Simon Fraser University

Peter Honeyman, University of Michigan

Vijay Kumar, University of Missouri-Kansas City

Jonathan Lazar, Towson University

Dennis McLeod, University of Southern California

Rokia Missaoui, University of Quebec in Montreal

Clifford Neuman, University of Southern California

Fabian Pascal, Consultant

Sudha Ram, University of Arizona

Krithi Ramamirtham, University of Massachusetts-Amherst, and IIT Bombay

**Andreas Reuter, International University in Germany, Bruchsal**

**Arijit Sengupta, Georgia State University**

**Munindar P. Singh, North Carolina State University**

**Greg Speegle, Baylor University**

**Junping Sun, Nova Southeastern University**

**Joe Trubisz, Consultant**

**Vassilis J. Tsotras, University of California, Riverside**

**Emilia E. Villarreal, California Polytechnic State University**

同时还要感谢以下人员,他们对我们的问题提供了补充信息和解答:**Don Chamberlin、Daniela Florescu、Jim Gray、Pankaj Gupta、Rob Kelly 和 C. Mohan。**

下列两位使用本书的测试版进行教学,并且提出了宝贵的意见和建议:**David S. Warren 和 Radu Grosu。** Joe Trubiez 不仅是手稿完成后的审阅者,同时对本书许多章节的早期版本提出了宝贵的意见。

许多学生对本书各部分的阅读和勘误提供了很大的帮助:**Ziyang Duan、Shiyong Lu、Swapnil Patil、Guizhen Yang 和 Yan Zhang。**

非常感谢 Stony Brook 计算机科学系的教职员,尤其是 Kathy Germana,她的工作使得一切都成为可能。

我们要特别感谢 Matt Goldstein 和 Maite Suarez-Rivas——Addison-Wesley 出版公司的编辑,他们在本书早期内容和方法的策划以及全书的编写过程中都起着非常重要的作用。我们也要感谢 Addison-Wesley 公司和 Windfall 软件公司的员工,感谢他们在本书的编辑和生产过程中的出色工作:**Jeffrey Holcomb、Paul Anagnostopoulos、Elisabeth Beller、Jennifer McClain 和 Joe Snowden。**

最后,我们需要感谢的是我们的妻子 Lora、Edie 和 Rhoda,感谢她们在我们编写本书的过程中所给予的大力支持和鼓励。

**Michael Kifer**

**Arthur Bernstein**

**Philip M. Lewis**

# Preface

---

We are publishing the second edition of our textbook in two versions:

- This version, which consists of introductory material, is appropriate for a first undergraduate or graduate course in databases.
- The second version, which is the complete book, is appropriate for three courses:
  - An introductory undergraduate or graduate course in databases
  - An undergraduate or graduate course in transaction processing for students who have had an introductory course in databases
  - An advanced undergraduate or a first graduate course in databases for students who have had an introductory course in databases

One of our goals was to reduce the size and make this introductory version more affordable to students. Another was to capitalize on our experience in using the first edition of the book to make an even better introductory text.

The chapters in this book are not just a subset of those in the complete book. We believe that instructors of an introductory database course should have the option of enriching an introductory course by including material on object databases and XML—topics that are covered in great detail in several chapters in the complete book. Therefore we have added to the introductory book two new chapters, Chapter 16, Introduction to Object Databases, and Chapter 17, Introduction to XML and Web Data, which contain an appropriately chosen subset of the material in the full version of this book.

To keep the book up-to-date with the rapidly changing technology, we have added a substantial amount of material on UML to a number of chapters and have included a new chapter on Database Tuning, Chapter 12, in both the introductory and complete books.

As with the first edition, our focus is on how to build applications using databases rather than on how to build the database management system itself. We believe that many more students will be implementing applications than will be building DBMSs. Thus, we include substantial material describing the languages and APIs used by transactions to access a database, such as embedded SQL, ODBC, and JDBC.

Although we cover many practical aspects of database and transaction processing applications, we are primarily concerned with the concepts that underlie these topics rather than with the details of particular commercial systems or applications.

Thus we concentrate on the concepts behind the relational and object data models. These concepts will remain the foundation of database processing long after SQL is obsolete.

To enhance students' understanding of the technical material, we have included a case study of a transaction processing application, the Student Registration System, which is carried through the book. While a student registration system can hardly be considered glamorous, it has the unique advantage that all students have interacted with such a system as users. More importantly, it turns out to be a surprisingly rich application, so we can use it to illustrate many of the issues in database design, query processing, and transaction processing.

A unique aspect of the book is a presentation of the software engineering concepts required to implement transaction processing applications, using the Student Registration System as an example. Since the implementations of many information systems fail because of poor project management and inadequate software engineering, we feel that these topics should be an important part of the student's education. Our treatment of software engineering issues is brief, since many students will take a separate course in this subject. However, we believe that they will be better able to understand and apply that material when they see it presented in the context of an information system implementation. Since the courses that use this text at Stony Brook are not software engineering courses, we do not cover this material in class. Instead, we ask the students to read it and require that they use good software engineering practice in their class projects. We do cover in class those aspects of the Student Registration System that illustrate important issues in databases and transaction processing.

## Changes in the Second Edition

The technology underlying database and transaction processing systems is changing so rapidly that we have made a large number of changes and additions to the material of the first edition. One rapidly advancing technology is the Unified Modeling Language, UML. We added substantial amount of material on UML in Chapter 4 on database design, in addition to the material on E-R diagrams that was already there. We also added UML to the material on software engineering in Chapters 2, 14, and 15.

A new chapter on Database Tuning, Chapter 12, was added because so much effort in the real world is spent increasing the throughput of database and transaction processing applications.

In addition, material has been added and updated in almost all the chapters. Significant examples of this are the coverage of SQL/XML and RAID technology.

One important area that is *not* included in this volume is Web Services. Since this is a rapidly developing and interesting application-oriented subject we have significantly revised the compete version of this text to include material on this topic. In addition to strengthening the book on the subject of XML Technology by updating the chapter on XML and Web Data and adding a section on SQL/XML, we have added a new chapter on Web Services that contains material on SOAP,

WSDL, BPEL, UDDI, and XML-based transaction processing using WS-Coordination and WS-Transaction. In the chapter on Security and Internet Commerce, we added a section on XML-based encryption, using XML-Encryption, XML-Signature, WS-Security, and SAML. And in the chapter on Architecture of Transaction Processing Systems, we added material on Web Application Servers and J2EE, which are used to implement the back-end of many Web services.

## Organization of the Book

Chapters 1 through 7 should be taught in the order in which they appear in the book. Chapter 8 contains much of the information that students need in order to put the knowledge they acquired in the preceding chapters into practice. However, subsequent chapters do not significantly depend on Chapter 8. Chapters 9 through 12 in Part 3 should be taught sequentially. Chapter 13 in the same part is largely independent. The software engineering chapters in Part 4 utilize the material of the chapters in Parts 2 and 3, but the software engineering chapters can be read in parallel with the database material. Chapters 16 and 17 in the advanced part of the book depend on the first seven chapters in Part 2.

Finally we note that the sections in this book that are marked with an asterisk (\*) are optional and can be omitted, if the instructor prefers to do so. Sections marked with the  icon in the table of contents deal with the case study. Also, exercises that are marked with an asterisk are slightly harder than the rest, and exercises that are marked with two asterisks are even harder.

## Supplements

In addition to the text, the following supplementary materials are available to assist instructors:

- Online PowerPoint presentations for all chapters
- Online PowerPoint slides of all figures
- An online solution manual containing solutions for the exercises
- Additional references, notes, errata, homeworks, and exams.

For more information on obtaining these supplements, please visit this book's Companion Website at [www.aw-bc.com/kifer](http://www.aw-bc.com/kifer). The solutions manual and PowerPoint presentations are available only to instructors through your Addison-Wesley sales representative. To contact your representative, please visit [www.aw.com](http://www.aw.com).

## Acknowledgments

We would like to thank the reviewers, whose comments and suggestions significantly improved the second edition of the book:

Tran Cao Son, New Mexico State University

Frantisek Franek, McMaster University

**Junping Sun, Nova Southeastern University**  
**Philip Cannata, Sun Microsystems**  
**Dehu Qi, Lamar University**  
**Nematollah Shiri, Concordia University**  
**Jian Pei, State University of New York at Buffalo**  
**Jack Wileden, University of Massachusetts Amherst**  
**Sibel Adali, Rensselaer Polytechnic Institute**  
**Roger King, University of Colorado at Boulder**  
**Markus Schneider, University of Florida**  
**Yaron Y. Goland, BEA**  
**Dennis Shasha, New York University**  
**Christelle Scharff, Pace University**  
**Zhiwei Wang, Graduate Programs in Software Engineering, IT, and IS University of St. Thomas.**

We would also like to thank the reviewers of the first edition of the book:

**Suad Alagic, Wichita University**  
**Catriel Beeri, The Hebrew University**  
**Rick Cattel, Sun Microsystems**  
**Jan Chomicki, SUNY Buffalo**  
**Henry A. Etlinger, Rochester Institute of Technology**  
**Leonidas Fegaras, University of Texas at Arlington**  
**Alan Fekete, University of Sidney**  
**Johannes Gehrke, Cornell University**  
**Hershel Gottesman, Consultant**  
**Jiawei Han, Simon Fraser University**  
**Peter Honeyman, University of Michigan**  
**Vijay Kumar, University of Missouri-Kansas City**  
**Jonathan Lazar, Towson University**  
**Dennis McLeod, University of Southern California**  
**Rokia Missaoui, University of Quebec in Montreal**  
**Clifford Neuman, University of Southern California**  
**Fabian Pascal, Consultant**  
**Sudha Ram, University of Arizona**  
**Krithi Ramamritham, University of Massachusetts-Amherst, and IIT Bombay**  
**Andreas Reuter, International University in Germany, Bruchsal**  
**Arijit Sengupta, Georgia State University**

Munindar P. Singh, North Carolina State University

Greg Speegle, Baylor University

Junping Sun, Nova Southeastern University

Joe Trubisz, Consultant

Vassilis J. Tsotras, University of California, Riverside

Emilia E. Villarreal, California Polytechnic State University

We would also like to thank the following people who were kind enough to provide us with additional information and answers to our questions: Don Chamberlin, Daniela Florescu, Jim Gray, Pankaj Gupta, Rob Kelly, and C. Mohan.

Two people taught out of beta versions of the book and made useful comments and suggestions: David S. Warren and Radu Grosu. Joe Trubicz served not only as a reviewer when the manuscript was complete, but provided critical comments on early versions of many of the chapters.

A number of students were very helpful in reading and checking the correctness of various parts of the book: Ziyang Duan, Shiyong Lu, Swapnil Patil, Guizhen Yang, and Yan Zhang.

Many thanks to the staff of the Computer Science Department at Stony Brook, and in particular Kathy Germana, who helped make things happen at work.

We would particularly like to thank Matt Goldstein and Maite Suarez-Rivas, our editors at Addison-Wesley, who played an important role in shaping the contents and approach of the book in its early stages and throughout the time we were writing it. We would also like to thank the various staff members of Addison-Wesley and Windfall Software, who did an excellent job of editing and producing the book: Jeffrey Holcomb, Paul Anagnostopoulos, Elisabeth Beller, Jennifer McClain, and Joe Snowden.

Last, but not least, we would like to thank our wives, Lora, Edie, and Rhoda, who provided much needed support and encouragement while we were writing the book.

# **Contents**

---

<b>Preface</b>	<b>xvii</b>
<b>PART ONE Introduction</b>	<b>1</b>
<b>1 Overview of Databases and Transactions</b>	<b>3</b>
1.1 What Are Databases and Transactions? 3	
1.2 Features of Modern Database and Transaction Processing Systems 6	
1.3 Major Players in the Implementation and Support of Database and Transaction Processing Systems 7	
1.4 Decision Support Systems—OLAP and OLTP 9	
<b>2 The Big Picture</b>	<b>13</b>
2.1 Case Study: A Student Registration System 13	
2.2 Introduction to Relational Databases 14	
2.3 What Makes a Program a Transaction—The ACID Properties 20 Bibliographic Notes 25 Exercises 25	
<b>PART TWO Database Management</b>	<b>29</b>
<b>3 The Relational Data Model</b>	<b>31</b>
3.1 What Is a Data Model? 31	
3.2 The Relational Model 35	
3.2.1 Basic Concepts 35	
3.2.2 Integrity Constraints 38	
3.3 SQL—Data Definition Sublanguage 46	
3.3.1 Specifying the Relation Type 46	
3.3.2 The System Catalog 46	
3.3.3 Key Constraints 47	
3.3.4 Dealing with Missing Information 48	

3.3.5 Semantic Constraints	49
3.3.6 User-Defined Domains	53
3.3.7 Foreign-Key Constraints	53
3.3.8 Reactive Constraints	56
3.3.9 Database Views	59
3.3.10 Modifying Existing Definitions	60
3.3.11 SQL-Schemas	62
3.3.12 Access Control	63
Bibliographic Notes	65
Exercises	66
<b>4 Conceptual Modeling of Databases with Entity-Relationship Diagrams and the Unified Modeling Language</b>	<b>69</b>
4.1 Conceptual Modeling with the E-R Approach	70
4.2 Entities and Entity Types	70
4.3 Relationships and Relationship Types	73
4.4 Advanced Features in Conceptual Data Modeling	78
4.4.1 Entity Type Hierarchies	78
4.4.2 Participation Constraints	81
4.4.3 The Part-of Relationship	83
4.5 From E-R Diagrams to Relational Database Schemas	86
4.5.1 Representation of Entities	86
4.5.2 Representation of Relationships	88
4.5.3 Representing IsA Hierarchies in the Relational Model	90
4.5.4 Representation of Participation Constraints	92
4.5.5 Representation of the Part-of Relationship	94
4.6 UML: A New Kid on the Block*	95
4.6.1 Representing Entities in UML	96
4.6.2 Representing Relationships in UML	97
4.6.3 Advanced Modeling Concepts in UML	101
4.6.4 Translation to SQL	105
4.7 A Brokerage Firm Example	106
4.7.1 An Entity-Relationship Design	106
4.7.2 A UML Design*	110
⑧ 4.8 Case Study: A Database Design for the Student Registration System	111
4.8.1 The Database Part of the Requirements Document	112
4.8.2 The Database Design	113
4.9 Limitations of Data Modeling Methodologies	119
Bibliographic Notes	123
Exercises	123

<b>5 Relational Algebra and SQL</b>	<b>127</b>
<b>5.1 Relational Algebra: Under the Hood of SQL</b>	<b>128</b>
<b>5.1.1 Basic Operators</b>	128
<b>5.1.2 Derived Operators</b>	137
<b>5.2 The Query Sublanguage of SQL</b>	<b>147</b>
<b>5.2.1 Simple SQL Queries</b>	148
<b>5.2.2 Set Operations</b>	154
<b>5.2.3 Nested Queries</b>	157
<b>5.2.4 Quantified Predicates</b>	163
<b>5.2.5 Aggregation over Data</b>	164
<b>5.2.6 Join Expressions in the FROM Clause</b>	170
<b>5.2.7 A Simple Query Evaluation Algorithm</b>	171
<b>5.2.8 More on Views in SQL</b>	174
<b>5.2.9 Materialized Views</b>	177
<b>5.2.10 The Null Value Quandary</b>	181
<b>5.3 Modifying Relation Instances in SQL</b>	<b>182</b>
<b>5.3.1 Inserting Data</b>	182
<b>5.3.2 Deleting Data</b>	184
<b>5.3.3 Updating Existing Data</b>	185
<b>5.3.4 Updates on Views</b>	185
Bibliographic Notes	187
Exercises	188
<b>6 Database Design with the Relational Normalization Theory</b>	<b>193</b>
<b>6.1 The Problem of Redundancy</b>	193
<b>6.2 Decompositions</b>	195
<b>6.3 Functional Dependencies</b>	198
<b>6.4 Properties of Functional Dependencies</b>	200
<b>6.5 Normal Forms</b>	207
<b>6.5.1 The Boyce-Codd Normal Form</b>	208
<b>6.5.2 The Third Normal Form</b>	210
<b>6.6 Properties of Decompositions</b>	211
<b>6.6.1 Lossless and Lossy Decompositions</b>	212
<b>6.6.2 Dependency-Preserving Decompositions</b>	215
<b>6.7 An Algorithm for BCNF Decomposition</b>	219
<b>6.8 Synthesis of 3NF Schemas</b>	221
<b>6.8.1 Minimal Cover</b>	222
<b>6.8.2 3NF Decomposition through Schema Synthesis</b>	224
<b>6.8.3 BCNF Decomposition through 3NF Synthesis</b>	226
<b>6.9 The Fourth Normal Form</b>	228

<b>6.10</b>	<b>Advanced 4NF Design*</b>	<b>233</b>
6.10.1	MVDs and Their Properties	234
6.10.2	The Difficulty of Designing for 4NF	235
6.10.3	A 4NF Decomposition How-To	238
<b>6.11</b>	<b>Summary of Normal Form Decomposition</b>	<b>240</b>
<b>6.12</b>	<b>Case Study: Schema Refinement for the Student Registration System</b>	<b>241</b>
<b>6.13</b>	<b>Tuning Issues: To Decompose or Not to Decompose?</b>	<b>244</b>
	Bibliographic Notes	245
	Exercises	246

<b>7</b>	<b>Triggers and Active Databases</b>	<b>251</b>
----------	--------------------------------------	------------

7.1	What Is a Trigger?	251
7.2	Semantic Issues in Trigger Handling	252
7.3	Triggers in SQL:1999	256
7.4	Avoiding a Chain Reaction	264
	Bibliographic Notes	265
	Exercises	265

<b>8</b>	<b>Using SQL in an Application</b>	<b>267</b>
----------	------------------------------------	------------

8.1	What Are the Issues Involved?	267
8.2	Embedded SQL	268
8.2.1	Status Processing	271
8.2.2	Sessions, Connections, and Transactions	273
8.2.3	Executing Transactions	274
8.2.4	Cursors	276
8.2.5	Stored Procedures on the Server	282
8.3	More on Integrity Constraints	285
8.4	Dynamic SQL	286
8.4.1	Statement Preparation in Dynamic SQL	287
8.4.2	Prepared Statements and the Descriptor Area*	290
8.4.3	Cursors	293
8.4.4	Stored Procedures on the Server	293
8.5	JDBC and SQLJ	294
8.5.1	JDBC Basics	294
8.5.2	Prepared Statements	297
8.5.3	Result Sets and Cursors	297
8.5.4	Obtaining Information about a Result Set	300
8.5.5	Status Processing	300

8.5.6 Executing Transactions	301
8.5.7 Stored Procedures on the Server	302
8.5.8 An Example	303
8.5.9 SQLJ: Statement-Level Interface to Java	303
<b>8.6 ODBC*</b>	<b>307</b>
8.6.1 Prepared Statements	309
8.6.2 Cursors	309
8.6.3 Status Processing	312
8.6.4 Executing Transactions	312
8.6.5 Stored Procedures on the Server	313
8.6.6 An Example	313
<b>8.7 Comparison</b>	<b>315</b>
Bibliographic Notes	316
Exercises	316
<b>PART THREE Optimizing DBMS Performance and Transaction Processing</b>	<b>319</b>
<b>9 Physical Data Organization and Indexing</b>	<b>321</b>
9.1 Disk Organization	322
9.1.1 RAID Systems	326
9.2 Heap Files	329
9.3 Sorted Files	333
9.4 Indices	337
9.4.1 Clustered versus Unclustered Indices	340
9.4.2 Sparse versus Dense Indices	342
9.4.3 Search Keys Containing Multiple Attributes	344
9.5 Multilevel Indexing	347
9.5.1 Index-Sequential Access	350
9.5.2 B <sup>+</sup> Trees	353
9.6 Hash Indexing	360
9.6.1 Static Hashing	360
9.6.2 Dynamic Hashing Algorithms	363
9.7 Special-Purpose Indices	371
9.7.1 Bitmap Indices	371
9.7.2 Join Indices	372
9.8 Tuning Issues: Choosing Indices for an Application	373
Bibliographic Notes	374
Exercises	375

<b>10 The Basics of Query Processing</b>	<b>379</b>
10.1 Overview of Query Processing	379
10.2 External Sorting	380
10.3 Computing Projection, Union, and Set Difference	384
10.4 Computing Selection	386
10.4.1 Selections with Simple Conditions	387
10.4.2 Access Paths	389
10.4.3 Selections with Complex Conditions	391
10.5 Computing Joins	392
10.5.1 Computing Joins Using Simple Nested Loops	393
10.5.2 Sort-Merge Join	396
10.5.3 Hash Join	398
10.6 Multirelational Joins*	399
10.7 Computing Aggregate Functions	401
Bibliographic Notes	401
Exercises	401
<b>11 An Overview of Query Optimization</b>	<b>405</b>
11.1 Query Processing Architecture	405
11.2 Heuristic Optimization Based on Algebraic Equivalences	407
11.3 Estimating the Cost of a Query Execution Plan	410
11.4 Estimating the Size of the Output	418
11.5 Choosing a Plan	420
Bibliographic Notes	425
Exercises	425
<b>12 Database Tuning</b>	<b>429</b>
12.1 Disk Caches	430
12.1.1 Tuning the Cache	431
12.2 Tuning the Schema	433
12.2.1 Indices	433
12.2.2 Denormalization	440
12.2.3 Repeating Groups	441
12.2.4 Partitioning	442
12.3 Tuning the Data Manipulation Language	443
12.4 Tools	446
12.5 Managing Physical Resources	447
12.6 Influencing the Optimizer	448
Bibliographic Notes	451

Exercises	451
<b>13 An Overview of Transaction Processing</b>	<b>455</b>
13.1 Isolation	455
13.1.1 Serializability	456
13.1.2 Two-Phase Locking	458
13.1.3 Deadlock	462
13.1.4 Locking in Relational Databases	463
13.1.5 Isolation Levels	465
13.1.6 Lock Granularity and Intention Locks	468
13.1.7 Summary	471
13.2 Atomicity and Durability	472
13.2.1 The Write-Ahead Log	472
13.2.2 Recovery from Mass Storage Failure	476
13.3 Implementing Distributed Transactions	477
13.3.1 Atomicity and Durability—The Two-Phase Commit Protocol	478
13.3.2 Global Serializability and Deadlock	480
13.3.3 Replication	482
13.3.4 Summary	484
Bibliographic Notes	484
Exercises	485
<b>PART FOUR Software Engineering Issues and Documentation</b>	<b>487</b>
<b>14 Requirements and Specifications</b>	<b>489</b>
14.1 Software Engineering Methodology	489
14.1.1 UML Use Cases	490
⑩ 14.2 The Requirements Document for the Student Registration System	493
⑩ 14.3 Requirements Analysis—New Issues	500
⑩ 14.4 Specifying the Student Registration System	502
14.4.1 UML Sequence Diagrams	503
⑩ 14.5 The Specification Document for the Student Registration System: Section III	504
14.6 The Next Step in the Software Engineering Process	506
Bibliographic Notes	506
Exercises	507

<b>15 Design, Coding, and Testing</b>	<b>509</b>
15.1 The Design Process	509
15.1.1 Database Design	510
15.1.2 Describing the Behavior of Objects with UML State Diagrams	510
15.1.3 Structure of the Design Document	512
15.1.4 Design Review	514
15.2 Test Plan	515
15.3 Project Planning	518
15.4 Coding	521
15.5 Incremental Development	523
15.6 The Project Management Plan	524
15.7 Design and Code for the Student Registration System	525
15.7.1 Completing the Database Design: Integrity Constraints	526
15.7.2 Design of the Registration Transaction	528
15.7.3 Partial Code for the Registration Transaction	530
Bibliographic Notes	533
Exercises	533
<b>PART FIVE Advanced Topics in Databases</b>	<b>535</b>
<b>16 Introduction to Object Databases</b>	<b>537</b>
16.1 Shortcomings of the Relational Data Model	537
16.2 The Conceptual Object Data Model	543
16.2.1 Objects and Values	544
16.2.2 Classes	545
16.2.3 Types	546
16.2.4 Object-Relational Databases	549
16.3 Objects in SQL:1999 and SQL:2003	550
16.3.1 Row Types	551
16.3.2 User-Defined Types	552
16.3.3 Objects	553
16.3.4 Querying User-Defined Types	554
16.3.5 Updating User-Defined Types	555
16.3.6 Reference Types	558
16.3.7 Inheritance	560
16.3.8 Collection Types	561
Bibliographic Notes	563
Exercises	564

<b>17 Introduction to XML and Web Data</b>	<b>567</b>
17.1 Semistructured Data	567
17.2 Overview of XML	570
17.2.1 XML Elements and Database Objects	573
17.2.2 XML Attributes	575
17.2.3 Namespaces	577
17.2.4 Document Type Definitions	582
17.2.5 Inadequacy of DTDs as a Data Definition Language	585
17.3 XML Schema	586
17.3.1 XML Schema and Namespaces	587
17.3.2 Simple Types	590
17.3.3 Complex Types	595
17.3.4 Putting It Together	603
17.3.5 Shortcuts: Anonymous Types and Element References	606
17.3.6 Integrity Constraints	608
17.4 XML Query Languages	615
17.4.1 XPath: A Lightweight XML Query Language	616
17.4.2 SQL/XML	623
Bibliographic Notes	633
Exercises	634
<b>Bibliography</b>	<b>639</b>
<b>Index</b>	<b>649</b>



## PART ONE

---

# Introduction

THE INTRODUCTORY PART of the book consists of two chapters.

In Chapter 1, we will try to get you excited about the fields of databases and transaction processing by giving you some idea of what the book is all about.

In Chapter 2, we will introduce many of the technical concepts underlying the fields of databases and transaction processing, including the SQL language and the ACID properties of transactions. We will expand on these concepts in the rest of the book.



# 1

## Overview of Databases and Transactions

### 1.1 What Are Databases and Transactions?

During your vacation, you stand at the checkout counter of a department store in Tokyo, hand the clerk your credit card, and wait anxiously for your purchases to be approved. In the few seconds you have to wait, messages are sent around the world to one or more banks and clearinghouses, accessing and updating a number of databases until finally the system approves your purchase. Over 100 million such credit card transactions are processed each day from over 10 million merchants through more than 20 thousand banks. Billions of dollars are involved, and the only record of what happens is stored in the databases on the network. The accuracy, security, and availability of these databases and the correctness and performance characteristics of the transactions that access them are critical to the entire credit card business.

**What is a database?** A **database** is a collection of data items related to some enterprise—for example, the depositor account information in a bank. A database might be stored on cards in a Rolodex or on paper in a file cabinet, but we are particularly interested in databases stored as bits and bytes in a computer. Such a database can be **centralized** on one computer or **distributed** over several, perhaps widely separated geographically.

An increasing number of enterprises depend on such databases for their very existence. No paper records exist within the enterprise; the only up-to-date record of its current status—for example, the balance of each bank customer's checking account—is stored in its databases. Many enterprises view their databases as their most important asset.

For example, the database of the company that manufactured the airplane on which you flew to Tokyo contains the only record of information about the engineering design, manufacturing processes, and subassembly suppliers involved in producing that plane 10 years ago, together with every test made on it over its lifetime. If, at some time in the future, a test shows that a turbine blade on one of the plane's jet engines has failed, the company can determine from its database which subcontractor supplied that particular engine, and the subcontractor can determine from its database the date on which that turbine blade was manufactured,

the machines and people involved, the source of the materials from which the blade was fabricated, and the results of quality assurance tests made while the blade was being manufactured. In this way it can determine the cause of the failure and increase the quality of future planes. The existence of these detailed historical databases, as well as the ability to search them for information about the fabrication of a specific turbine blade in a specific jet engine on a specific airplane manufactured 10 years ago, gives the airplane manufacturer a significant strategic advantage over any other manufacturer that does not maintain such databases.

In some cases, a database is the major asset of an enterprise—for example, the database of the credit history company that your credit card company consulted when you applied for your card. In other cases, the accuracy of the information in the database is critical for human life—for example, the database in the air traffic control system at the Tokyo airport.

**What is a database management system?** To make access to them convenient, databases are generally encapsulated within a **database management system (DBMS)**. The DBMS supports a high-level language in which the application programmer describes the database access it wishes to perform. Typically, all database access is classified into two broad categories: **queries** and **updates**. A query is a request to retrieve data, and an update is a request to insert, delete, or modify existing data items. The most commonly used data access language, and the one we study the most in this text, is the Structured Query Language (SQL). Although it is called a *query* language, updates are also done through SQL. The beauty of SQL lies in its declarative nature: the application programmer need only state what is to be done; the DBMS figures out how to do it efficiently. The DBMS interprets each SQL statement and performs the action it describes. The application programmer need not know the details of how the database is stored, need not formulate the algorithm for performing the access, and need not be concerned about many other aspects of managing the database. Compare this to the regular file systems where the programmer not only has to know the details of the file structure but also provide the algorithms to search the files to retrieve the desired information.

**What is a transaction?** Databases frequently store information that describes the current state of an enterprise. For example, a bank's database stores the current balance in each depositor's account. When an event happens in the real world that changes the state of the enterprise, a corresponding change must be made to the information stored in the database. With online DBMSs, these changes are made in real time by programs called **transactions**, which execute when the real-world event occurs. For example, when a customer deposits money in a bank (an event in the real world), a deposit transaction is executed. Each transaction must be designed so that it maintains the correctness of the relationship between the database state and the real-world enterprise it is modeling. In addition to changing the state of the database, the transaction itself might initiate some events in the real world. For example, a withdraw transaction at an automated teller machine (ATM) initiates the event of dispensing cash, and a transaction that establishes a connection for a

telephone call requires the allocation of resources (bandwidth on a long-distance link) in the telephone company's infrastructure.

Credit card approval is only one example of a transaction that you executed on your vacation in Tokyo. Your flight arrangements involved a transaction with the airline's reservation database, your passage through passport control at the airport involved a transaction with the immigration services database, and your check-in at the hotel involved a transaction with the hotel reservation database. Even the phone call you made from your hotel room to tell your family you had arrived safely involved transactions with the hotel billing database and with a long-distance carrier to arrange billing and to establish the call.

Other examples of transactions you probably execute regularly involve ATM systems, supermarket scanning systems, and university registration and billing systems. Increasingly, these transactions entail access to **distributed databases**: multiple databases managed by different DBMSs stored at different geographical locations. Your phone call transaction at the Tokyo hotel is an example.

**What is a transaction processing system?** A **transaction processing system** (TPS) includes one or more databases that store the state of an enterprise, the software for managing the transactions that manipulate that state, and the transactions themselves that constitute the application code. In its simplest form the TPS involves a single DBMS that contains the software for managing transactions. More complex systems involve several DBMSs. In this case, transaction management is handled both within the DBMSs and without, by additional code called a **TP monitor** that coordinates transactions across multiple sites (see Figure 1.1).

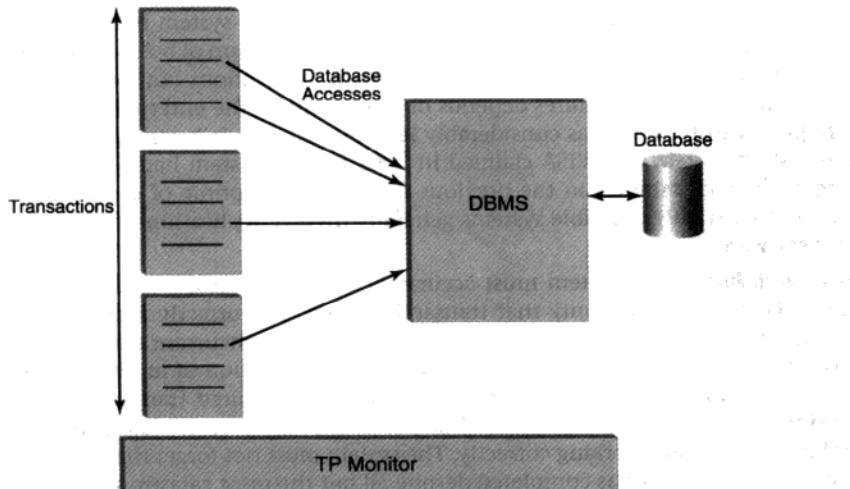


FIGURE 1.1 The structure of a transaction processing system.

The database is at the heart of a transaction processing system because it persists beyond the lifetime of any particular transaction. An increasing number of enterprises depend on such systems for their business. For example, one might say that the credit card transaction processing system is the credit card business.

Our concern in this book is with the technical aspects of databases and the transaction processing systems that use them. Specifically, we are interested in the design and implementation of applications, including the organization of the application database, but we are not concerned with the algorithms and data structures used to implement the underlying DBMS and transaction processing system modules. Nevertheless, we must learn enough about these underlying systems so that we can use them intelligently in an application.

## 1.2 Features of Modern Database and Transaction Processing Systems

Modern computer and communication technology has led to significant advances in the architecture, design, and use of database and transaction processing systems. Their enhanced functionality has lead to important new business opportunities for the enterprises that deploy them and, in turn, implies a number of additional requirements on their operation:

- **High availability.** Because the system is online, it must be operational at all times when the enterprise is open for business. In some enterprises, this means that the system must always be available. For example, an airline reservation system might be required to accept requests for flight reservations from ticket offices spread over a large number of time zones, so the system is never shut down. With online systems, failures can result in a disruption of business—if the computer in an airline reservation system is down, reservations cannot be made. The ability to tolerate failures depends on the nature of the enterprise. Clearly a flight control system has considerably less tolerance for failures than a flight reservation system has. VISA claimed in 2002 that its system had been down a total of eight minutes in the previous five years (an uptime of greater than 99.9999%). Highly available systems generally involve replication of hardware and software.
- **High reliability.** The system must accurately reflect the results of all transactions. This implies not only that transactions must be correctly programmed but also that errors must not be introduced because of concurrent execution of (correctly programmed) transactions or intercommunication of modules while the transaction is executing. Furthermore, large, distributed transaction processing systems include thousands of hardware and software modules, and it is unlikely that all are working correctly. The system must not forget the results of any transaction that has completed despite all but the most catastrophic forms of failure. For example, the database in a banking system must accurately reflect

the effect of all the deposits and withdrawals that have completed and cannot lose the results of any such transactions should it subsequently crash.

- **High throughput.** Because the enterprise has many customers who must use the transaction processing system, the system must be capable of performing many transactions per second. For example, a credit card approval system might perform thousands of transactions per second during its busiest periods. As we shall see, this requirement implies that individual transactions cannot be executed sequentially but must be executed concurrently—thus significantly complicating the design of the system.
- **Low response time.** Because customers might be waiting for a response from it, the system must respond quickly. Response requirements may differ depending on the application. Whereas you might be willing to wait fifteen seconds for an ATM to output cash, you expect a telephone connection to be made in no more than one or two seconds. Furthermore, in some applications, if the response does not occur within a fixed period of time, the transaction will not perform properly. For example, in a factory automation system the transaction might be required to actuate a device before some unit passes a particular position on the conveyor belt. Applications of this type are said to have **hard real-time** constraints.
- **Long lifetime.** Transaction processing systems are complex and not easily replaced. They must be designed in such a way that individual hardware or software modules can be replaced with newer versions (that perform better or have additional functionality) without necessitating major changes to the surrounding system.
- **Security.** Many transaction processing systems contain information about the private concerns of individuals (e.g., the items they purchase, their credit card number, the videos they view, and their health and financial records). Because these systems can be accessed by a large number of people from a large number of places (perhaps over the Internet), security is important. Individual users must be authenticated (are they who they claim to be?), users must be allowed to execute only those transactions they are authorized to execute (only a bank teller can execute a transaction to generate a certified check), the information in the database must not be corrupted or read by an attacker, and the information transmitted between the user and the system must not be altered or overheard by an eavesdropper.

### 1.3 Major Players in the Implementation and Support of Database and Transaction Processing Systems

A transaction processing system, together with its associated databases, can be an immensely complex assemblage of hardware and software, with which many different types of people interact in various roles. Examining these roles is a useful

way of understanding what a transaction processing system is. First consider the people involved in the design and implementation of a transaction processing system:

- **System analyst.** The system analyst works with the customer of a proposed application system to develop formal requirements and specifications for it. He or she must understand both the business rules of the enterprise for which the application is being implemented and the database and transaction processing technology underlying the implementation so that the application will meet the customer's needs and execute efficiently. The specifications developed by the system analyst are then refined into the design of the database formats and the individual transactions that will access the database.
- **Database designer.** The database designer specifies the structure of the database appropriate for an application. The database contains the information that describes the current state of the real-world application. The structure must support the accesses required by the transactions and allow those accesses to be performed in a timely manner.
- **Application programmer.** The application programmer implements the graphical user interface and the individual transactions in the system. He or she must ensure that the transactions maintain the correspondence between the state of the real-world application and the state of the database. Together with the database designer, the application programmer must ensure that the rules governing the workings of the enterprise are enforced. For example, in the Student Registration System, to be discussed in Section 2.1, the number of students enrolled in a course should not exceed the number of seats in the room assigned to the course.
- **Project manager.** The project manager is responsible for the successful completion of the implementation project. He or she prepares schedules and budgets, assigns people to tasks, and monitors day-to-day project operation. Project management is surprisingly difficult. According to a widely quoted report of the Standish Group, an Information Technology (IT) consulting group, of the more than eight thousand IT projects the group surveyed, only 16% completed successfully—on time and on budget [Standish 2000]. The primary reason for the failures was almost always poor project management.<sup>1</sup>

The people interacting with (as opposed to building) an operational transaction processing system include the following:

- **User.** The user causes the execution of individual transactions, usually by interacting through some graphical user interface. The user interface must be

<sup>1</sup> For large companies, the success rate dropped to 9%. For projects that completed late or over budget, the average completion time was 222% of the scheduled time and the average cost was 189% of the budgeted cost. An astonishing 31% of the projects were canceled before they were completed. At the time this book was written, information about this study, called Chaos, could be found in [Standish 2000].

appropriate to the capabilities of the intended class of users. As an example, the user interface presented by an ATM is simple enough that an average person can use the system to perform bank deposit and withdraw transactions without any training or instructions except those presented on the screen. By contrast, the interface to an airline reservation system, which is used by reservation clerks or travel agents, requires advanced training. In both cases, however, most of the complexities of the system are hidden from the user.

- **Database administrator.** The database administrator is responsible for supporting the database while the system is running. Among his or her concerns are allocating storage space for the database, monitoring and optimizing database performance, and monitoring and controlling database security. In addition, the database administrator might modify the structure of the database to accommodate changes in the enterprise or to handle performance bottlenecks.
- **System administrator.** The system administrator is responsible for supporting the system as a whole while it is running. Among the things he or she must keep track of are
  - **System architecture.** What hardware and software modules are connected to the system at any instant, and how are they interconnected?
  - **Configuration management.** What version of each software module exists on each machine?
  - **System status.** What is the health of the system? Which systems and communication links are operational or congested, and what is being done to repair the situation? How is the system currently performing?

Our main interest in this book lies at the application level. Thus, we are particularly concerned with the roles of the system analyst, the application programmer, and the database designer. However, in order for someone working at the application level to take full advantage of the capabilities of the underlying system, he or she must be knowledgeable about the other roles as well.

## 1.4 Decision Support Systems—OLAP and OLTP

Transaction processing is not the only application domain in which databases play a key role. Another such domain is **decision support**. While transaction processing is concerned with using a database to maintain an accurate model of some real-world situation, decision support is concerned with using the information in a database to guide management decisions. To illustrate the differences between these two domains, we discuss the roles they might play in the operation of a national supermarket chain.

**Transaction processing.** Each local supermarket in a chain maintains a database of the prices and current inventory of all the items it sells. It uses that database (together with a bar code scanner) as part of a transaction processing system at the checkout counters. One transaction in this system might be, "Three cans of Campbell soup

and one box of Ritz crackers were purchased; compute the price, print out a receipt, update the balance in the cash drawer, and subtract these items from the store's inventory." The customer expects this transaction to complete in a few seconds.

The main goal of such a transaction processing system is to maintain the correspondence between the database and the real-world situation it is modeling as events occur in the real world. In this case, the event is the customer's purchase, and the real-world situation is the store's inventory and the amount of cash in the cash drawer.

**Decision support.** The managers of the supermarket chain might want to analyze the data stored in the databases in each store to help them make decisions for the chain as a whole. Such decision support applications are becoming increasingly important as enterprises attempt to turn the *data* in their databases into *information* they can use to advance their long-term strategic goals.

Decision support applications involve queries to one or more databases, possibly followed by some mathematical analysis of the information returned by the queries. Decision support applications are sometimes called **online analytic processing (OLAP)**, in contrast with the **online transaction processing (OLTP)** applications we have been discussing.

In some decision support applications, the queries are so simple they can be implemented as transactions in the same local database used for OLTP applications—for example, "Print out a report of the weekly produce sales in Store 27 for the past six months."

In many applications, however, the queries are quite complex and cannot be efficiently executed against the local databases. They take too long to execute (because the database has been optimized for OLTP transactions) and cause the local transactions—for example, the checkout transactions—to execute too slowly. The supermarket chain therefore maintains a separate database specifically for such complex OLAP queries. The database contains historical information about sales and inventory from all its branches for the past 10 years. This information is extracted from the individual store databases at various times and updated once a day. Such a database is called a **data warehouse**.

A manager can enter a complex query about the data in the data warehouse—for example, "During the winter months of the last five years, what is the percentage of customers in northeast urban supermarkets who bought crackers at the same time they bought soup?" (Perhaps these items should be placed near each other on the shelves.)

Data warehouses can contain terabytes ( $10^{12}$  bytes) of data and require special hardware to maintain that data. An OLAP query might be quite difficult to formulate and might require query language concepts more powerful than those needed for OLTP queries. OLAP queries usually do not have severe constraints on execution time and might take several hours to execute. The warehouse database might have been structured to speed up the execution of such queries. The database need be updated only periodically because minute-by-minute correctness is not needed for

the types of queries it supports—satisfactory responses might be obtained even if the database is less than 100% accurate.

**Data mining.** A manager might also be interested in making a much less structured query about the data in the warehouse database—for example, “Are there *any* interesting combinations of items bought by customers?” Such queries are called **data mining**. In contrast with OLAP, in which requests are made to obtain specific information, data mining can be viewed as knowledge discovery—an attempt to extract new knowledge from the data stored in the database.

Data mining queries can be extremely difficult to formulate and might require sophisticated mathematics or techniques from the field of artificial intelligence. A query might require many hours to execute and might involve several interactions with the manager for obtaining additional information or reformulating parts of the query.

One widely repeated but perhaps apocryphal success story of data mining is that a convenience store chain used the above query (“Are there *any* interesting combinations . . . ”) and found an unexpected correlation. In the early evenings, a high percentage of male customers who bought diapers also bought beer—presumably these customers were fathers who were going to stay home that night with their babies.



# 2

## The Big Picture

### 2.1 Case Study: A Student Registration System

Your university is interested in implementing a student registration system so that students can register for courses from their home PCs. You have been asked to build a prototype of that system as a project in this course. The registrar has prepared the following preliminary **Statement of Objectives** for the system.

The objectives of the Student Registration System are to allow students and faculty (as appropriate) to

1. Authenticate themselves as users of the system
2. Register and deregister for courses (offered for the next semester)
3. Obtain reports on a particular student's status
4. Maintain information about students and courses
5. Enter final grades for courses that a student has completed

This brief description is typical of what might be supplied as a starting point for a system implementation project, but it is not specific or detailed enough to serve as the basis for the project's design and coding phases. We will be developing the student registration scenario throughout this book and will be using it to illustrate the various concepts in databases and transaction processing.

Our next step is to meet with the registrar, faculty, and students to expand this brief description into a formal Requirements Document for the system. We will discuss the Requirements Document in Chapter 14, which we expect you to read at appropriate times as you proceed through the rest of the book. In this chapter, we will take a closer look at some of the underlying concepts of databases and transaction processing that are needed for that system.

The following sections provide a brief overview of these concepts. Although we will revisit these concepts in a more detailed fashion in subsequent chapters, an overview will help you see the big picture and will set the stage for better understanding of the following chapters.

## 2.2 Introduction to Relational Databases

A database is at the heart of most transaction processing systems. At every instant of time, the database must contain an accurate description—often the only one—of the real-world enterprise the transaction processing system is modeling. For example, in the Student Registration System the database is the only source of information about which students have registered for each course.

**Relations and tuples.** We are particularly interested in databases that use the **relational model** [Codd 1970, 1990], in which data is stored in **tables**. The Student Registration System, for example, might include the STUDENT table, shown in Figure 2.1. A table contains a set of **rows**. In the figure, each row contains information about one student. Each **column** of the table describes the student in a particular way. In the example, the columns are **Id**, **Name**, **Address**, and **Status**. Each column has an associated type, called its **domain**, from which the value in a particular row for that column is drawn. For example, the domain for **Id** is integer and the domain for **Name** is string.

This database model is called “relational” because it is based on the mathematical concept of a relation. A **mathematical relation** captures the notion that elements of different sets are related to one another. For example, John Doe, an element of the set of all humans, is related to 123 Main St., an element of the set of all addresses, and to 111111111, an element of the set of all Ids. A relation is a set of **tuples**. Following the example of the table STUDENT, we might define a relation called STUDENT containing the tuple (111111111, John Doe, 123 Main St., Freshman). The STUDENT relation presumably contains a tuple describing every student.

We can view a relation as a predicate. A **predicate** is a declarative statement that is either true or false depending on the values of its arguments—for example, the predicate “It rained in Detroit on date X” is either true or false depending on the value chosen for the argument X. When we view a relation as a predicate, the arguments of the predicate correspond to the elements of a tuple, and the predicate is defined to be true for arguments  $a_1, \dots, a_n$  exactly when the tuple  $(a_1, \dots, a_n)$  is in the relation. For instance, we might define the predicate STUDENT

Id	Name	Address	Status
111111111	John Doe	123 Main St.	Freshman
666666666	Joseph Public	666 Hollow Rd.	Sophomore
111223344	Mary Smith	1 Lake St.	Freshman
987654321	Bart Simpson	Fox 5 TV	Senior
023456789	Homer Simpson	Fox 5 TV	Senior
123454321	Joe Blow	6 Yard Ct.	Junior

FIGURE 2.1 The table STUDENT. Each row describes a single student.

with arguments **Id**, **Name**, **Address**, and **Status**. Then we can say that the predicate **STUDENT** (111111111, John Doe, 123 Main St., Freshman) is true, because the tuple (111111111, John Doe, 123 Main St., Freshman) is in the table **STUDENT** shown in Figure 2.1.

The correspondence between tables and relations should now be clear: the tuples of a relation correspond to the rows of a table, and the column names of a table are the names of the **attributes** of the relation. Thus, the rows of the **STUDENT** table can be viewed as enumerating the set of all 4-tuples (tuples with four attributes of the appropriate types) that satisfy the **STUDENT** relation (i.e., the **Id**, **Name**, **Address**, and **Status** of a student).

**Operations on tables are mathematically defined.** In real applications, tables can become quite large—a **STUDENT** table for our university would contain over 15 thousand rows, and each row would likely contain much more information about each student than is shown here. In addition to the **STUDENT** table, the complete database for the Student Registration System at our university would contain a number of other tables, each with a large number of rows, containing information about other aspects of student registration. For example, a **TRANSCRIPT** table might contain a row for each course that every student has ever taken. Hence, the databases for most applications contain a large amount of information and are generally held in mass storage.

In most applications, the database is under the control of a database management system (DBMS), which is supplied by a commercial vendor. When an application wants to perform an operation on the database, it does so by making a request to the DBMS. A typical operation might extract some information from the rows of one or more tables, modify some rows, or add or delete rows. For example, when a new student is admitted to the university, a row is added to the **STUDENT** table.

In addition to the fact that tables in the database can be modeled by mathematical relations, operations on the tables can also be modeled as mathematical operations on the corresponding relations. Thus, a particular unary operation might take a table,  $T$ , as an argument and produce a result table containing a subset of the rows of  $T$ . For example, an instructor might want to display the roster of students registered for a course. Such a request might involve scanning the **TRANSCRIPT** table, locating the rows corresponding to the course, and returning them to the application. A particular binary operation might take two tables as arguments and construct a new table containing the union of the rows of the argument tables. A complex query against a database might be equivalent to an expression involving many such relational operations involving many tables.

Because of this mathematical description, relational operations can be precisely defined and their mathematical properties, such as commutativity and associativity, can be proven. As we shall see, this mathematical description has important practical implications. Commercial DBMSs contain a **query optimizer** module that converts queries into expressions involving relational operations and then uses these mathematical properties to simplify those expressions and thus optimize query execution.

**SQL: Basic SELECT statement.** An application describes the access that it wants the DBMS to perform on its behalf in a language supported by the DBMS. We are particularly interested in SQL, the most commonly used database language, which provides facilities for accessing a relational database and is supported by almost all commercial DBMSs.

The basic structure of the SQL statements for manipulating data is straightforward and easy to understand. Each statement takes one or more tables as arguments and produces a table as a result. For example, to find the name of the student whose Id is 987654321, we might use the statement

---

```
SELECT    Name
FROM      STUDENT
WHERE     Id = '987654321'
```

---

2.1

More precisely, this statement asks the DBMS to extract from the table named in the **FROM** clause—that is, the table **STUDENT**—all rows satisfying the condition in the **WHERE** clause—that is, all rows whose **Id** column has value 987654321—and then from each such row to delete all columns except those named in the **SELECT** clause—that is, **Name**. The resulting rows are placed in a result table produced by the statement. In this case, because **Ids** are unique, at most one row of **STUDENT** can satisfy the condition, and so the result of the statement is a table with one column and at most one row.

Thus, the **FROM** clause identifies the table to be used as input, the **WHERE** clause identifies the rows of that table from which the answer is to be generated, and the **SELECT** clause identifies the columns of those rows that are to be output in the result table.

The result table generated by this example contains only one column and at most one row. As a somewhat more complex example, the statement

---

```
SELECT    Id, Name
FROM      STUDENT
WHERE     Status = 'senior'
```

---

2.2

returns a result table (shown in Figure 2.2) containing two columns and multiple rows: the **Ids** and names of all seniors. If we want to produce a table containing all the columns of **STUDENT** but describing only seniors, we use the statement

---

```
SELECT    *
FROM      STUDENT
WHERE     Status = 'senior'
```

---

Id	Name
987654321	Bart Simpson
023456789	Homer Simpson

**FIGURE 2.2** The database table returned by the SQL SELECT statement (2.2).

The asterisk is simply shorthand that allows us to avoid listing the names of all the columns of STUDENT.

In some situations the user is interested not in outputting a result table but in information *about* the result table. An example is the statement

---

```
SELECT COUNT(*)
FROM STUDENT
WHERE Status = 'senior'
```

---

which returns the number of rows in the result table (i.e., the number of seniors). COUNT is referred to as an **aggregate** function because it produces a value that is a function of all the rows in the result table. Note that when an aggregate is used, the SELECT statement produces a single value instead of a table.

The WHERE clause is the most interesting component of the SELECT statement; it contains a general condition that is evaluated over each row of the table named in the FROM clause. Column values from the row are substituted into the condition, yielding an expression that has either a true or a false value. If the condition evaluates to true, the row is retained for processing by the SELECT clause and then stored in the result table. Hence, the WHERE clause acts as a filter.

Conditions can be much more complex than we have seen so far: A condition can be a Boolean combination of terms. If we want the result table to contain information describing seniors whose Ids are in a particular range, for example, we might use

---

```
WHERE Status = 'senior' AND Id > '888888888'
```

---

OR and NOT can also be used. Furthermore, a number of predicates are provided in the language for expressing particular relationships. For example, the IN predicate tests set membership.

---

```
WHERE Status IN ('freshman', 'sophomore')
```

---

Additional aggregates and predicates and the full complexity of the WHERE clause are discussed in Chapter 5.

**Multi-table SELECT statements.** The result table can contain information extracted from several base tables. Thus, if we have a table TRANSCRIPT with columns StudId, CrsCode, Semester, and Grade, the statement

---

```
SELECT  Name, CrsCode, Grade
FROM    STUDENT, TRANSCRIPT
WHERE   StudId = Id AND Status = 'senior'
```

---

can be used to form a result table in which each row contains the name of a senior, a particular course she took, and the grade she received.

The first thing to note is that the attribute values in the result table come from different base tables: Name comes from STUDENT; CrsCode and Grade come from TRANSCRIPT. As in the previous examples, the FROM clause produces a table whose rows are input to the WHERE clause. In this case the table is the Cartesian product of the tables listed in the FROM clause: a row of this table is the concatenation of a row of STUDENT and a row of TRANSCRIPT. Many of these rows make no sense. For example, Bart Simpson's row in STUDENT is not related to a row in TRANSCRIPT describing a course that Bart did not take. The first conjunct of the WHERE clause ensures that the rows of TRANSCRIPT for a particular student are associated with the appropriate row of STUDENT by matching the Id values of the rows of the two tables. For example, if TRANSCRIPT has a row (987654321, CS305, F1995, C), it will match only Bart Simpson's row in STUDENT, producing the row (Bart Simpson, CS305, C) in the result table.

**Query optimization.** One very important feature of SQL is that the programmer does not have to specify the algorithm the DBMS should use to satisfy a particular query. For example, tables are frequently defined to include auxiliary data structures, called **indices**, which make it possible to locate particular rows without using lengthy searches through the entire table. Thus, an index on the Id column of the STUDENT table might contain a list of pairs  $\langle Id, \text{pointer} \rangle$  where the pointer points to the row of the table containing the corresponding Id. If such an index were present, the DBMS would automatically use it to find the row that satisfies the query (2.1). If the table also had an index on the column Status, the DBMS would use that index to find the rows that satisfy the query (2.2). If this second index did not exist, the DBMS would automatically use some other method to satisfy (2.2)—for example, it might look at every row in the table in order to locate all rows having the value senior in the Status column. The programmer does not specify what method to use—just the condition the desired result table must satisfy.

In addition to selecting appropriate indices to use, the query optimizer uses the properties of the relational operations to further improve the efficiency with which a query can be processed—again, without any intervention by the programmer. Nevertheless, programmers should have some understanding of the strategies the DBMS uses to satisfy queries so they can design the database tables, indices, and

SQL statements in such a way that they will be executed in an efficient manner consistent with the requirements of the application.

**Changing the contents of tables.** The following examples illustrate the SQL statements for modifying the contents of a table. The statement

---

```
UPDATE STUDENT
SET Status = 'sophomore'
WHERE Id = '1111111111'
```

---

updates the STUDENT table to make John Doe a sophomore. The statement

---

```
INSERT INTO STUDENT (Id, Name, Address, Status)
VALUES ('9999999999', 'Winston Churchill', '10 Downing St',
'senior')
```

---

inserts a new row for Winston Churchill in the STUDENT table. The statement

---

```
DELETE FROM STUDENT
WHERE Id = '1111111111'
```

---

deletes the row for John Doe from the STUDENT table. Again, the details of how these operations are to be performed need not be specified by the programmer.

**Creating tables and specifying constraints.** Before you can store data in a table, the table structure must be created. For instance, the STUDENT table could have been created with the SQL statement

---

```
CREATE TABLE STUDENT(
Id          INTEGER,
Name        CHAR(20),
Address     CHAR(50),
Status      CHAR(10),
PRIMARY KEY(Id))
```

---

2.3

where we have declared the name of each column and the domain (type) of the data that can be stored in that column. We have also declared the Id column to be a primary key to the table, which means that each row of the table must have a unique value in that column and the DBMS will (most probably) automatically construct an index on that column. The DBMS will enforce this uniqueness constraint by not allowing any INSERT or UPDATE statement to produce a row with a value in the Id column that duplicates a value of Id in another row. This requirement is an

example of an **integrity constraint** (sometimes called a **consistency constraint**)—an application-based restriction on the values that can appear as entries in the database. We discuss integrity constraints in more detail in the next section.

We have given simple examples of each statement type to highlight the conceptual simplicity of the basic ideas underlying SQL, but be aware that the complete language has many subtleties. Each statement type has a large number of options that allow very complex queries and updates. For this reason, mastery of SQL requires significant effort. We continue our discussion of relational databases and SQL in Chapter 3.

## 2.3 What Makes a Program a Transaction— The ACID Properties

In many applications, a database is used to model the state of some real-world enterprise. In such applications, a transaction is a program that interacts with that database so as to maintain the correspondence between the state of the enterprise and the state of the database. In particular, a transaction might update the database to reflect the occurrence of a real-world event that affects the enterprise state. An example is a deposit transaction at a bank. The event is that the customer gives the teller the cash and a deposit slip. The transaction updates the customer's account information in the database to reflect the deposit.

Transactions, however, are not just ordinary programs. Requirements are placed on them, particularly on the way they are executed, that go beyond what is normally expected of regular programs. These requirements are enforced by the DBMS and the TP monitor.

**Consistency.** A transaction must access and update the database in such a way that it preserves all database integrity constraints. Every real-world enterprise is organized in accordance with certain rules that restrict the possible states of the enterprise. For example, the number of students registered for a course cannot exceed the number of seats in the room assigned to the course. When such a rule exists, the possible states of the database are similarly restricted.

The restrictions are stated as integrity constraints. The integrity constraint corresponding to the above rule asserts that the value of the database item that records the number of course registrants must not exceed the value of the item that records the room size. Thus, when the registration transaction completes, the database must satisfy this integrity constraint (assuming that the constraint was satisfied when the transaction started).

Although we have not yet designed the database for the Student Registration System, we can make some assumptions about the data that will be stored and postulate some additional integrity constraints:

- **IC0.** The database contains the Id of each student. These Ids must be unique.
- **IC1.** The database contains a list of prerequisites for each course and, for each student, a list of completed courses. A student cannot register for a course without having taken all prerequisite courses.

- **IC2.** The database contains the maximum number of students allowed to take each course and the number of students who are currently registered for each course. The number of students registered for each course cannot be greater than the maximum number allowed for that course.
- **IC3.** It might be possible to determine the number of students registered for (or enrolled in) a particular course from the database in two ways: the number is stored as a count in the information describing the course, and it can be calculated from the information describing each student by counting the number of student records that indicate that the student is registered for (or enrolled in) the course. These two determinations must yield the same result.

In addition to maintaining the integrity constraints, each transaction must update the database in such a way that the new database state reflects the state of the real-world enterprise that it models. If John Doe registers for CS305, but the registration transaction records Mary Smith as the new student in the class, the integrity constraints will be satisfied but the new state will be incorrect. Hence, consistency has two dimensions.

**Consistency.** The transaction designer can assume that when execution of the transaction is initiated, the database is in a state in which all integrity constraints are satisfied and, in addition, the database correctly models the current state of the enterprise. The designer has the responsibility of ensuring that when execution has completed, the database is once again in a state in which all integrity constraints are satisfied and, in addition, that the new state reflects the transformation described in the transaction's specification (in other words, that the database still correctly models the state of the enterprise).

SQL provides some support for the transaction designer in maintaining consistency. When the database is being designed, the database designer can specify certain types of integrity constraints and include them within the statements that declare the format of the various tables in the database. The primary key constraint of the SQL statement (2.3) is an example of this. Later, as each transaction is executed, the DBMS automatically checks that each specified constraint is not violated and prevents completion of any transaction that would cause a constraint violation.

**Atomicity.** In addition to the transaction designer's responsibility for consistency, the TP monitor must provide certain guarantees concerning the manner in which transactions are executed. One such condition is atomicity.

**Atomicity.** The system must ensure that the transaction either runs to completion or, if it does not complete, has no effect at all (as if it had never been started).

In the Student Registration System, either a student has registered for a course or he has not registered for a course. Partial registration makes no sense and might leave the database in an inconsistent state. For example, as indicated by constraint IC3, two items of information in the database must be updated when a student registers.

If a registration transaction were to have a partial execution in which one update completed but the system crashed before the second update could be executed, the resulting database would be inconsistent.

When a transaction has successfully completed, we say that it has **committed**. If the transaction does not successfully complete, we say that it has **aborted** and the TP monitor has the responsibility of ensuring that whatever partial changes the transaction has made to the database are undone, or **rolled back**. **Atomic execution** means that every transaction either commits or aborts.

Notice that ordinary programs do not necessarily have the property of atomicity. For example, if the system were to crash while a program that was updating a file was executing, the file could be left in a partially updated state when the system recovered.

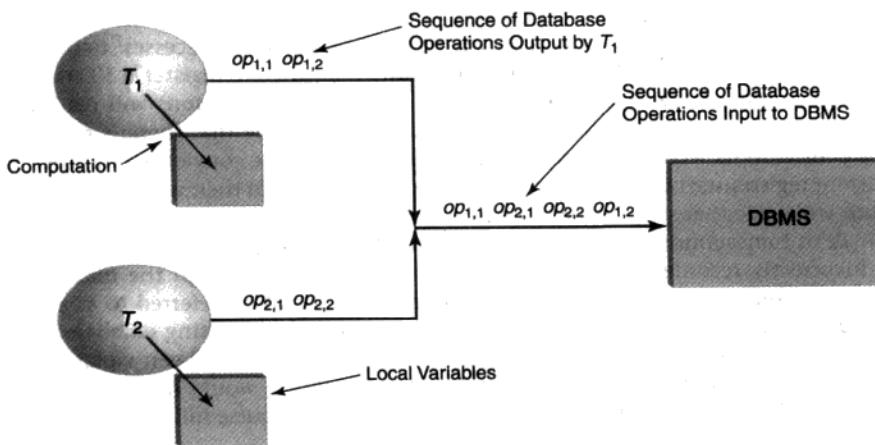
**Durability.** A second requirement of the transaction processing system is that it does not lose information.

**Durability.** The system must ensure that once the transaction commits, its effects remain in the database even if the computer, or the medium on which the database is stored, subsequently crashes.

For example, if you successfully register for a course, you expect the system to remember that you are registered even if it later crashes. Notice that ordinary programs do not necessarily have the property of durability either. For example, if a media failure occurs after a program that has updated a file has completed, the file might be restored to a state that does not include the update.

**Isolation.** In discussing consistency, we concentrated on the effect of a single transaction. We next examine the effect of executing a set of transactions. We say that a set of transactions is executed sequentially, or **serially**, if one transaction in the set is executed to completion before another is started. The good news about serial execution is that if all transactions are consistent and the database is initially in a consistent state, serial execution maintains consistency. When the first transaction in the set starts, the database is in a consistent state and, since the transaction is consistent, the database will be consistent when the transaction completes. Because the database is consistent when the second transaction starts, it too will perform correctly and the argument will repeat.

Serial execution is adequate for applications that have modest performance requirements. However, many applications have strict requirements on response time and throughput, and often the only way to meet the requirements is to process transactions concurrently. Modern computing systems are capable of servicing more than one transaction simultaneously, and we refer to this mode of execution as **concurrent**. Concurrent execution is appropriate in a transaction processing system serving many users. In this case, there will be many active, partially completed transactions at any given time.



**FIGURE 2.3** The database operations output by two transactions in a concurrent schedule might be interleaved in time. (Note that the figure should be interpreted as meaning that  $op_{1,1}$  arrives first at the DBMS, followed by  $op_{2,1}$ , etc.)

In concurrent execution, the database operations of different transactions are effectively interleaved in time, a situation shown in Figure 2.3. Transaction  $T_1$  alternately computes using its local variables and sends requests to the database system to transfer data between the database and its local variables. The requests are made in the sequence  $op_{1,1}$ ,  $op_{1,2}$ . We refer to that sequence as a **transaction schedule**.  $T_2$  performs its computation in a similar way. Because the execution of the two transactions is not synchronized, the sequence of operations arriving at the database, called a **schedule**, is an arbitrary merge of the two transaction schedules. The schedule in the figure is  $op_{1,1}$ ,  $op_{2,1}$ ,  $op_{2,2}$ ,  $op_{1,2}$ .

When transactions are executed concurrently, the consistency of each transaction is not sufficient to guarantee that the database that exists after both have completed correctly reflects the state of the enterprise. For example, suppose that  $T_1$  and  $T_2$  are two instances of the registration transaction invoked by two students who want to register for the same course. A possible schedule of these transactions is shown in Figure 2.4, where time progresses from left to right and the notation  $r(cur\_reg : n)$  means that a transaction has read the database object  $cur\_reg$ , which

**FIGURE 2.4** A schedule in which two registration transactions are not isolated from each other.

$T_1 : r(cur\_reg: 29)$

$w(cur\_reg: 30)$

$T_2 :$

$r(cur\_reg: 29)$

$w(cur\_reg: 30)$

records the number of current registrants, and the value  $n$  has been returned. A similar notation is used for  $w(cur\_reg : n)$ . The figure shows only the accesses<sup>1</sup> to  $cur\_reg$ .

Assume that the maximum number of students allowed to register is 30 and the current number is 29. In its first step, each of the two transactions will read this value and store it in its local variable, and both will decide that there is room in the course. In its second step, each will increment its private copy of the number of current registrants; hence, both will calculate the value 30. In their write operations, both will write that same value, 30, into  $cur\_reg$ .

Both transactions complete successfully, but the number of current registrants is incorrectly recorded as 30 when it is actually 31 (even though the maximum allowable number is 30). This is an example of what is often referred to as a **lost update** because one of the increments has been lost. The resulting database does not reflect the real-world state, and integrity constraint IC2 has been violated. By contrast, if the transactions had executed sequentially,  $T_1$  would have completed before  $T_2$  was allowed to start. Hence,  $T_2$  would find the course full and would not register the student.

As this example demonstrates, we must specify some restriction on concurrent execution that is guaranteed to maintain the consistency of the database and the correspondence between the enterprise state and the database state. One such restriction that is obviously sufficient follows.

**Isolation.** Even though transactions are executed concurrently, the overall effect of the schedule must be the same as if the transactions had executed serially in some order.

It should be evident that if the transactions are consistent and if the overall effect of a concurrent schedule is the same as that of some serial schedule, the concurrent schedule will maintain consistency. Concurrent schedules that satisfy this condition are called **serializable**.

As was the case with atomicity and durability, ordinary programs do not necessarily have the property of isolation. For example, if programs that update a common set of files are executed concurrently, updates might be interleaved and produce an outcome that is quite different from that obtained if they had been executed in any serial order. That result might be totally unacceptable.

**ACID properties.** The features that distinguish transactions from ordinary programs are frequently referred to by the acronym ACID [Haerder and Reuter 1983]:

- **Atomic.** Each transaction is executed completely or not at all.
- **Consistent.** Each transaction maintains database consistency.
- **Isolated.** The concurrent execution of a set of transactions has the same effect as some serial execution of that set.

<sup>1</sup> In a relational database,  $r$  and  $w$  represent SELECT and UPDATE statements.

- **Durable.** The effects of committed transactions are permanently recorded in the database.

When a transaction processing system supports the ACID properties, the database maintains a consistent and up-to-date model of the real world and the transactions supply responses to users that are always correct and up to date.

## BIBLIOGRAPHIC NOTES

The relational model for databases was introduced in [Codd 1970, 1990]. The SQL language is described by the various SQL standards, such as [SQL 1992]. The term “ACID” was coined by [Haerder and Reuter 1983], but the individual components of ACID were introduced in earlier papers—for example, [Gray et al. 1976] and [Eswaran et al. 1976].

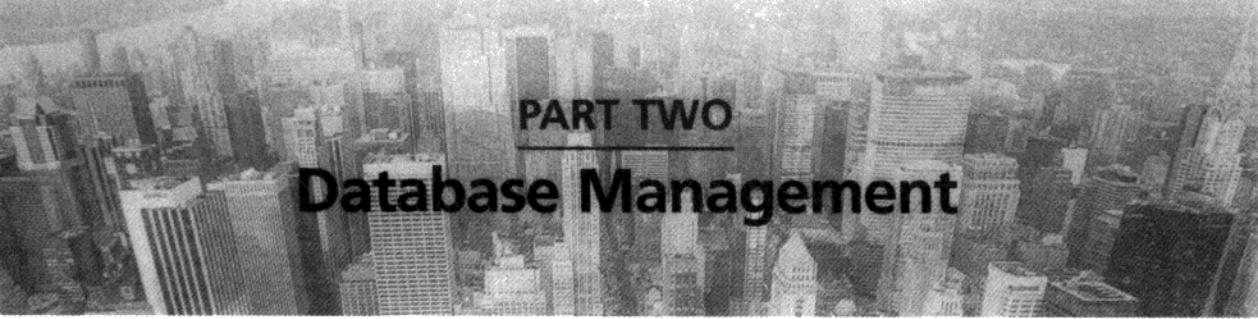
## EXERCISES

- 2.1 Given the relation MARRIED that consists of tuples of the form  $\langle a, b \rangle$ , where  $a$  is the husband and  $b$  is the wife, the relation BROTHER that has tuples of the form  $\langle c, d \rangle$ , where  $c$  is the brother of  $d$ , and the relation SIBLING, which has tuples of the form  $\langle e, f \rangle$ , where  $e$  and  $f$  are siblings, describe how you would define the relation BROTHER-IN-LAW, where tuples have the form  $\langle x, y \rangle$  with  $x$  being the brother-in-law of  $y$ .
- 2.2 Design the following two tables (in addition to that in Figure 2.1) that might be used in the Student Registration System. Note that the same student Id might appear in many rows of each of these tables.
  - a. A table implementing the relation COURSESREGISTEREDFOR, relating a student’s Id and the identifying numbers of the courses for which she is registered
  - b. A table implementing the relation COURSESTAKEN, relating a student’s Id, the identifying numbers of the courses he has taken, and the grade received in each courseSpecify the predicate corresponding to each of these tables.
- 2.3 Write an SQL statement that
  - a. Returns the Ids of all seniors in the table STUDENT
  - b. Deletes all seniors from STUDENT
  - c. Promotes all juniors in the table STUDENT to seniors
- 2.4 Write an SQL statement that creates the TRANSCRIPT table.
- 2.5 Using the TRANSCRIPT table, write an SQL statement that
  - a. Deregisters the student with Id = 123456789 from the course CS305 for the fall of 2001
  - b. Changes to an A the grade assigned to the student with Id = 123456789 for the course CS305 taken in the fall of 2000
  - c. Returns the Id of all students who took CS305 in the fall of 2000

- 2.6 Write an SQL statement that returns the names (not the IDs) of all students who received an A in CS305 in the fall of 2000.
- 2.7 State whether or not each of the following statements could be an integrity constraint of a checking account database for a banking application. Give reasons for your answers.
- The value stored in the balance column of an account is greater than or equal to \$0.
  - The value stored in the balance column of an account is greater than it was last week at this time.
  - The value stored in the balance column of an account is \$128.32.
  - The value stored in the balance column of an account is a decimal number with two digits following the decimal point.
  - The social\_security\_number column of an account is defined and contains a nine-digit number.
  - The value stored in the check\_credit\_in\_use column of an account is less than or equal to the value stored in the total\_approved\_check\_credit column. (These columns have their obvious meanings.)
- 2.8 State five integrity constraints, other than those given in the text, for the database in the Student Registration System.
- 2.9 Give an example in the Student Registration System where the database satisfies the integrity constraints IC0–IC3 but its state does not reflect the state of the real world.
- 2.10 State five (possible) integrity constraints for the database in an airline reservation system.
- 2.11 A reservation transaction in an airline reservation system makes a reservation on a flight, reserves a seat on the plane, issues a ticket, and debits the appropriate credit card account. Assume that one of the integrity constraints of the reservation database is that the number of reservations on each flight does not exceed the number of seats on the plane. (Of course, many airlines purposely over-book and so do not use this integrity constraint.) Explain how transactions running on this system might violate
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- 2.12 Describe informally in what ways the following events differ from or are similar to transactions with respect to atomicity and durability.
- A telephone call from a pay phone (Consider line busy, no answer, and wrong number situations. When does this transaction “commit?”)
  - A wedding ceremony (Suppose that the groom refuses to say “I do.” When does this transaction “commit?”)
  - The purchase of a house (Suppose that, after a purchase agreement is signed, the buyer is unable to obtain a mortgage. Suppose that the buyer backs out during the closing. Suppose that two years later the buyer does not make the mortgage payments and the bank forecloses.)
  - A baseball game (Suppose that it rains.)

- 2.13 Assume that, in addition to storing the grade a student has received in every course he has completed, the system stores the student's cumulative GPA. Describe an integrity constraint that relates this information. Describe how the constraint would be violated if the transaction that records a new grade were not atomic.
- 2.14 Explain how a lost update could occur if, under the circumstances of the previous problem, two transactions that were recording grades for a particular student (in different courses) were run concurrently.





## PART TWO

---

# Database Management

Now we are ready to begin a more in-depth study of databases.

In Chapter 3, we will discuss how data items are specified in modern database management systems and how they appear to the transactions that use them. In other words, we will learn a few things about data models and data definition languages.

In Chapter 4, we will study conceptual database design, which includes methodologies for organizing data around a set of high-level concepts.

In Chapter 5, we will discuss how transactions access and modify data in a DBMS using data manipulation and query languages—in particular, SQL.

In Chapter 6, we will resume the design theme and will talk about the Relational Normalization Theory. This theory provides algorithms and objective measures for improving the quality of database design.

Chapter 7 introduces the mechanism of triggers—a powerful device for maintaining the consistency of databases and for enabling databases to react to external events.

Chapter 8 concludes this part of the book with a discussion of how SQL statements can be executed from within a host language, such as C or Java.



# 3

## The Relational Data Model

This chapter is an introduction to the relational data model. First we define its main abstract concepts, and then we show how these concepts are embodied in the concrete syntax of SQL. Specifically, this chapter covers the data definition subset of SQL, which is used to specify data structures, constraints, and authorization policies in databases.

### 3.1 What Is a Data Model?

**Data independence.** Ultimately, all data is recorded as bytes on a disk. However, as a programmer you know that working with data at this low level of abstraction is quite tedious. Few people are interested in how sectors, tracks, and cylinders are allocated for storing information. Most programmers much prefer to work with data stored in *files*, which is a more reasonable abstraction for many applications.

From a course on file structures, you might be familiar with a variety of methods for storing data in files. **Sequential** files are best for applications that access records in the order in which they are stored. **Direct access** (or **random access**) files are best when records are accessed in a more or less unpredictable order. Files might have **indices**, which are auxiliary data structures that enable applications to retrieve records based on the value of a **search key**. We will discuss various index types in Chapter 9. Files might also consist of fixed-length records or records that have variable lengths.

The details of how data is stored in files belong to the **physical level** of data modeling. This level is specified using a **physical schema**, which in the field of databases refers to the syntax that describes the structure of files and indices.

Early data-intensive applications worked directly with the physical schema instead of the higher levels of abstraction provided by a modern DBMS. This choice was made for a number of reasons. First, commercial database systems were rare and costly. Second, computers were slow, and working directly with the file system offered a performance advantage. Third, most early applications were primitive by today's standards, and building a level of abstraction between those programs and the file system did not seem justified.

A serious drawback of this approach is that changes to the file format at the physical level could have costly repercussions for software maintenance. The “year 2000 problem” was a good example of such repercussions. In the 1960s and 1970s, it was common to write programs in which the data item representing the calendar year was hard-coded as a two-digit number. The rationale was that these programs would be replaced within fifteen to twenty years, so using four digits (or using a data abstraction for the DATE data type) was a waste of precious disk space. The result was that every routine that worked with dates expected to find the year in the two-digit format. Hence, any change to that format implied finding and changing code throughout the application. The consequence of these past decisions was the multibillion-dollar bill presented to the industry in the late 1990s for fixing outdated software.

If a data abstraction, DATE, had been used in those programs, the whole problem could have been avoided. Applications would have viewed years as four-digit numbers, even though they had been physically stored in the database as two-digit numbers. To adjust to the change of millennium, designers could have changed the underlying physical representation of years in the database to four-digit numbers by (1) building a simple program that converted the database by adding “1900” to every existing year field and (2) correspondingly changing the implementations of the appropriate functions within the DATE data type to access the new physical representation. None of the existing applications would have had to be modified because they could still use the same DATE data abstraction.

When the underlying data structures are subject to change (even infrequently), basing the design of data-intensive applications on a bare file system becomes problematic. Even trivial changes, such as adding or deleting a field in a file, imply that every application that uses this file must be manually updated, recompiled, and retested. Less trivial changes, such as merging two fields or splitting a field into two, might impact the existing applications quite significantly. Accommodating such changes can be labor intensive and error prone. In addition, the data in the original file needs to be converted to the new representation, and without the appropriate tools such conversion can be costly.

Also, the file system offers too low a level of abstraction to support the development of an application that requires frequent and rapid implementation of new queries. For such applications, the conceptual level of data modeling becomes appropriate.

The conceptual model hides the details of the physical data representation and instead describes data in terms of higher-level concepts that are closer to the way humans view it. For instance, the **conceptual schema**—the syntax used to describe the data at the conceptual level—could represent some of the information about students as

---

STUDENT (Id: INT, Name: STRING, Address: STRING, Status: STRING)

---

While this schema might look similar to the way file records are represented, the important point is that the different pieces of information it describes might be *physically* stored in a different way than that described in the schema. Indeed, these pieces of information might not even reside in the same file (perhaps not even on the same computer!).

The possibility of having separate schemas at the physical and conceptual levels leads to the simple, yet powerful, idea of **physical data independence**. Instead of working directly with the file system, applications see only the conceptual schema. The DBMS maps data between the conceptual and physical levels *automatically*. If the physical representation changes, all that needs to be done is to change the mapping between the levels, and *all* applications that deal exclusively with the conceptual schema will continue to work with the new physical data structures.

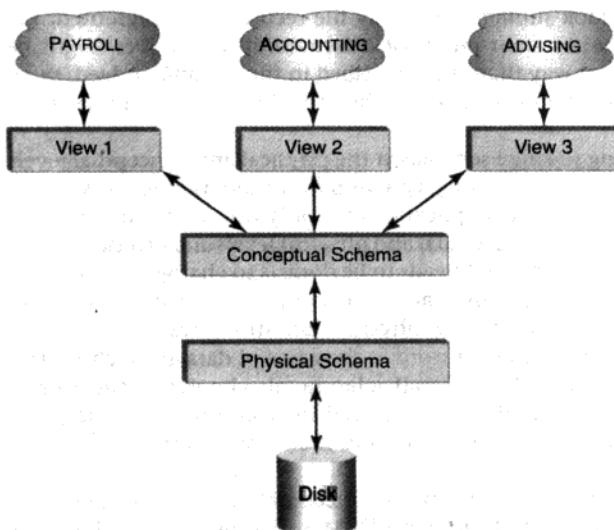
The conceptual schema is not the last word in the game of data abstraction. The third level of abstraction is called the **external schema** (also known as the **user** or **view** abstraction level). The external schema is used to customize the conceptual schema to the needs of various classes of users, and it also plays a role in database security (as we will see later).

The external schema looks and feels like a conceptual schema, and both are defined in essentially the same way in modern DBMSs. However, while there is a single conceptual schema per database, there might be several external schemas (i.e., views on the conceptual schema), usually one per user category. For example, to generate proper student billing information, the bursar's office might need to know each student's GPA and status and the total number of credits the student has taken, but not the names of the courses and the grades received. Even though the GPA and total number of credits might not be stored in the database explicitly, the bursar's office can be presented with a view in which these items appear as regular fields (whose values are calculated at run time when the field is accessed), and all fields and relations that are irrelevant to billing are omitted. Similarly, an academic advisor does not need to know anything about billing, so much of this information can be omitted from the advisor's view of the registration system.

These ideas lead to the principle of **conceptual data independence**: Applications tailored to the needs of specific user groups can be designed to use the external schemas appropriate for these groups. The mapping between the external and conceptual schemas is the responsibility of the DBMS, so applications are insulated from changes in the conceptual schema *as well as* from changes in the physical schema. The overall picture is shown in Figure 3.1.

**Data model.** A **data model** consists of a set of concepts and languages for describing

1. **Conceptual and external schemas.** A schema specifies the structure of the data stored in the database. Schemas are described using a **data definition language (DDL)**.



**FIGURE 3.1** Levels of data independence.

2. **Constraints.** A constraint specifies a condition that the data items in the database must satisfy. A constraint specification sublanguage is usually part of the DDL.
3. **Operations on data.** Operations on database items are described using a **data manipulation language (DML)**. The DML is usually the most important and interesting part of any data model because it is the set of operations that ultimately gives us the high-level data abstraction.

In addition, all commercial systems provide some kind of **storage definition language (SDL)**, which allows the database designer to *influence* the physical schema (although most systems reserve the final say). The SDL is usually tightly integrated with the DDL. Changes in the physical schema that might occur if the database administrator introduces new SDL statements into a database do not affect the semantics of the applications because physical data independence shields the application from changes at the storage level. Hence, although the performance of an application might change, the results it produces do not.

In Sections 3.2 and 3.3, we describe the mother of all data models used by commercial DBMSs, the *relational model*, and the *lingua franca* these DBMS speak, **Structured Query Language (SQL)**. Be aware, however, that despite its name SQL is not just a query language; it is an amalgamation of a DML, a DDL, and an SDL—three for the price of one!

## 3.2 The Relational Model

The *relational data model* was proposed in 1970 by E. F. Codd and was considered a major breakthrough at the time. In fact, database research and development in the 1970s and 1980s was largely shaped by the ideas presented in Codd's original work [Codd 1970, 1990]. Even today, most commercial DBMSs are based on the relational model, although they are beginning to acquire object-oriented features, especially due to the increased use of XML-based data.

The main attraction of the relational model is that it is built around a simple and natural mathematical structure—the *relation* (or table). Relations have a set of powerful, high-level operators, and data manipulation languages are deeply rooted in mathematical logic. This solid mathematical background means that relational expressions (i.e., queries) can be analyzed. Hence, any expression can potentially be transformed (by the DBMS itself) into another, *equivalent*, expression that can be executed more efficiently, in a process called *query optimization*. Thus, application programmers need not study the nitty-gritty details of the internals of each database and need not be aware of how query evaluators work. The application programmer can formulate a query in a simple and natural way and leave it to the query optimizer to find an equivalent query that is more efficient to execute.

Nevertheless, query optimizers have limitations that can result in performance penalties for certain classes of complex queries. It is therefore important for both programmers and database designers to understand the heuristics they use. With this knowledge, programmers can formulate queries that the DBMS can optimize more easily, and database designers can speed up the evaluation of important queries by adding appropriate indices and using other design techniques.

### 3.2.1 Basic Concepts

The central construct in the relational model is the **relation**. A relation is two things in one: a **schema** and an **instance** of that schema.

**Relation instance.** A **relation instance** is nothing more than a table with rows and named columns. When no confusion arises, we refer to relation instances as just “relations.” The rows in a relation are called *tuples*; they are similar to *records* in a file, but unlike file records all tuples have the same number of columns (this number is called the **arity** of the relation), and no two tuples in a relation instance can be the same. In other words, a relational instance is a *set* of unique tuples. The **cardinality** of a relation instance is the number of tuples in it.

Figure 3.2 shows one possible instance for the STUDENT relation. The columns in this relation are named, which is the usual convention in the relational model. These named columns are also known as **attributes**. Because relations are sets of tuples, the order of these tuples is considered immaterial. Similarly, because columns are named, their order in a table is of no importance either. The relations in Figures 3.2 and 3.3 are thus considered to be the same relation.

STUDENT	Id	Name	Address	Status
	1111111111	John Doe	123 Main St.	Freshman
	6666666666	Joseph Public	666 Hollow Rd.	Sophomore
	111223344	Mary Smith	1 Lake St.	Freshman
	987654321	Bart Simpson	Fox 5 TV	Senior
	023456789	Homer Simpson	Fox 5 TV	Senior
	123454321	Joe Blow	6 Yard Ct.	Junior

FIGURE 3.2 Instance of the STUDENT relation.

STUDENT	Id	Name	Status	Address
	111223344	Mary Smith	Freshman	1 Lake St.
	987654321	Bart Simpson	Senior	Fox 5 TV
	1111111111	John Doe	Freshman	123 Main St.
	023456789	Homer Simpson	Senior	Fox 5 TV
	6666666666	Joseph Public	Sophomore	666 Hollow Rd.
	123454321	Joe Blow	Junior	6 Yard Ct.

FIGURE 3.3 STUDENT relation with different order of columns and tuples.

We should note that the terms “tuple,” “attribute,” and “relation” are preferred in relational database theory, while “row,” “column,” and “table” are the terms used in SQL. However, it is common to use these terms interchangeably.

The value of a particular attribute in any row of a relation is drawn from a set called the **attribute domain**—for example, the Address attribute of the STUDENT relation has as its domain the set of all strings. One important requirement placed on the values in a domain is **data atomicity**.<sup>1</sup> Data atomicity does not mean that these values are not decomposable. After all, we have seen that the values can be strings of characters, which means that they *are* decomposable. Rather, data atomicity means that the relational model does not specify any means for looking into the internal structure of the values, so that the values appear indivisible to the relational operators.

This atomicity restriction is sometimes seen as a shortcoming of the relational model, and most commercial systems relax it in various ways. Some remove it altogether, which leads to a breed of data models known as *object-relational*. We will return to the object-relational model in Chapter 16.

<sup>1</sup> The notion of *data atomicity* should not be confused with the unrelated notion of *transaction atomicity*, which we discussed in Section 2.3.

**Brain Teaser:** Can a relation have zero attributes?

**Relation schema.** A **relation schema** consists of

1. The **name** of the relation. Relation names must be unique across the database.
2. The names of the *attributes* in the relation along with their associated *domain names*. An **attribute** is simply the name given to a column in a relation instance. All columns in a relation must be named, and no two columns in the same relation can have the same name. A **domain name** is just a name given to some well-defined set of values. In programming languages, domain names are usually called *types*. Examples are INTEGER, REAL, and STRING.
3. The *integrity constraints (IC)*. **Integrity constraints** are restrictions on the relational instances of this schema (i.e., restrictions on which tuples can appear in an instance of the relation). An instance of a schema is said to be **legal** if it satisfies all ICs associated with the schema.

To illustrate, let us revisit the schema that was mentioned before:

---

**STUDENT(Id:INTEGER, Name:STRING, Address:STRING, Status:STRING)**

---

This schema states that STUDENT relations must have exactly four attributes: Id, Name, Address, and Status with associated domains INTEGER and STRING. As seen from this example, different attributes in the same schema must have distinct names but can share domains.

The domains specify that in STUDENT relations all values in the column Id must belong to the domain INTEGER, while the values in all other columns must belong to the domain STRING. Naturally, we assume that the domain INTEGER consists of all integers and that the domain STRING consists of all character strings. However, schemas can also have *user-defined* domains, such as SSN or STATUS, that can be constrained to contain precisely the values appropriate for the attributes at hand. For instance, the domain STATUS can be defined to consist just of the symbols "freshman," "sophomore," and so forth, and the domain SSN can be defined to contain all (and only) nine-digit positive numbers. The point of this discussion is that relation schemas impose so-called type constraints.

A **type constraint** is a requirement that if **S** is a relation schema and **s** is a relation instance, then **s** must satisfy the following two conditions:

1. *Column naming.* Each column in **s** must correspond to an attribute in **S** (and vice versa), and the column names must be the same as the names of the corresponding attributes.
2. *Domain constraints.* For each attribute-domain pair, attr:DOM, in **S**, the values that appear in the column attr in **s** must belong to the domain DOM.

**FIGURE 3.4** Fragment of the Student Registration database schema.

```
STUDENT (Id:INTEGER, Name:STRING, Address:STRING, Status:STRING )
PROFESSOR (Id:INTEGER, Name:STRING, DeptId:STRING)
COURSE (DeptId:STRING, CrsCode:STRING, CrsName:STRING, Descr:STRING)
TRANSCRIPT (StudId:INTEGER, CrsCode:STRING, Semester:STRING, Grade:STRING)
TEACHING (ProfId:INTEGER, CrsCode:STRING, Semester:STRING)
```

As we shall see, typing is just one of the several classes of constraints associated with relation schemas. To be legal, a schema instance must therefore satisfy typing as well as those additional constraints.

**Relational database.** A **relational database** is a finite set of relations. Because a relation is two things in one, a database is also two things: a set of relation schemas (and other entities that we will describe shortly)—called a **database schema**—and a set of corresponding relation instances—called a **database instance**. When confusion does not arise, it is common to use the term “database” to refer to database instances only. Figure 3.4 depicts one possible fragment of a database schema for our Student Registration System. Figure 3.5 gives examples of instances corresponding to these relation schemas. Observe that each relation satisfies the type constraint specified by the corresponding schema.

*Brain Teaser:* If you solved the previous teaser, what are the tuples of a 0-ary relation? How many tuples can such a relation have?

### 3.2.2 Integrity Constraints

We discussed the role that integrity constraints play in an application in Section 2.2. Now we have to fit these constraints into the database schema that supports that application. An **integrity constraint** (IC) is a statement about all *legal instances* of a database. That is, to be qualified as a legal instance a set of relations must satisfy all ICs associated with the database schema. We have already seen the type and domain constraints, and we will discuss several other kinds of constraints later.

Some integrity constraints are based on the business rules of the enterprise. The statement “No employee can earn more than his boss” is one example. Such constraints are often listed in the Requirements Document of the application. Other constraints, such as type and domain constraints, are based on the schema design and are specified by the database designer.

Since ICs are part of the database schema, they are usually specified in the original schema design. It is also possible to add or remove ICs later, after the database has been created and populated with data. Once constraints have been specified in the schema, it is the responsibility of the DBMS to make sure that they are not violated by the execution of any transactions.

PROFESSOR	Id	Name	DeptId
	101202303	John Smyth	CS
	783432188	Adrian Jones	MGT
	121232343	David Jones	EE
	864297531	Qi Chen	MAT
	555666777	Mary Doe	CS
	009406321	Jacob Taylor	MGT
	900120450	Ann White	MAT

COURSE	CrsCode	DeptId	CrsName	Descr
	CS305	CS	Database Systems	On the road to high-paying job
	CS315	CS	Transaction Processing	Recover from your worst crashes
	MGT123	MGT	Market Analysis	Get rich quick
	EE101	EE	Electronic Circuits	Build your own computer
	MAT123	MAT	Algebra	The world where $2 * 2 \neq 4$

TRANSCRIPT	StudId	CrsCode	Semester	Grade
	666666666	MGT123	F1994	A
	666666666	EE101	S1991	B
	666666666	MAT123	F1997	B
	987654321	CS305	F1995	C
	987654321	MGT123	F1994	B
	123454321	CS315	S1997	A
	123454321	CS305	S1996	A
	123454321	MAT123	S1996	C
	023456789	EE101	F1995	B
	023456789	CS305	S1996	A
	111111111	EE101	F1997	A
	111111111	MAT123	F1997	B
	111111111	MGT123	F1997	B

FIGURE 3.5 Examples of database instances.

TEACHING	ProfId	CrsCode	Semester
	009406321	MGT123	F1994
	121232343	EE101	S1991
	555666777	CS305	F1995
	864297531	MGT123	F1994
	101202303	CS315	S1997
	900120450	MAT123	S1996
	121232343	EE101	F1995
	101202303	CS305	S1996
	900120450	MAT123	F1997
	783432188	MGT123	F1997
	009406321	MGT123	F1997

FIGURE 3.5 (continued)

An IC can be **intrarelational**, meaning that it involves only one relation, or it can be **interrelational**, meaning that it involves more than one relation. The type constraint is an example of an intrarelational constraint; another example is a constraint that states that the value of the Id attribute in all rows of an instance of the STUDENT table must be unique. The latter is called a *key constraint* (discussed later). The constraint that asserts that the value of the attribute Id of each professor shown as teaching a course must appear as the value of the Id attribute of some row of the table PROFESSOR is an example of an interrelational constraint called a *foreign-key constraint* (discussed later). It expresses the requirement that each faculty member teaching a course must be described by some row of the table that describes all faculty members. The constraint that no employee can earn more than the boss<sup>2</sup> can be intrarelational or interrelational, depending on whether the salary information and the management structure information are stored in the same or different relations. This constraint belongs to the class of **semantic constraints**, which we will discuss later in this section.

The constraints up to this point were **static ICs**. **Dynamic ICs** are different: instead of restricting the legal instances of a database, they restrict the evolution of legal instances. This type of constraint is particularly useful for representing the business rules of an enterprise. An example is a rule that salaries must not increase or decrease by more than 5% per transaction. Another example is a rule that the marital status of a person cannot change from single to divorced. A bank might have a rule that if an overdraft has been made, it must be covered by the end of the next business day through a transfer of funds from the line-of-credit account.

<sup>2</sup> Let us not worry about such subtleties as how this constraint applies to the company president, who has no boss.

Unfortunately, the mainstream data manipulation languages (such as SQL) and commercial DBMSs provide little support for automatic enforcement of dynamic constraints. Therefore, application designers must provide code that enforces such constraints within the transactions that update the database. Because there is no easy way to verify that the transactions actually obey those rules, the integrity of such databases depends on the competence of the design, coding, and quality assurance groups that implement the transactions.

The situation with static ICs is much more satisfactory. Such constraints are both easier to specify and—in most cases—easier to enforce than dynamic ICs. In this section, we discuss the most common static integrity constraints. Section 3.3 shows how they are specified in SQL.

**Key constraints.** We have already seen one example of a key constraint: values of the `Id` attribute in an instance of the `STUDENT` table must be unique. For a more complex example, consider the `TRANSCRIPT` relation. Because it seems reasonable to assume that a student can get only one (final) grade for any course in any given semester, we can specify that  $\{\text{StudId}, \text{CrsCode}, \text{Semester}\}$  is a key. This specification ensures that for any given value for `StudId`, `CrsCode`, and `Semester`, there is *at most* one transcript record with these values. If such a tuple actually exists, it specifies the one and only grade that a given student got for a given course in a given semester.

With this intuition in mind, we can give a more precise definition of a key constraint. A **key constraint**,  $\text{key}(\bar{K})$ , associated with a relation schema,  $S$ , consists of a subset,  $\bar{K}$  (called a **key**), of attributes in  $S$  with the following **minimality property**: if  $\bar{L}$  is a proper subset of  $\bar{K}$  then  $\text{key}(\bar{L})$  cannot be specified as a key constraint in the same schema  $S$ . A relation instance,  $s$ , of the schema  $S$  **satisfies** the constraint  $\text{key}(\bar{K})$  if it has the following **uniqueness property**:  $s$  does not contain a pair of distinct tuples whose values agree on *all* of the attributes in  $\bar{K}$ .

Therefore, it is an error to specify, for example, both  $\text{key}(A)$  and  $\text{key}(A, B)$  as key constraints in the same relation schema. Also, if  $\{A, B\}$  is a key, then at most one tuple can have a given pair of values,  $a$  and  $b$ , in attributes  $A$  and  $B$ , respectively. However, it is possible for two different tuples to have the same value in the attribute  $A$  but not in  $B$ , and vice versa.

**Example 3.2.1 (Key Constraint).** The `TRANSCRIPT` relation of Figure 3.5 satisfies the uniqueness property for the constraint  $\text{key}(\text{StudId}, \text{CrsCode}, \text{Semester})$  since there are no distinct tuples whose values agree on each of these three attributes. On the other hand, this relation does not satisfy the constraint  $\text{key}(\text{StudId}, \text{CrsCode})$  because, for example, tuples 1 and 3 are distinct and yet their values over `StudId` and `CrsCode` are the same. Similarly, the constraint  $\text{key}(\text{StudId}, \text{Semester})$  is not satisfied because of, say, the last two tuples.

Note that some particular instance of `TRANSCRIPT` *could* satisfy the uniqueness property for, say,  $\text{key}(\text{StudId}, \text{CrsCode})$ . However, since this is not a reasonable constraint (since it implies that students are not allowed to re-take courses), it is unlikely to appear as one of the constraints for the Student Registration System. Note also that if this *were* specified as a constraint after all, then  $\text{key}(\text{StudId}, \text{CrsCode})$ ,

`Semester`) could not have been a constraint at the same time, due to the minimality property.

Thus, if we decided to adopt the constraint `key(StudId, CrsCode, Semester)` for our system, then it would be allowed to have different tuples that record the same course taken by the same student, provided that this course was taken in different semesters (this would correspond to re-taking the course). It would also be allowed for any student to take several different courses during the same semester. However, it would not be possible for a student to get two different grades for the same course in the same semester. ■

The following points are important for understanding the notion of a key:

1. *Superset of a key has key-like properties.* If `key( $\bar{K}$ )` is a key constraint in schema  $S$ , and  $\bar{L}$  is a set of attributes in  $S$  that contains  $\bar{K}$ , then legal instances of  $S$  cannot have *distinct* tuples that agree on every attribute in  $\bar{L}$ . Indeed, if  $t$  and  $s$  are tuples that have the same values for each attribute in  $\bar{L}$ , then they must have the same values for each attribute in  $\bar{K}$ . But because `key( $\bar{K}$ )` is a key constraint, it must have the uniqueness property, and thus  $t$  and  $s$  must be the same tuple.

These ideas lead to the following notion: a set of attributes in  $S$  that contains a key is called a **superkey** of  $S$ . Thus, every key is also a superkey. The converse is not always true. For instance, in our TRANSCRIPT example, `{StudId, CrsCode, Semester, Grade}` is a superkey but not a key (because `{StudId, CrsCode, Semester}` is said to be a key, and they both cannot be keys at the same time, due to the minimality property). In other words, a superkey is like a key but without the minimality condition.

2. *Every relation has a key (and hence a superkey).* Indeed, the set of all attributes in a schema,  $S$ , is always a superkey because if a legal instance of  $S$  has a pair of tuples that agree on all attributes in  $S$ , then these must be identical tuples: since relations are sets, they cannot have identical elements. Now, if the set of all attributes in  $S$  is not a minimal superkey, there must be a superkey that is a strict subset of  $S$ . If that superkey is not a minimal superkey, there must be an even smaller superkey. As the number of attributes in a relation is finite, we will eventually hit the minimal superkey, which must then be a key, by definition.
3. *A schema can have several different keys.* For instance, in the COURSE relation, `CrsCode` can be one key. But because it is unlikely that the same department will offer two different courses with the same name, we can specify that `{DeptId, CrsName}` is also a key in the same relation.

If a relation has several keys, they are referred to as **candidate keys**. However, one key is often designated as the **primary key**. A primary key might or might not have any particular semantic significance in the application (often a primary key is just the first among equals). However, commercial DBMSs treat primary keys as hints for optimizing the storage structures to enable efficient access to data whenever the value of a primary key is given. Thus, the choice of a primary key affects the physical schema and may affect performance.

**FIGURE 3.6** Fragment of the Student Registration database with key constraints.

```

STUDENT(Id:INTEGER, Name:STRING, Address:STRING, Status:STRING)
  Key: {Id}
PROFESSOR(Id:INTEGER, Name:STRING, DeptId:STRING)
  Key: {Id}
COURSE(CrsCode:STRING, DeptId:STRING, CrsName:STRING, Descr:STRING)
  Keys: {CrsCode}, {DeptId,CrsName}
TRANSCRIPT(StudId:INTEGER, CrsCode:STRING, Semester:STRING, Grade:STRING)
  Key: {StudId,CrsCode,Semester}
TEACHING(ProfId:INTEGER, CrsCode:STRING, Semester:STRING)
  Key: {CrsCode,Semester}

```

---

Our fragment of the student registration database schema, with all of the key constraints included, is summarized in Figure 3.6. Note that the relation COURSE has two keys.

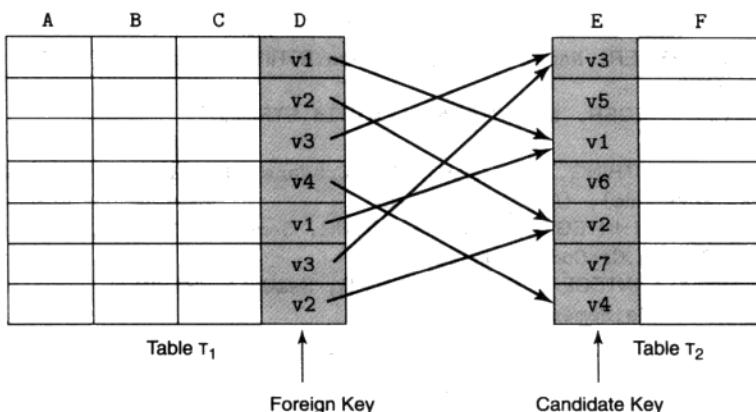
**Referential integrity.** In relational databases, it is common for tuples in one relation to reference tuples in the same or other relations. For instance, the value 009406321 of ProfId in the first tuple in TEACHING (Figure 3.5) refers to Professor Jacob Taylor, whose tuple in the table PROFESSOR has the same value in the Id field. Likewise, the value MGT123 in the first tuple of TEACHING references the Market Analysis course described by a tuple in the COURSE table.

In many situations, it is a violation of data integrity if the referenced tuple does not exist in the appropriate relation. For instance, it makes little sense to have a TEACHING tuple (009406321, MGT123, F1994) and not have the tuple describing MGT123 in the COURSE relation: otherwise, which course is Jacob Taylor teaching? Likewise, if the PROFESSOR relation has no tuple with Id 009406321, who is said to be teaching Market Analysis?

The requirement that the referenced tuples must exist (when the semantics of the data so requires) is called **referential integrity**. One important type of referential integrity is the *foreign-key constraint*.

Suppose that S and T are relation schemas,  $\bar{F}$  is a list of attributes in S, and key( $\bar{K}$ ) is a key constraint in T. Suppose further that there is a known 1-1 correspondence between the attributes of  $\bar{F}$  and  $\bar{K}$  (but the names of the attributes in  $\bar{F}$  and  $\bar{K}$  need not be the same). We say that relation instances s and t (over schemas S and T, respectively) satisfy the **foreign-key constraint** "S( $\bar{F}$ ) references T( $\bar{K}$ )" and that  $\bar{F}$  is a **foreign key** if and only if, for every tuple  $s \in s$ , there is a tuple  $t \in t$  that has the same values over the attributes in  $\bar{K}$  as does  $s$  over the corresponding attributes in  $\bar{F}$ .

The concept of a foreign-key constraint is illustrated in Figure 3.7, where attribute D in table T<sub>1</sub> has been declared a foreign key that refers to attribute E in table T<sub>2</sub>. E must be a candidate (or primary) key of T<sub>2</sub>, but note that the names of the referring and referenced attributes (D and E) in the two tables need not be the same. Note also that, although each row of T<sub>1</sub> must reference exactly one row of T<sub>2</sub>, not all



**FIGURE 3.7** Attribute D in Table  $T_1$  is a foreign key that refers to the candidate key, attribute E, in Table  $T_2$ .

rows of  $T_2$  need be referenced, and two or more rows in  $T_1$  can reference the same row in  $T_2$ .

In Section 3.3, we will show how foreign-key constraints are specified in SQL and will discuss the precise semantics of such constraints (which are slightly more permissive than the definition we have just given).

In a foreign-key constraint, the referring and the referenced relations need not be distinct. For instance, in the schema

---

**EMPLOYEE(Id:INTEGER, Name:STRING, MngrId:INTEGER)**

---

with the key {Id} the supervisor is also an employee. Thus, we have the constraint “EMPLOYEE(MngrId) references EMPLOYEE(Id).” This constraint implies that in every tuple of the EMPLOYEE relation—for example, (....., ...., 998877665)—the MngrId value 998877665 must occur in the Id field in this or some other tuple of the same relation.

**Example 3.2.2 (Foreign Keys for the Student Registration System).** The following is a set of foreign-key constraints that is appropriate for the schema of the fragment of our Student Registration System shown in Figure 3.6.

---

**TRANSCRIPT(StudId) references STUDENT(Id)**  
**TRANSCRIPT(CrsCode) references COURSE(CrsCode)**  
**TEACHING(ProfId) references PROFESSOR(Id)**  
**TEACHING(CrsCode) references COURSE(CrsCode)**  
**TRANSCRIPT(CrsCode,Semester) references TEACHING(CrsCode,Semester)**

---

These constraints illustrate several important points. First, the attributes used to cross-reference relations need not have the same name. For instance, the attribute StudId of TRANSCRIPT references a STUDENT attribute named Id, not StudId. The foreign-key constraint in the EMPLOYEE relation above is another example of the same phenomenon.

Second, a foreign key can consist of more than one attribute, as in the last constraint. In plain English, this constraint says that if a student took a course in a particular semester, there must be a professor who taught that course in that semester. ■

Not all referential constraints are foreign-key constraints. Indeed, contrary to a popular belief, professors do not teach empty classes—at least at some universities. In other words,

---

TEACHING(CrsCode, Semester)  
    references TRANSCRIPT(CrsCode, Semester)

---

3.1

appears to be an appropriate constraint for our database: at least one student must have taken (or be taking) the course named in each tuple of TEACHING. Hence, (3.1) is a referential integrity constraint, but is it a foreign-key constraint? The definition of foreign keys requires that the set of the *referenced* attributes must be a candidate key in the referenced relation.

The set  $\langle \text{CrsCode}, \text{Semester} \rangle$  is not a candidate key of TRANSCRIPT because, naturally, we must allow several students to take the same course in any given semester. Thus, (3.1) is a referential integrity constraint, but not a foreign-key constraint.

The above constraint is known as an **inclusion dependency** in the database theory. A foreign-key constraint is just a special kind of inclusion dependency, one where the referenced attribute set is a key. Unfortunately, because of the complexity of automatic enforcement of general inclusion dependencies, such dependencies are not part of SQL's DDL. However, they can be expressed using SQL's assertion mechanism. This is illustrated later in this chapter: the CREATE ASSERTION statement (3.4) is one possible representation of constraint (3.1).

**Semantic constraints.** Type, domain, key, and foreign-key constraints deal with the structure of the data. Other types of constraints might have little to do with structure but rather implement a business rule or convention in a particular enterprise. Such constraints are *semantic* because they are derived from the particular application domain being modeled by the database.

Some semantic constraints were mentioned earlier: the number of students registered for a course must not exceed the capacity of the classroom where the course is scheduled to meet; a student registered for a course must meet all the prerequisites for the course; no employee can earn more than the boss; and so forth.

As we will see later, SQL provides support for specifying a wide range of semantic constraints.

## 3.3 SQL—Data Definition Sublanguage

Having familiarized ourselves with the basic concepts of the relational model—tables and constraints—we are now ready to look at how these concepts are specified in the “real world”—the data definition sublanguage of SQL. We base our discussion on the SQL-92 standard, but be aware that most database vendors do not fully support this standard. Because most SQL manuals are hundreds of pages long (and the actual standard has several thousand pages), we discuss only the most salient points of the language. You will need a vendor-specific reference manual if you plan to undertake serious SQL projects.

Still, SQL-92 belongs to the past, while we are planning for the future. So, to prepare for the things to come, we will also be peeking into the new standards, SQL:1999 and SQL:2003. We will discuss other parts of SQL:1999 and SQL:2003 later in the book. The most significant of those are triggers in Section 7.3 and object-relational databases in Section 16.3. Some vendors are beginning to support parts of this standard in their latest releases.

Schemas are specified using the CREATE TABLE statement of SQL. This statement has a rich syntax, which we will introduce gradually. As a bare minimum, CREATE TABLE specifies the typing constraint: the name of a relation and the names of the attributes with their associated domains. However, the same statement can also specify primary and candidate keys, foreign-key constraints, and even certain semantic constraints.

### 3.3.1 Specifying the Relation Type

The type for the STUDENT relation is defined as follows.

---

```
CREATE TABLE STUDENT (
    Id          INTEGER,
    Name        CHAR(20),
    Address     CHAR(50),
    Status      CHAR(10) )
```

---

You should have no difficulty relating this SQL schema to earlier examples. Note that SQL allows the same symbolic name to be used for the name of a relation, an attribute, or even an attribute domain. Therefore, to distinguish the different parts of an SQL clause, we will be using different fonts for different syntactic categories.

### 3.3.2 The System Catalog

A DBMS must use information describing the structure of the database when it translates a statement of the DML into an executable program. It finds this information

COLUMNS	AttrName	RelName	Position	Format
	AttrName	Columns	1	CHAR(255)
	RelName	Columns	2	CHAR(255)
	Position	Columns	3	CHAR(255)
	Format	Columns	4	CHAR(255)
	CrsCode	Course	1	CHAR(6)
	DeptId	Course	2	CHAR(4)
	CrsName	Course	3	CHAR(20)
	Descr	Course	4	CHAR(100)
	Id	Student	1	INTEGER
	Name	Student	2	CHAR(20)
	Address	Student	3	CHAR(50)
	Status	Student	4	CHAR(10)
	...	...	...	...

FIGURE 3.8 Catalog relation.

in the **system catalog**. Therefore, while conceptually the purpose of the CREATE TABLE clause is to define a schema, technically this means inserting rows that describe the schema of a created table into the catalog. The catalog is a collection of special relations with their own schema. Figure 3.8 shows a table, called COLUMNS, which could be part of a catalog. Each row of COLUMNS contains information about a column in some database table, and all columns in all database tables are described in this way. The four columns of the table COURSE, for example, are described by four rows in the middle of the COLUMNS table.

Because COLUMNS is a table itself, its description must also be recorded. Rather than creating special machinery for this purpose, it is convenient to describe COLUMNS (and the other tables of the catalog) as a set of tuples in the catalog itself! Thus, the first few rows of COLUMNS describe the schema of that very relation. For example, the first row says that the first column of COLUMNS has the attribute name AttrName and the domain CHAR (255).

How then does all this referential complexity get bootstrapped? The catalog schema of a DBMS is designed by the vendor, and an instance of the catalog is created automatically whenever a new database is created by the database administrator.

### 3.3.3 Key Constraints

Primary keys and candidate keys are specified in SQL using two separate statements: PRIMARY KEY and UNIQUE. For instance, the schema of the table COURSE might look like this:

---

```
CREATE TABLE COURSE (
    GrsCode      CHAR(6),
    DeptId       CHAR(4),
    CrsName      CHAR(20),
    Descr        CHAR(100),
    PRIMARY KEY  (CrsCode),
    UNIQUE        (DeptId,CrsName) )
```

---

### 3.3.4 Dealing with Missing Information

Relations, as we have defined them, consist of tuples, which in turn are sequences of *known* values. For instance, in tuple (111111111, Doe John, 123 Main St., freshman), we have known values for Id, Name, Address, and so forth. In practice, the values of certain attributes might not be known. For example, when John Doe initially registers as a student we might not know his address. We might ask him to supply it as soon as possible, but we do not want to keep him out of our database until he complies. Instead, we use a placeholder, called **NULL**, and store it in place of the address until more information becomes available. Similarly, a tuple is entered in the TRANSCRIPT relation when a student registers for a course, but the Grade attribute for that tuple has no value until the semester completes.

The **NULL** placeholder is commonly referred to as a **null value**, but this is somewhat misleading because **NULL** is not a value—it indicates the *absence* of a “normal” value. In database theory and practice, **NULL** is treated as a special value that is a member of every attribute domain but is different from any other value in any domain. In fact, as we will see in Chapter 5, **NULL** is not even considered to be equal to itself!

In our example, null values arise because of a lack of information. In other situations, they arise by design. For instance, the attribute **MaidenName** is applicable to females but not to males. A database designer might decide that the schema

---

```
EMPLOYEE(Id:INT, Name:STRING, MaidenName:STRING)
```

---

is an appropriate description of a company’s employees. If such a relation includes tuples for male employees, those tuples will not and cannot have any value for the **MaidenName** attribute. Again, we can use **NULL** here.

As we will show later, null values often introduce additional problems, especially in query processing. For these reasons and others, it is sometimes desirable not to allow null values in certain sensitive places, such as the primary key. Indeed, how can we interpret a row of STUDENT of the form (NULL, Doe John, 123 Main St., freshman)? What if John Doe is sharing a room with a friend, also John Doe, who attends the same university and is a freshman? In this case, we might end up having two identical tuples in the same relation (each representing a different John Doe) and yet not being able to tell which John is represented by which tuple.

To preclude the above semantic difficulty, it is necessary to ensure that there is at least one key in each relation where null values are prohibited, and the primary key is

the logical choice for this. Thus, the SQL standard does not permit any attribute of a primary key to have a null value. In addition to the primary key, there may be other places where NULL is inappropriate. For instance, while it might be acceptable to temporarily allow NULL in the address field, a missing student name would certainly be a problem.

Although null values are not allowed in primary keys, they are allowed in other candidate keys (unless the database designer explicitly prohibits this). Note that a null in a candidate key does not violate the definition of a key. Indeed, as mentioned earlier, NULL is not equal to itself. Therefore, if a null value occurs in a candidate key of a tuple,  $t$ , no other tuple can agree with  $t$  on that candidate key.

In summary, database designers can deal with the null value problem by not allowing nulls in attributes that are deemed crucial to the semantic integrity of the database. We can, for instance, banish the nulls from the Name field as follows.

---

```
CREATE TABLE STUDENT (
    Id          INTEGER,
    Name        CHAR(20) NOT NULL,
    Address     CHAR(50),
    Status      CHAR(10) DEFAULT 'freshman',
    PRIMARY KEY (Id) )
```

---

In this example, null values are not allowed in the primary key, Id (which we do *not* need to specify explicitly), or in Name (which we *do* need to specify). One additional feature to note: The user can specify a *default* value for an attribute. This value will be automatically assigned to the attribute of a tuple should the tuple be inserted without this attribute being given a specific value.

### 3.3.5 Semantic Constraints

Semantic constraints are specified using the CHECK clause, whose basic syntax is

---

```
CHECK ( conditional expression )
```

---

The conditional expression can be any predicate or Boolean combination of predicates that can appear in the WHERE clause of an SQL statement. The integrity constraint is said to be violated if the conditional expression evaluates to false.

The CHECK clause is not used as a stand-alone statement: it is either attached to a CREATE TABLE statement, in which case it serves as an *intrarelational* constraint on that particular relation, or it can be attached to a CREATE ASSERTION statement, in which case it is an *interrelational* constraint.

**CHECK constraints in table definitions.** CHECK constraints attached to CREATE TABLE statements are generally used to impose conditions on the content of individual relations. The following example illustrates how the CHECK clause can limit the range of an attribute.

---

```
CREATE TABLE TRANSCRIPT (
    StudId   INTEGER,
    CrsCode  CHAR(6),
    Semester CHAR(6),
    Grade    CHAR(1),
    CHECK ( Grade IN ('A', 'B', 'C', 'D', 'F') ),
    CHECK ( StudId > 0 AND StudId < 1000000000 ) )
```

---

3.2

Restricting the applicable range of attributes is not the only use of the CHECK constraint in the above context. Using the somewhat contrived relation schema below, we can express the constraint that managers must always earn more than their subordinates.

---

```
CREATE TABLE EMPLOYEE (
    Id        INTEGER,
    Name      CHAR(20),
    Salary    INTEGER,
    MngrSalary INTEGER,
    CHECK ( MngrSalary > Salary ) )
```

---

The semantics of the CHECK clause inside the CREATE TABLE statement requires that *every tuple* in the corresponding relation satisfy all of the conditional expressions associated with all CHECK clauses in the corresponding CREATE TABLE statement.

One important consequence of this semantics is that the *empty relation*—a relation that contains no tuples—*always satisfies all CHECK constraints* as there are no tuples to check. This can lead to certain unexpected results. Consider the following syntactically correct schema definition.

---

```
CREATE TABLE EMPLOYEE (
    Id        INTEGER,
    Name      CHAR(20),
    Salary    INTEGER,
    DepartmentId CHAR(4),
    MngrId    INTEGER,
    CHECK ( 0 < (SELECT COUNT(*) FROM EMPLOYEE) ),
    CHECK ( (SELECT COUNT(*) FROM MANAGER)
            < (SELECT COUNT(*) FROM EMPLOYEE) ) )
```

---

3.3

Both CHECK clauses involve SELECT statements that count the number of rows in the named relation. Hence, the first CHECK clause presumably says that the EMPLOYEE relation cannot be empty. However natural this constraint may seem to be, it *does not* achieve its intended goal. Indeed, as we have remarked, this condition is supposed to be satisfied by *every tuple* in the EMPLOYEE relation, *not* by the relation

itself. Therefore, if the relation is empty, it satisfies every CHECK constraint, even the one that supposedly says that the relation must not be empty!

The second CHECK clause in (3.3) shows that in principle nothing stops us from trying to (mis)use this facility for interrelational constraints. We have assumed that there is a relation, MANAGER, that has a tuple for each manager in the company. The constraint presumably says that there must be more employees than managers, which it in fact does, but only if the EMPLOYEE relation is not empty.

**General constraints: ASSERTIONS.** Apart from the subtle bug, the second constraint in (3.3) looks particularly unintuitive because it is symmetric by nature and yet it is asymmetrically hardwired into the table definition of just one of the two relations involved. To overcome this problem, SQL provides one more way to use the CHECK clause—inside the CREATE ASSERTION statement. An assertion is a component of the database schema, like a table, so incorporating the CHECK clause within it puts the constraint in a symmetric relationship with the two tables. Thus, the two constraints can be restated as follows (and this time correctly!):

---

```
CREATE ASSERTION THOUShALTNOTFIREEVERYONE
    CHECK ( 0 < (SELECT COUNT(*) FROM EMPLOYEE) )
CREATE ASSERTION WATCHADMINCOSTS
    CHECK ( (SELECT COUNT(*) FROM MANAGER)
            < (SELECT COUNT(*) FROM EMPLOYEE) )
```

---

Unlike the CHECK conditions that appear inside a table definition, those in the CREATE ASSERTION statement must be satisfied by the contents of the entire database rather than by individual tuples of a host table. Thus, a database satisfies the first assertion (above) if and only if the number of tuples in the EMPLOYEE relation is greater than zero. Likewise, the second assertion is satisfied whenever the MANAGER relation has fewer tuples than the EMPLOYEE relation has.

For another example of the use of assertions, suppose that the salary information about managers and employees is kept in different relations. We can then state our rule about who should earn more using the following assertion, which literally says that there must not exist an employee who has a boss who earns less. For the sake of this example, we assume that the MANAGER relation has the attributes Id and Salary.

---

```
CREATE ASSERTION THOUShALTNOTOUTEARNYOURBOSS
    CHECK ( NOT EXISTS
        (SELECT * FROM EMPLOYEE, MANAGER
         WHERE EMPLOYEE.Salary > MANAGER.Salary
           AND EMPLOYEE.MngrId = MANAGER.Id ))
```

---

An interesting question now is, what if, at the time of specifying the constraint THOUShALTNOTFIREEVERYONE, the EMPLOYEE relation is empty? And what if, at the

time of specifying THOU SHALT NOT OUT EARN YOUR BOSS, there already is an employee who earns more than the boss? The SQL standard states that if a new constraint is defined and the existing database does not satisfy it, the constraint is *rejected*. The database designer then has to find out the cause of constraint violation and either amend the constraint or rectify the database.

Our last example is a little more complex.<sup>3</sup> It shows how assertions can be used to specify inclusion dependencies that are not foreign-key constraints. More specifically, we express the inclusion dependency (3.1) on page 45 using the assertion statement of SQL.

---

```
CREATE ASSERTION COURSESSHALLNOTBEEMPTY
  CHECK (NOT EXISTS (
    SELECT * FROM TEACHING
    WHERE NOT EXISTS (
      SELECT * FROM TRANSCRIPT
      WHERE Teaching.CrsCode = Transcript.CrsCode
        AND Teaching.Semester = Transcript.Semester)))
```

---

3.4

The CHECK constraint here verifies that there is no tuple in the TEACHING relation (the outer NOT EXISTS statement) for which no matching class exists in the TRANSCRIPT relation (the inner NOT EXISTS statement). A tuple in the TEACHING relation refers to the same class as does a tuple in the TRANSCRIPT relation if in both tuples the CrsCode and Semester components are equal. This test is performed in the innermost WHERE clause.

Different assertions have different maintenance costs (the time required for the DBMS to check that the assertion is satisfied). Generally, intrarelational constraints come cheaper than do interrelational constraints. Among the interrelational constraints, those that are based on keys are easier to enforce than those that are not. Thus, for instance, foreign-key constraints come cheaper than do general inclusion dependencies, such as (3.4).

The automatic checking of integrity constraints by a DBMS is one of the more powerful features of SQL. It not only protects the database from errors that might be introduced by untrustworthy users (or sloppy application programmers) but can simplify access to the database as well. For example, a primary key constraint ensures that at most one tuple containing a particular primary key value exists in a table. If a DBMS did not automatically check this constraint, an application program attempting to insert a new tuple or to update the key attributes of an existing tuple would have to scan the table first to ensure that the primary key constraint is maintained.

<sup>3</sup> It involves the use of a nested, correlated subquery. If you do not understand (3.4), plan to come back here after reading Chapter 5.

### 3.3.6 User-Defined Domains

We have already seen how the CHECK clause lets us limit the range of the attributes in a table. SQL provides an alternative way to enforce such constraints by allowing the user to define appropriate ranges of values, give them domain names, and then use these names in various table definitions. This approach makes the design more modular. We could, for example, create the domain GRADES and use it in the TRANSCRIPT relation instead of using the CHECK constraint directly in the definition of that relation.

---

```
CREATE DOMAIN GRADES CHAR(1)
    CHECK ( VALUE IN ('A', 'B', 'C', 'D', 'F', 'I') )
```

---

The only difference between this and the previous constraint (3.2) on page 50, which was directly imposed on the table STUDENT, is that here we use a special keyword, VALUE, instead of the attribute name—we cannot use attribute names here, because the domain is not attached to any particular table. Now we can add

---

```
Grade GRADES
```

---

to the definition of STUDENT. The overall effect is the same, but we can use this predefined domain name in several tables without having to repeat the definition. At a later time, if we need to change this domain definition, the change will automatically propagate to all the tables that use that domain. A domain is a component of the database schema, like a table or an assertion.

Note that, as with assertions, we can use complex queries to define fairly nontrivial domains.

---

```
CREATE DOMAIN UPPERDIVISIONSTUDENT INTEGER
    CHECK ( VALUE IN (SELECT Id FROM STUDENT
                      WHERE Status IN ('senior', 'junior')
                        AND VALUE IS NOT NULL ) )
```

---

The domain UPPERDIVISIONSTUDENT consists of student Ids that belong to students whose status is either senior or junior. In addition, the last clause excludes NULL from that domain. Observe that, in order to verify that the constraint imposed by this domain is satisfied, a query against the database is run. Since such queries might be quite expensive, not every vendor supports the creation of such “virtual” domains.

### 3.3.7 Foreign-Key Constraints

SQL provides a simple and natural way of specifying foreign keys. The following statement makes CrsCode a foreign key referencing COURSE and makes ProfId a foreign key referencing the PROFESSOR relation.

---

```
CREATE TABLE TEACHING (
    ProfId   INTEGER,
    CrsCode  CHAR(6),
    Semester CHAR(6),
    PRIMARY KEY (CrsCode, Semester),
    FOREIGN KEY (CrsCode) REFERENCES COURSE,
    FOREIGN KEY (ProfId) REFERENCES PROFESSOR (Id) )
```

---

If the names of the referring and the referenced attributes are the same, the referenced attribute can be omitted. The attribute CrsCode above is an example of this situation. If the referenced attribute has a different name than that of the referring attribute, both attributes must be specified. The term PROFESSOR (Id) in the second FOREIGN KEY clause shows how this is done.

It should be noted that, although the SQL standard does not require that the referenced attributes form a *primary key* (they can form *any candidate key*), some database vendors impose the primary key restriction.

In the above example, whenever a TEACHING tuple has a course code in it, the actual course record with this course code must exist in the COURSE relation. Similarly, the professor's Id in a TEACHING tuple must reference an existing tuple in the PROFESSOR relation. The DBMS is expected to enforce these constraints automatically once they are specified. Thus, as part of the procedure for deleting a tuple in the PROFESSOR relation, a check is made to ensure that there is no corresponding tuple in the TEACHING relation.

**Foreign keys and nulls.** What if, in a particular tuple, the value of an attribute in a foreign key is NULL? Should we insist that there be a corresponding tuple in the referenced relation with a null value in a key attribute? Not a good idea, especially if the referenced key is a primary key. Therefore, SQL *relaxes the foreign-key constraint* by letting foreign keys have null values. In this case there need not be a corresponding tuple in the referenced relation.

**Chicken-and-egg problems.** Foreign-key constraints raise other subtle issues too. Consider the table EMPLOYEE defined in (3.3). Suppose that we also have a table that describes departments.

---

```
CREATE TABLE DEPARTMENT (
    DeptId   CHAR(4),
    Name     CHAR(40),
    Budget   INTEGER,
    MngrId   INTEGER,
    FOREIGN KEY (MngrId) REFERENCES EMPLOYEE (Id) )
```

---

Now, if we look back at the `DepartmentId` attribute of the `EMPLOYEE` table, it is clear that this attribute is intended to represent valid department IDs (i.e., IDs of the departments stored in the `DEPARTMENT` relation). In other words, the constraint

---

```
FOREIGN KEY (DepartmentId) REFERENCES DEPARTMENT (DeptId)
```

---

is in order as part of the `CREATE TABLE EMPLOYEE` statement.

The problem is that either `EMPLOYEE` or `DEPARTMENT` has to be defined first. If `EMPLOYEE` comes first, we cannot have the above foreign-key constraint in the `CREATE TABLE EMPLOYEE` statement because it refers to the yet-to-be-defined table `DEPARTMENT`. If `DEPARTMENT` is defined before `EMPLOYEE`, the DBMS will issue an error trying to process the foreign-key constraint in the `CREATE TABLE DEPARTMENT` statement because this constraint references the yet-to-be-defined table `EMPLOYEE`. We are facing a chicken-and-egg problem.

The solution is to *postpone* the introduction of the foreign-key constraint in the first table. That is, if `CREATE TABLE EMPLOYEE` is executed first, we should not have the `FOREIGN KEY` clause in it. However, after `CREATE TABLE DEPARTMENT` has been processed, we can *add* the desired constraint to `EMPLOYEE` using the `ALTER TABLE` directive. This directive will be described in detail later in this section. Here we give only the final result.

---

```
ALTER TABLE EMPLOYEE
ADD CONSTRAINT EMPDEPTCONSTR
FOREIGN KEY (DepartmentId) REFERENCES DEPARTMENT (DeptId)
```

---

If, after settling this circular reference problem, we now want to start populating the database, we are in for another surprise. Suppose that we want to put the first tuple, `(00000007, James Bond, 7000000, B007, 000000000)`, into the `EMPLOYEE` relation. Since at this moment the `DEPARTMENT` table is empty, the foreign-key constraint that prescribes that `B007` must refer to a valid tuple in the `DEPARTMENT` relation is violated.

One solution is to initially replace the `DepartmentId` component in all tuples in the `EMPLOYEE` relation with `NULL`. Then, when `DEPARTMENT` is populated with appropriate tuples, we can scan the `EMPLOYEE` relation and replace the null values with valid department IDs. However, this solution is awkward and error-prone. A better solution is to use a transaction and deferred checking of integrity constraints.

In Chapter 2, we pointed out that the intermediate states of the database produced by a transaction might be inconsistent—they might temporarily violate integrity constraints. The only important thing is that constraints must be preserved when the transaction commits. To accommodate the possibility of temporary constraint violations, SQL allows the programmer to specify the mode of a particular integrity constraint to be either `IMMEDIATE`, in which case a check is made after each SQL statement that changes the database, or `DEFERRED`, in which case a check is

made only when a transaction commits. Then, to deal with the circular reference problem just described, we can

1. Declare the foreign-key constraints in the two tables, EMPLOYEE and DEPARTMENT, as INITIALLY DEFERRED to set the initial mode of constraint checking.
2. Make the updates that populate these tables part of the same transaction. This will allow the intermediate states to be temporarily inconsistent.
3. Make sure that when all updates are done, the foreign-key constraints are satisfied. Otherwise, the transaction will be aborted when it terminates.

The full details of how transactions are defined in SQL and how they interact with constraints will be discussed in Chapter 8.

### 3.3.8 Reactive Constraints

When a constraint is violated, the corresponding transaction is typically aborted. However, in some cases, other remedial actions are more appropriate. Foreign-key constraints are one example of this situation.

Suppose that a tuple  $\langle 007007007, \text{MGT123}, \text{F1994} \rangle$  is inserted into the TEACHING relation. Because the table PROFESSOR does not have a professor with the Id 007007007, this insertion violates the foreign-key constraint that requires all non-NULL values in the ProfId field of TEACHING to reference existing professors. In such a case, the semantics of SQL is very simple: the insertion is rejected.

When constraint violation occurs because of deletion of a referenced tuple, SQL offers more choices. Consider the tuple  $t = \langle 009406321, \text{MGT123}, \text{F1994} \rangle$  in the table TEACHING. According to Figure 3.5,  $t$  references Professor Taylor in the PROFESSOR relation, and the course Market Analysis in the COURSE relation. Suppose that Professor Taylor leaves the university. What should happen to  $t$ ? One solution is to temporarily set the value of ProfId in  $t$  to NULL until a replacement lecturer is found. Another solution is to have the attempt to delete Professor Taylor's tuple from the PROFESSOR relation fail, which might reflect the policy that professors are not allowed to leave in the middle of a semester. Finally, if Professor Taylor is the only faculty member capable of teaching the course, we might remove MGT123 from the curriculum altogether. By deleting the referencing tuple,  $t$ , the violation of referential integrity is resolved.

These possibilities can be rephrased as **reactive constraints**. A reactive constraint is a static constraint coupled with a specification of *what to do* if a certain event happens. For instance, the first alternative above is a constraint that requires that whenever a PROFESSOR tuple is deleted, the field ProfId of all the referencing tuples in TEACHING must be set to NULL. The second alternative is a constraint that asserts that if a referencing tuple exists it cannot be deleted. The third alternative asserts that all referencing tuples are deleted when the referenced tuple is deleted.

We can specify the appropriate response to an event using **triggers**, which are statements of the form

---

**WHENEVER event DO action**

---

**Triggers attached to foreign-key constraints.** SQL supports a special kind of triggers, which are attached to foreign-key constraints. These triggers are specified as part of the FOREIGN KEY clause using the options ON DELETE and ON UPDATE, which indicate what to do if a referenced tuple is deleted or updated. To illustrate, let us revisit the definition of TEACHING.

---

```
CREATE TABLE TEACHING (
    ProfId INTEGER,
    CrsCode CHAR(6),
    Semester CHAR(6),
    PRIMARY KEY (CrsCode, Semester),
    FOREIGN KEY (ProfId) REFERENCES PROFESSOR(Id)
        ON DELETE NO ACTION
        ON UPDATE CASCADE,
    FOREIGN KEY (CrsCode) REFERENCES COURSE (CrsCode)
        ON DELETE SET NULL
        ON UPDATE CASCADE )
```

---

Here we have specified four triggers. One is **fired** (i.e., executed) whenever a PROFESSOR tuple is deleted, one whenever a PROFESSOR tuple is modified, one when a COURSE tuple is deleted, and one when a COURSE tuple is modified. The clause ON DELETE NO ACTION means that any attempt to remove a PROFESSOR tuple must be rejected outright if the professor is referenced by a TEACHING tuple. NO ACTION is the default situation when an ON DELETE or ON UPDATE clause is not specified. The clause ON UPDATE CASCADE means that if the Id number of a PROFESSOR tuple is changed, the change must be propagated to all referencing TEACHING tuples (i.e., the new Id must be stored in the referencing tuples). Hence, the same professor is recorded as teaching the course. (Similarly, a specification ON DELETE CASCADE causes the referencing tuple to be deleted.) ON DELETE SET NULL tells the DBMS that if a COURSE tuple is removed and there is a referencing TEACHING tuple, the referencing attribute, CrsCode, in that tuple must be set to NULL. Alternatively, the designer can specify SET DEFAULT (instead of SET NULL): if CrsCode was defined with a DEFAULT option (e.g., the Status attribute in the STUDENT relation), then it will be reset to its default value if the referenced tuple is deleted; otherwise, it will be set to NULL (which is the default value for the DEFAULT option).

Any combination of DELETE or UPDATE triggers with NO ACTION, CASCADE, or SET NULL/DEFAULT options is allowed in foreign-key triggers. The action taken to repair a foreign-key violation in one table,  $T_2$ , in response to a change in another

table,  $T_1$ , (e.g., delete a row in  $T_2$  if a row is deleted in  $T_1$ ) might cause a violation of a foreign-key constraint in  $T_3$  that refers to  $T_2$ . The action specified in  $T_3$  controls how that violation is handled. If the entire chain of violations cannot be resolved (e.g., the action specified in  $T_3$  is NO ACTION), the initial deletion from  $T_1$  is rejected.

**General triggers.** The ON DELETE/UPDATE triggers are simple and powerful, but they are not powerful enough to capture a wide variety of constraint violations that arise in database applications and are not due to foreign keys. For instance, the referential integrity constraint (3.1) on page 45 is *not* a foreign-key constraint and yet the same problems arise here when tuples of the TRANSCRIPT relation are modified or deleted. More importantly, foreign-key triggers cannot even begin to address common needs such as preventing salaries from changing by more than 5% in the same transaction.

To handle these needs, all major database vendors took destiny into their own hands and retrofitted their products with trigger mechanisms. Interestingly, the original design of SQL—before there was an SQL-92 standard—did have relatively powerful triggers. Triggers reappeared in SQL with the SQL:1999 standard, but some vendors are yet to align their offerings with the new standard. We will briefly describe the general trigger mechanism here and leave the details to Chapter 7.

The basic idea behind triggers is simple: whenever a specified *event* occurs, execute some specified *action*. Consider the following simple trigger defined using the syntax of SQL:1999. The trigger fires whenever CrsCode or Semester is changed in a tuple in the TRANSCRIPT relation. When the trigger fires and the grade recorded for the course is not NULL, an exception is raised and the changes made by the transaction are rolled back. Otherwise (if the grade is NULL), we interpret the change as a student dropping one course in favor of another, so the trigger does nothing and the change is allowed to take hold. This trigger is created with the statement

---

```
CREATE TRIGGER CRSCHANGETRIGGER
  AFTER UPDATE OF CrsCode, Semester ON TRANSCRIPT
  WHEN ( Grade IS NOT NULL )
    ROLLBACK
```

---

This definition is self-explanatory except, perhaps, for the WHEN clause, which acts as a guard, that is, as a precondition that must be satisfied in order for the trigger to fire. If the precondition is true, the statements following WHEN are executed. In our case, the statement aborts the transaction.

In general, many more details might need to be specified in order to define a trigger. For instance, should the action be executed just before the triggering update is applied to the database or after it? Should this action be executed immediately after the event or at some later time? Can a triggered action trigger another action? Moreover, to specify the guard in the WHEN clause, we might need to refer to both the *old* and the *new* values of the modified tuples (e.g., to check that salaries have not been changed by more than 5%). We postpone the discussion of these issues

until Chapter 7, where many more examples of triggers will be given. In particular, we will discuss how general triggers can be used to maintain inclusion dependencies in the presence of updates (analogous to how ON DELETE and ON UPDATE triggers are used to maintain foreign-key constraints).

### 3.3.9 Database Views

In Section 3.1, we discussed the three levels of abstraction in databases: the physical level, the conceptual level, and the external level. We have already shown how the conceptual layer is defined in SQL. We now discuss the external (or view) layer of SQL. The physical layer will be discussed in detail in Chapter 9.

In SQL, the external schema is defined using the CREATE VIEW statement. In many respects, a view is like an ordinary table: you can query it, modify it, or control access to it. However, in several important ways a view is not a table. For one thing, the rows of a view are derived from tables (and other views) of the database. Thus, in reality a view repackages information stored elsewhere. Furthermore, the contents of a view do not physically exist in the database. Instead, a recipe for *constructing* the contents on the fly from other database tables is stored in the system catalog. As will be seen shortly, the view definition is a hybrid of the CREATE TABLE statement and the SELECT statement introduced in Chapter 2. Because of this, views are often called **virtual tables**.

To illustrate, consider the following view, which tells which professors have taught which students (a professor is said to have taught a student if the student took a course in the semester in which the professor offered it).

---

```
CREATE VIEW PROFSTUD (Prof, Stud) AS
  SELECT TEACHING.ProfId, TRANSCRIPT.StudId
  FROM TRANSCRIPT, TEACHING
  WHERE TRANSCRIPT.CrsCode = TEACHING.CrsCode
    AND TRANSCRIPT.Semester = TEACHING.Semester
```

---

3.5

The first line defines the name of the view and its attributes. The rest is just an SQL query that tells how to obtain the contents of the view. These contents, with respect to the database instance of Figure 3.5, are shown in Figure 3.9. To help you understand where the tuples in the view come from, each tuple is annotated with a “justification.” (A justification for a tuple  $(p, s)$  is a course code together with the semester in which student  $s$  took that course from professor  $p$ .)

The view PROFSTUD might be part of the external schema that helps the university keep in touch with its alumni since establishing the relationship between students and professors through courses might be an important and frequent operation in such an application. So, instead of this relationship being reinvented by every single application, it can be defined once and for all in the form of a view. Once it is defined, all applications can refer to the view as if it were an ordinary table. The rows of the view are constructed at the time it is accessed, so the contents change as the underlying relations are updated by transactions.

PROFSTUD	Prof	Stud	Justification
	009406321	666666666	MGT123,F1994
	121232343	666666666	EE101,S1991
	900120450	666666666	MAT123,F1997
	555666777	987654321	CS305,F1995
	009406321	987654321	MGT123,F1994
	101202303	123454321	CS315,S1997; CS305,S1996
	900120450	123454321	MAT123,S1996
	121232343	023456789	EE101,F1995
	101202303	023456789	CS305,S1996
	900120450	111111111	MAT123,F1997
	009406321	111111111	MGT123,F1997
	783432188	111111111	MGT123,F1997

**FIGURE 3.9** Contents of the view defined by SQL statement (3.5).

In Chapter 5, we will expand our discussion of the view mechanism and show how views can be used to modularize the construction of complex queries. The authorization mechanism is another important use of views. In Section 3.3.12, we will see that views can be treated as ordinary tables for the purpose of granting selective access rights to the information stored in the database.

### 3.3.10 Modifying Existing Definitions

Although database schemas are not supposed to change frequently, they do evolve. Occasionally, new fields are added to relations or existing fields are dropped; new constraints and domains are created, or old ones become invalid (perhaps because business rules change). Of course, we can always copy the old contents of a relation to a temporary space, erase the old relation and its schema, and then create a new relation schema with the old name. However, this process is tedious and error-prone. To simplify schema maintenance, SQL provides the **ALTER** statement, which in its simplest form looks like this.

---

```
ALTER TABLE STUDENT
ADD COLUMN Gpa INTEGER DEFAULT 0
```

---

This command adds a new field to the STUDENT relation and initializes the field's value in each tuple to 0. You can also use **DROP COLUMN** to remove a column from a relation and add or drop constraints. For instance,

---

```
ALTER TABLE STUDENT
    ADD CONSTRAINT GPARANGE CHECK (Gpa >= 0 AND Gpa <= 4)
ALTER TABLE TEACHING
    ADD CONSTRAINT TEACHKEY UNIQUE(ProfId, Semester, Time)
```

---

If the current instance of STUDENT violates the new constraint GPARANGE, or if TEACHING violates TEACHKEY, the newly added constraints are rejected.

In order for a constraint to be “droppable” from a table definition, the constraint must be named at the time when it is defined—an option we have not used until now. We make up for this by naming every constraint in a revised definition of TRANSCRIPT.

---

```
CREATE TABLE TRANSCRIPT (
    StudId      INTEGER,
    CrsCode     CHAR(6),
    Semester    CHAR(6),
    Grade       GRADES,
    CONSTRAINT TrKEY PRIMARY KEY (StudId, CrsCode, Semester),
    CONSTRAINT STUDFK FOREIGN KEY (StudId) REFERENCES STUDENT,
    CONSTRAINT CRSFK FOREIGN KEY (CrsCode) REFERENCES COURSE,
    CONSTRAINT IdRANGE CHECK ( StudId > 0 AND
                                StudId < 1000000000 ))
```

---

Now we can alter the above definition by dropping any one of the specified integrity constraints. For example,

---

```
ALTER TABLE TRANSCRIPT DROP CONSTRAINT TrKEY
```

---

When a table is no longer needed, its definition can be erased from the catalog. In this case, the schema of the table and its instance are *both* lost. Previously defined assertions and domains can also be dropped. For example,

---

```
DROP TABLE EMPLOYEE RESTRICT
DROP ASSERTION THOUShALTNOTFIREEVERYONE
DROP DOMAIN GRADES
```

---

**Brain Teaser:** What should happen to the foreign key CrsCode in TRANSCRIPT if we drop COURSE?

The DROP TABLE command has two options: RESTRICT and CASCADE. With the RESTRICT option, the DROP statement would refuse to delete a table if it is used in some other definition, such as integrity constraint. For instance, the constraint

THOU SHALL NOT FIRE EVERYONE would prevent deletion of EMPLOYEE in the above case. The CASCADE option, in contrast, deletes a table definition along with any other definition that uses this table. So, for example, if we used CASCADE in the above DROP TABLE command, the assertion THOU SHALL NOT FIRE EVERYONE would be deleted along with the EMPLOYEE table (and with the constraint WATCHADMIN-COSTS). In this case, the above DROP ASSERTION statement would be redundant.

The DROP DOMAIN command has its own quirks. For instance, deleting the domain GRADES above will *not* leave the attribute Grade of TRANSCRIPT in limbo. Instead, the CHECK clause that defines GRADES is copied over and is attached to all tables where this domain is used. Only after the orphaned attributes are taken care of will the GRADE domain be erased from the system catalog.

*Brain Teaser:* Can a relation instance become invalid after execution of an ADD CONSTRAINT statement? What about DROP CONSTRAINT? DROP DOMAIN?

### 3.3.11 SQL-Schemas

The structure of a database is described in the system catalog. A catalog is SQL's version of a directory, in which elements are schema objects, such as tables and domains. Thus, for example, Figure 3.8 on page 47 is a simplified version of a part of the catalog for the Student Registration System. SQL partitions the catalog into SQL-schemas. An **SQL-schema**<sup>4</sup> is a description of a portion of a database that is under the control of a single user who has the authorization to create and access the objects within it. For example,

---

```
CREATE SCHEMA SRS_STUDINFO AUTHORIZATION JohnDoe
```

---

creates the SQL-schema SRS\_STUDINFO describing the part of the Student Registration System database that contains information about students. The AUTHORIZATION clause specifies the user (JohnDoe in our case) who controls the permissions for accessing tables and other objects defined in that SQL-schema.

The naming mechanism used in conjunction with SQL-schemas is similar to that used for directories in operating systems. For example, if JohnDoe wants to create a STUDENT table in the SRS\_STUDINFO SQL-schema, he refers to it as SRS\_STUDINFO.STUDENT. To refer to the STUDFK constraint, he uses SRS\_STUDINFO.STUDFK. Thus, an SQL-schema also serves as a kind of namespace mechanism that allows use of the same name for different relations (domains, constraints, etc.) by putting them under the scope of different schemas.

As with every CREATE statement, there is a matching DROP SCHEMA statement. For instance, JohnDoe can delete the above schema (and the entire portion of the database under it) using the following statement:

<sup>4</sup> Note that this use of the word "schema" is different than our previous use to describe a relation schema or database schema. We use the term "SQL-schema" to refer to the SQL usage.

---

```
DROP SCHEMA SRS_STUDINFO
```

---

SQL does not specify the format in which the information in an SQL-schema must be stored, but it does require that each system catalog contains one particular schema, named INFORMATION\_SCHEMA, whose contents are precisely specified. INFORMATION\_SCHEMA contains a set of SQL tables that repeat, in a precisely defined way, all the definitions from all other SQL-schemas in the catalog. The information in INFORMATION\_SCHEMA can be accessed by any authorized user.

Finally, we note that SQL defines a *cluster* as a set of catalogs. A cluster describes the set of databases that can be accessed by a single SQL program. Thus, our university might define a cluster describing all of the databases it maintains that can be accessed by a single transaction.

### 3.3.12 Access Control

Databases often contain sensitive information. Therefore, the system must ensure that only those authenticated users who are authorized to access the database are allowed to and that they are only allowed to access information that has been specifically made available to them. Many transaction processing systems provide extensive authentication and authorization mechanisms. Authentication occurs prior to access. It might be the result of providing a password to the DBMS, or it can be a more elaborate scheme involving a separate security server. In any case, once authentication has been completed, the user is assumed to be (correctly) associated with an *authorization Id* and access to the database can begin. In SQL, *authorization Ids* are tokens that denote sets of privileges. Several database users can have the same *authorization Id*, in which case they would all have the same privileges.

The creator of a table or other object is assumed to own that object and has all privileges with respect to it. The owner can grant other users certain specific privileges with respect to that object by using the GRANT statement

---

```
GRANT { privilege-list | ALL PRIVILEGES }
    ON object
    TO { user-list | PUBLIC } [ WITH GRANT OPTION ]
```

---

where WITH GRANT OPTION means that the recipient can subsequently grant to others the privileges she has been granted.

If the object is a table or a view, *privilege-list* can include

---

```
SELECT
DELETE
INSERT [(column-comma-list)]
UPDATE [(column-comma-list)]
REFERENCES [(column-comma-list)]
```

---

The first four options grant the privilege of performing the specified statement. The options that include (*column-comma-list*) grant the privilege only for the specified columns. For example, if the INSERT privilege has been granted, only the values of the attributes named in *comma-list* can be specified in the inserted tuple. All *comma-lists* are optional, as indicated by the square brackets.

REFERENCES grants the privilege of referring to the table or column using a foreign key. It might seem strange to control this type of access, but security is not complete if foreign-key constraints are not controlled. Two problems arise if foreign-key constraints can be set up arbitrarily. Suppose that a student is permitted to create the table

---

```
CREATE TABLE DONTDISMISSME (
    Id INTEGER,
    FOREIGN KEY (Id) REFERENCES STUDENT)
```

---

If she inserts a single row in DONTDISMISSME containing her Id, the registrar will not be able to dismiss the student—that is, delete the student's row from STUDENT—because a deletion would cause a violation of referential integrity and hence be rejected by the DBMS.

Unrestricted access to foreign keys can also create certain information leaks. Suppose that, in the interest of protecting student information, SELECT access to STUDENT is granted only to university employees in the registrar's office. If however, an intruder were allowed to create the above table (perhaps with the name PROBEPROTECTEDINFO), this restriction could be circumvented. If the intruder inserted the Id of a particular individual in the table and the insertion were permitted by the DBMS, the intruder could conclude that a row for that individual existed in STUDENT (since referential integrity would otherwise be violated). Similarly, if the insertion were denied, the intruder could conclude that the individual was not a student. Hence, even though the intruder did not have permission to access the table through a SELECT statement, he was able to extract some information.

**Example 3.3.1 (Grant Statement).** The following GRANT statement gives John Smyth and Mary Doe the permission to read a row and to update the ProfId column of the TEACHING relation.

---

```
GRANT SELECT, UPDATE (ProfId) ON STUDREGSYSTEM.TEACHING
    TO JohnSmyth, MaryDoe WITH GRANT OPTION
```

---

It also gives them permission to pass on the same privileges to other users. Note, however, that these users are not allowed to delete tuples from that relation. Nor can they change other columns. However, they can see the information stored in all columns of the relation. ■

**Example 3.3.2 (Authorization through Views).** SQL allows control of not only direct access to databases but also indirect access through views. For instance, the

following statement gives all users who are classified as alumni unrestricted query access to the PROFSTUD view defined in statement (3.5) on page 59.

---

**GRANT SELECT ON PROFSTUD TO Alumnus**

---

However, this GRANT statement does not permit the alumni to pass their query rights to others, and they cannot update the view. What is more interesting is that these users do not even have the rights to access TRANSCRIPT and TEACHING—the two relations that supply the contents for the PROFSTUD view. Their access is *indirect* and only to the parts of these relations that are visible through the view. ■

It would have been convenient to use views to selectively grant UPDATE, INSERT, or DELETE privileges. For example, one might want to grant prof\_smith the right to update the Grade column of the TRANSCRIPT table, but only on the rows corresponding to courses he has taught. Unfortunately, not every view is updatable, and so it is not always possible to use views in this way. Chapter 5 will have further discussion on this subject.

Privileges can also be granted for objects other than tables (for example, domains). We omit the details.

Privileges, or the grant option for privileges, can be revoked using the REVOKE statement.

---

**REVOKE [ GRANT OPTION FOR ] *privilege-list*  
ON *object*  
FROM *user-list* {CASCADE | RESTRICT}**

---

CASCADE means that if some user,  $U_1$ , whose user name appears on the list *user-list*, has granted those privileges to another user,  $U_2$ , the privileges granted to  $U_2$  are also revoked. If  $U_2$  has granted those privileges to still another user, those privileges are revoked as well, and so on. The option RESTRICT means that if any such dependent privileges exist, the REVOKE statement is rejected.

In many applications, granting privileges at the level of database operations, such as SELECT or UPDATE, is not adequate. For example, only a depositor can deposit in a bank account and only a bank official can add interest to the account, but both the deposit and interest transactions might use the same UPDATE statement. For such applications, it is more appropriate to grant privileges at the level of subroutines or transactions. Many transaction processing systems control access at this level.

## BIBLIOGRAPHIC NOTES

The relational data model was introduced in [Codd 1970, 1990]. Later on [Codd 1979] proposed various extensions to the original model in order to capture more semantic information.

These ideas were extended and implemented in the two pioneering relational systems: System R [Astrahan et al. 1981] and INGRES [Stonebraker 1986]. Eventually, System R became DB2, a commercial product from IBM, and INGRES became a commercial product under the same name (currently sold by Computer Associates, Intl.).

A rich body of theory has been developed for relational databases, much of which found its way into research prototypes and commercial products. More in-depth discussion as well as additional topics not covered in this book can be found in [Maier 1983; Atzeni and Ántonellis 1993; Abiteboul et al. 1995].

## EXERCISES

- 3.1 Define data atomicity as it relates to the definition of relational databases. Contrast data atomicity with transaction atomicity as used in a transaction processing system.
- 3.2 Prove that every relation has a key.
- 3.3 Define the following concepts:
  - a. Key
  - b. Candidate key
  - c. Primary key
  - d. Superkey
- 3.4 Define
  - a. Integrity constraint
  - b. Static, as compared with dynamic, integrity constraint
  - c. Referential integrity
  - d. Reactive constraint
  - e. Inclusion dependency
  - f. Foreign-key constraint
- 3.5 Looking at the data that happens to be stored in the tables for a particular application at some particular time, explain whether or not you can tell
  - a. What the key constraints for the tables are
  - b. Whether or not a particular attribute forms a key for a particular table
  - c. What the integrity constraints for the application are
  - d. Whether or not a particular set of integrity constraints is satisfied
- 3.6 We state in the book that once constraints have been specified in the schema, it is the responsibility of the DBMS to make sure that they are not violated by the execution of any transactions. SQL allows the application to control when each constraint is checked. If a constraint is in *immediate mode*, it is checked immediately after the execution of any SQL statement in a transaction that might make it false. If it is in *deferred mode*, it is not checked until the transaction requests to commit. Give an example where it is necessary for a constraint to be in deferred mode.
- 3.7 Suppose we do not require that all attributes in the primary key are non-null and instead request that, in every tuple, at least one key (primary or candidate) does

- not have nulls in it. (Tuples can have nulls in other places and the non-null key can be different for different tuples.) Give an example of a relational instance that has two distinct tuples that *might* become one once the values for all nulls become known (that is, are replaced with real values). Explain why this is not possible when one key (such as the primary key) is designated to be non-null for all tuples in the relation.
- 3.8 Use SQL DDL to specify the schema of the Student Registration System fragment shown in Figure 3.4, including the constraints in Figure 3.6 and Example 3.2.2. Specify SQL domains for attributes with small numbers of values, such as `DeptId` and `Grade`.
- 3.9 Consider a database schema with four relations: `SUPPLIER`, `PRODUCT`, `CUSTOMER`, and `CONTRACTS`. Both the `SUPPLIER` and the `CUSTOMER` relations have the attributes `Id`, `Name`, and `Address`. An `Id` is a nine-digit number. `PRODUCT` has `PartNumber` (an integer between 1 and 999999) and `Name`. Each tuple in the `CONTRACTS` relation corresponds to a contract between a supplier and a customer for a specific product in a certain quantity for a given price.
- a. Use SQL DDL to specify the schema of these relations, including the appropriate integrity constraints (primary, candidate, and foreign key) and SQL domains.
  - b. Specify the following constraint as an SQL assertion: *there must be more contracts than suppliers*.
- 3.10 You have been hired by a video store to create a database for tracking DVDs and videocassettes, customers, and who rented what. The database includes these relations: `RENTALITEM`, `CUSTOMER`, and `RENTALS`. Use SQL DDL to specify the schema for this database, including all the applicable constraints. You are free to choose reasonable attributes for the first two relations. The relation `RENTALS` is intended to describe who rented what and should have these attributes: `CustomerId`, `ItemId`, `RentedFrom`, `RentedUntil`, and `DateReturned`.
- 3.11 You are in a real estate business renting apartments to customers. Your job is to define an appropriate schema using SQL DDL. The relations are `PROPERTY(Id, Address, NumberOfUnits)`, `UNIT(ApartmentNumber, PropertyId, RentalPrice, Size)`, `CUSTOMER` (choose appropriate attributes); `RENTALS` (choose attributes; this relation should describe who rents what, since when, and until when), and `PAYMENTS` (should describe who paid for which unit, how much, and when). Assume that a customer can rent more than one unit (in the same or different properties) and that the same unit can be co-rented by several customers.
- 3.12 You love movies and decided to create a personal database to help you with trivia questions. You chose to have the following relations: `ACTOR`, `STUDIO`, `MOVIE`, and `PLAYEDIN` (which actor played in which movie). The attributes of `MOVIE` are `Name`, `Year`, `Studio`, and `Budget`. The attributes of `PLAYEDIN` are `Movie` and `Actor`. You are free to choose the attributes for the other relations as appropriate. Use SQL DDL to design the schema and all the applicable constraints.
- 3.13 You want to get rich by operating an auction Web site, similar to eBay, at which students can register used textbooks that they want to sell and other students can bid on purchasing those books. The site is to use the same proxy bidding system used by eBay (<http://www.ebay.com>).

Design a schema for the database required for the site. In the initial version of the system, the database must contain the following information:

1. For each book being auctioned: name, authors, edition, ISBN number, bookId (unique), condition, initial offering price, current bid, current maximum bid, auction start date and time, auction end date and time, userId of the seller, userId of the current high bidder, and an indication that the auction is either currently active or complete
  2. For each registered user: name, userId (unique), password, and e-mail address
- 3.14** You want to design a room-scheduling system that can be used by the faculty and staff of your department to schedule rooms for events, meetings, classes, etc. Design a schema for the database required for the system. The database must contain the following information:
1. For each registered user: name, userId (unique), password, and e-mail address
  2. For each room: room number, start date of the event, start time of the event, duration of the event, repetition of the event (once, daily, weekly, monthly, mon-wed-fri, or tues-thurs), and end date of repetitive event
- 3.15** Design the schema for a library system. The following data should either be contained directly in the system or it should be possible to calculate it from stored information:
1. About each patron: name, password, address, Id, unpaid fines, identity of each book the patron has currently withdrawn, and each book's due date
  2. About each book: ISBN number, title, author(s), year of publication, shelfId, publisher, and status (on-shelf, on-loan, on-hold, or on-loan-and-on-hold). For books on-loan the database shall contain the Id of the patron involved and the due date. For books on hold the database shall contain a list of Ids of patrons who have requested the book.
  3. About each shelf: shelfId and capacity (in number of books)
  4. About each author: year of birth
- The system should enforce the following integrity constraints. You should decide whether a particular constraint will be embedded in the schema, and, if so, show how this is done or will be enforced in the code of a transaction.
1. The number of books on a shelf cannot exceed its capacity.
  2. A patron cannot withdraw more than two books at a time.
  3. A patron cannot withdraw a book if his/her unpaid fines exceed \$5. Assume that a book becomes overdue after two weeks and that it accumulates a fine at the rate of \$.10 a day.
- 3.16** Suppose that the fragment of the Student Registration System shown in Figure 3.4 has two user accounts: Student and Administrator. Specify the permissions appropriate for these user categories using the SQL GRANT statement.
- 3.17** Suppose that the video store of Exercise 3.10 has the following accounts: Owner, Employee, and User. Specify GRANT statements appropriate for each account.
- 3.18** Explain why the REFERENCES privilege is necessary. Give an example of how it is possible to obtain partial information about the contents of a relation by creating foreign-key constraints referencing that relation.

# 4

## Conceptual Modeling of Databases with Entity-Relationship Diagrams and the Unified Modeling Language

We have interviewed the users of the proposed Student Registration System, understood the requirements, and prepared a detailed Specification Document. That document has been approved by the university registrar, and we are ready to begin designing the database portion of the system. Ultimately this means coming up with a set of appropriate *CREATE* statements that declare the database schema—tables, indices, domains, assertions, and so forth.

The main issue in database design is to provide an accurate model of a large enterprise in the form of a relational database that can be efficiently accessed by concurrently executing transactions. As in other engineering disciplines, the complexity of the task requires that the design process be performed according to a well-defined methodology and be evaluated according to a set of objective criteria.

In this chapter, we present two design methodologies for relational databases: the *entity-relationship (E-R) approach* [Chen 1976] and *UML class diagrams* [Booch et al. 1999]. While the E-R approach is the more established of the two, UML is quickly gaining in popularity due to its rich feature set and because—unlike the many E-R notations—it has been standardized by the Object Management Group (<http://www.omg.org/>).

Database design is typically a two-stage process. The initial phase is based on the E-R or UML methodology, which we will learn about in this chapter. The result of this phase is then refined using the *relational normalization theory*, which provides objective criteria for evaluating alternative designs. This theory is discussed in Chapter 6.

A typical database design involves many dozens of relations, hundreds of attributes, and dozens of constraints—a task of daunting complexity. The good news is that many of the mechanisms underlying the E-R approach, UML, and the relational normalization theory have been captured in design software, which relieves humans of the most arduous, routine work. Still, database design requires a good deal of creativity, experience, technical expertise, and understanding of the fundamental principles. You will make significant headway toward the last two requirements by the time you finish reading this chapter.

## 4.1 Conceptual Modeling with the E-R Approach

First, keep in mind that the E-R approach is *not* a relative, a derivative, or a generalization of the relational data model. In fact, it is not a data model at all but a *design methodology*, which can be applied (but is not limited) to the relational model. The term “relationship” refers to one of the two main components of the methodology rather than to the relational data model.

The two main components of the E-R approach are the concepts of *entity* and *relationship*. Entities model the objects that are involved in an enterprise—for example, the students, professors, and courses in a university. Relationships model the connections among the entities—for example, professors *teach* courses. In addition, *integrity constraints* on the entities and relationships form an important part of an E-R specification, much as they do in the relational model. For example, a professor can teach only one course at a given time on a given day.

An **entity-relationship (E-R) diagram** (peek ahead at Figure 4.1, page 72, and Figure 4.2, page 74) is a graphical representation of the entities, relationships, and constraints that make up a given design. As in other visually oriented design methodologies, it provides a graphical summary of the database structure, which is extremely useful to the designer—not only in validating the correctness of the design but also in discussing it with colleagues and in explaining it to the programmers who will be using it. Unfortunately, there is no standard drawing convention for E-R diagrams, and hence there is a good deal of variation among database texts in many aspects of this approach.

Once the enterprise is represented by a set of E-R diagrams, there are standard ways of converting the diagrams into sets of **CREATE TABLE** statements. Unfortunately, not all aspects of an E-R diagram can be adequately captured with **CREATE** statements. These and related issues will be discussed in Section 4.5.

The creative part of the E-R methodology consists in deciding what entities, relationships, and constraints to use in modeling the enterprise. Some examples included in this and other texts might make these decisions look easy, but in practice designers must combine a detailed understanding of the workings of the enterprise with a considerable amount of technical knowledge and experience.

An important advantage of the methodology is that the designer can focus on complete and accurate modeling of the enterprise, without (initially) worrying about efficiently executing the required queries and updates against the final database. Later, when the E-R diagrams are to be converted to **CREATE TABLE** statements, the designer can add efficiency considerations to the final table designs using the normalization theory (Chapter 6) and tuning techniques to be discussed in Chapter 10.

## 4.2 Entities and Entity Types

The first step in the E-R approach is to select the entities that will be used to model the enterprise. An **entity** (or **entity instance**) is quite similar to an *object*, except that an entity does not have methods, that is, operations that take arguments and either return values or change the entity in some way. An entity can be a concrete object in the real world, such as John Doe, the Cadillac parked at 123 Main Street, or the

Empire State Building. Or it might be an abstract object, such as the Citibank account 123456789, the database course CS305, or the Computer Science Department at Stony Brook.

Similar entities are aggregated into **entity types**. For instance, John Doe, Mary Doe, Joe Blow, and Ann White might be aggregated into the entity type PERSON based on the fact that these entities represent humans. John Doe and Joe Blow might also belong to the entity type STUDENT because in our sample database of Chapter 3 these objects represented students. Similarly, Mary Doe and Ann White might be classified as members of the entity type PROFESSOR.

Other examples of entity types include

- CS305, MGT315, and EE101—entities of type COURSE
- Alf and E. T.—entities of type SPACEALIEN
- CIA, FBI, and IRS—entities of type GOVERNMENTAGENCY

**Attributes.** Like relations and objects, entities are described using attributes. Every attribute of an entity specifies a particular property of that entity. For instance, the Name attribute of a PERSON entity normally specifies a string of characters that denotes the real-world name of the person represented by that database entity. Similarly, the Age attribute specifies the number of times the earth had circled around the sun since the moment that a particular person was born. As in the relational model, the **domain** of an attribute specifies the set from which its values are drawn.

In principle, it is possible for two different entities of the same type to have identical values in all of their attributes. This is one important difference with tuples in the relational model. However, in practice it is not advisable to introduce entity types that can have entities that cannot be distinguished by their attributes.

In all of our examples, each particular entity type included only semantically related entities. Indeed, it is usually pointless to classify people, cars, and paper clips in one entity type because they have little in common in a typical enterprise. It is more useful to classify semantically similar entities in one entity type, since they are likely to have common attributes that describe them. For example, in any enterprise, people have many common attributes, such as Name, Age, and Address. Classification into entity types allows us to associate these attributes with the entity type instead of with the individual entities.

Of course, different entity types will generally have different sets of attributes. For instance, the PAPERCLIP entity type might have attributes Size and Price, while COURSE might have attributes CrsName, CrsCode, Credits, and Description.

**Brain Teaser:** Is it possible for an entity type not to have attributes?

Unlike the relational model, E-R attributes can be **set-valued**. This means that the value of an attribute can be a set of values from the corresponding domain rather than a single value. For example, an entity type PERSON might have set-valued attributes ChildrenNames and Hobbies.

The inability to express set-valued attributes conveniently was one of the major criticisms of the relational data model that motivated the development of the object-oriented data model. However, the use of set-valued attributes in the E-R model is just a matter of convenience. Relations (as defined in Chapter 3) can be used to model entities with set-valued attributes with some extra effort.

**Keys.** As with the relational model, it is useful to introduce key constraints associated with entity types. A **key constraint** on an entity type,  $\mathcal{S}$ , is a set of attributes,  $\bar{A}$ , of  $\mathcal{S}$  such that

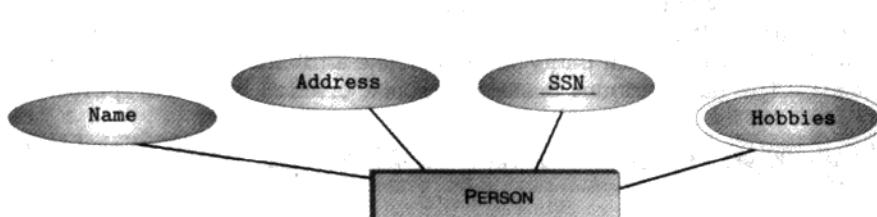
1. No two entities in  $\mathcal{S}$  have the same value for each of the attributes in  $\bar{A}$  (for instance, two different COMPANY entities cannot have the same value in both the Name and the Address attributes).
2. No proper subset of the attributes in  $\bar{A}$  has property 1 (i.e., the set  $\bar{A}$  is *minimal* with respect to property 1).

**Brain Teaser:** Does an entity type necessarily have to have a key?

Entity keys are analogous to candidate keys in the relational model. One subtle difference is that attributes in the E-R approach can be set-valued and such an attribute can be part of a key. However, in practice set-valued attributes that occur in keys are not very natural and are often indicative of poor design.

**Schema.** As in the relational model, we define the **schema** of an entity type to consist of the name of the type, the collection of its attributes (with their associated domains and the indicator of whether each attribute is set-valued or single-valued), and the key constraints.

**E-R diagram representation.** Entity types are represented in E-R diagrams as rectangles, and their attributes are represented as ovals. Set-valued attributes are represented as double ovals. Underlined attributes are keys. Figure 4.1 depicts one possible representation of the PERSON entity type.



**FIGURE 4.1** Fragment of the E-R diagram for the entity type PERSON. Here **Hobbies** is a set-valued attribute, and **SSN** is underlined to indicate that it is a key.

## 4.3 Relationships and Relationship Types

The E-R approach makes sharp distinction between the entities themselves and the mechanism that relates them to each other. This mechanism is called **relationship** (or **relationship instance**). Just as entities are classified into entity types, relationships that relate the same types of entities and that have the same meaning are grouped into **relationship types**.

For instance, STUDENT entities are related to PROGRAM entities via relationships of type MAJORSIN. Thus two instances of MAJORSIN might be the relationships between John Doe and computer science and Joe Blow and economics. Likewise, PROFESSOR entities are related to the departments they work for via relationships of type WORKSIN.

The concept of a *relationship* in the E-R approach is distinct from the concept of a *relation* (i.e., table) in the relational data model. Along with entities, relationships are modeling primitives in the arsenal of the E-R approach, and they are not tied to a particular data model. For instance, in a relational DBMS, both entity and relationship types are typically represented as relations. In an object-oriented database, they are typically modeled as classes. We will see, however, that in some cases relationship types are represented not as tables or classes, but rather as attributes and constraints.

**Attributes and roles.** Like entities, relationships can have attributes. For instance, the relationship MAJORSIN might have an attribute Since, which indicates the date the student was admitted into the corresponding major. The WORKSIN relationship might have the attribute Since to indicate the start date of employment.

Attributes do not provide a complete description of relationships. Consider the entity type EMPLOYEE and the relationship REPORTSTO, which relates employees to other employees. The first type of employee is the subordinate while the second is the boss. Thus, if we just say that (John, Bill) is a relationship, of type REPORTSTO, we still do not know who reports to whom.

Splitting the EMPLOYEE entity type into SUBORDINATE and SUPERVISOR does not help, because REPORTSTO might represent the entire chain of reporting in a corporate hierarchy, making some employees subordinates and supervisors at the same time.

The solution is to recognize that the various entity types participating in a relationship type play different roles in that relationship. For each entity type participating in a relationship type we define a **role** and give that role a name. For example, Subordinate and Supervisor are two roles that connect a relationship instance of REPORTSTO to the two entity instances that it relates in EMPLOYEE. Thus, a role is similar to an attribute, but instead of specifying some property of a relationship, it specifies in what way an entity type participates in the relationship. Both roles and attributes are part of the schema of the relationship type.

For example, the relationship type WORKSIN has two roles, Professor and Department. The Professor role identifies the PROFESSOR entity involved in a WORKSIN relationship, and the Department role identifies the corresponding

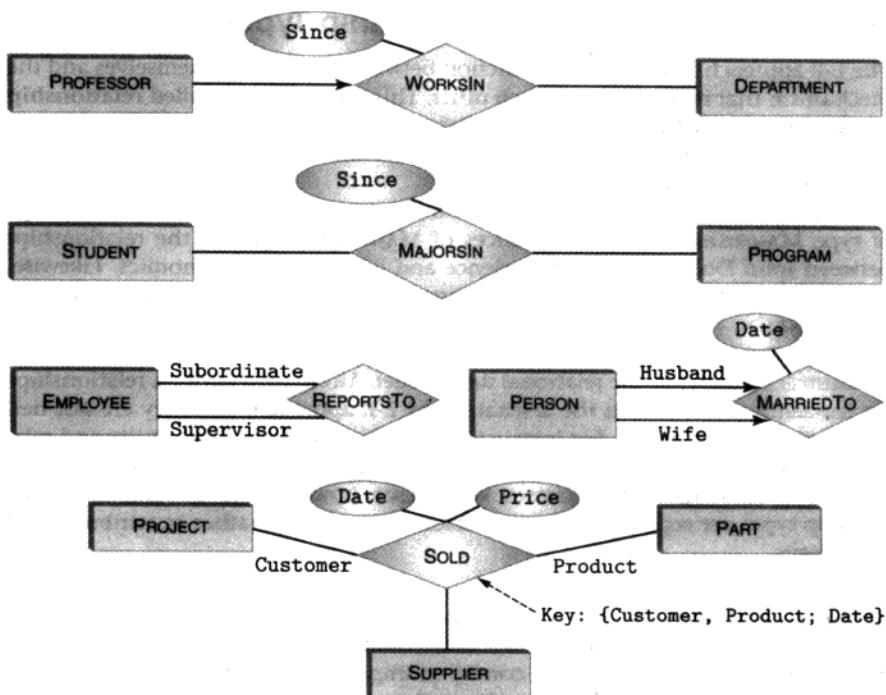


FIGURE 4.2 E-R diagrams for several relationship types.

DEPARTMENT entity in the relationship. Similarly, the relationship type MAJORSIN has two roles, Student and Program.

When all of the entities involved in a relationship belong to distinct entity types (as in WORKSIN and MAJORSIN), it is not necessary to explicitly indicate the roles, because we can always adopt some convention, such as naming the roles after the corresponding entity types (which is typical in practice).<sup>1</sup> Thus, the roles of WORKSIN are Professor and Department. This simplifying convention is not possible when some of the entities involved are drawn from the same entity type, as is the case with the REPORTSTO relationship. Here, we have to explicitly indicate the roles, Subordinate and Supervisor. In other situations (e.g., the relationship SOLD), naming the roles explicitly can help understand the intent behind the particular entity type. Figure 4.2 shows several examples of relationships, including those where roles are named explicitly.

<sup>1</sup> When confusion might arise, we will use different fonts to distinguish entity types from the roles they play in various relationships.

To summarize, the **schema of a relationship type** includes

- A list of attributes along with their corresponding domains. An attribute can be single-valued or set-valued.
- A list of roles along with their corresponding entity types. Unlike attributes, roles are always single-valued.
- A set of constraints. In Figure 4.2, some constraints are represented as arrows. This will be explained later.

The number of roles engaged in a relationship type is called the **degree** of the type.

We can now define the concept of a relationship more precisely. A relationship type  $R$  of degree  $n$  is defined by its attributes  $A_1, \dots, A_k$  and roles  $R_1, \dots, R_n$ . The relationships populating  $R$  are defined to be tuples of the form

$$(e_1, e_2, \dots, e_n; a_1, a_2, \dots, a_k)$$

where  $e_1, \dots, e_n$  are entities involved in the relationship in roles  $R_1, \dots, R_n$ , respectively, and  $a_1, a_2, \dots, a_k$  are values of the attributes  $A_1, \dots, A_k$ , respectively. We assume that all of the values of the attributes in the relationship are in their respective domains, as defined in the relationship type, and all of the entities are of the correct entity types, as defined in their respective roles.

For instance, the relationship type MAJORSIN can have the schema

$$(\text{Student}, \text{Program}; \text{Since})$$

where Student and Program are roles and Since is an attribute. One instance in this relationship type might be

$$('Homer Simpson', \text{EE}; 1994)$$

This relationship states that the entity Homer Simpson is a student who has been enrolled since 1994 in the program represented by the entity EE. The first two components in the tuple are entities; the last is a constant from the domain of years.

**Brain Teaser:** Is it possible for a relationship type not to have attributes? Roles?

**E-R diagram representation.** In E-R diagrams, relationship types are represented as diamonds and roles are represented as edges that connect relationship types with the appropriate entity types. If a role must be named explicitly, the name is included in the diagram. Figure 4.2 shows the E-R diagram for several of the relationships we have been discussing (we omitted the attributes of all entities to reduce clutter). The first three relationships in the figure are **binary** because they each relate two entity types. The last relationship is **ternary** because it relates three entity types. This last diagram also illustrates the point that sometimes the semantics of a diagram can be easier to convey if default role names, such as Project and Part are renamed into something more appropriate, such as Customer and Product, respectively.

**Keys.** The key of a relationship enables the designer to express many constraints naturally and uniformly. In the case of the entity types, a key is just a set of attributes that uniquely identifies each entity. However, attributes alone do not fully characterize relationships. Roles must also be taken into account, so we define the **key of a relationship type**,  $R$ , to be a minimal set of roles and attributes of  $R$  whose values uniquely identify the relationship instances in that relationship type.

In other words, let  $R_1, \dots, R_k$  be a subset of the set of all roles of  $R$ , and  $A_1, \dots, A_s$  be a subset of the attributes of  $R$ . Then the set  $\{R_1, \dots, R_k; A_1, \dots, A_s\}$  is a key of  $R$  if the following holds:

1. *Uniqueness.*  $R$  does not have a pair of distinct relationship instances that have the same values for every role and attribute in  $\{R_1, \dots, R_k; A_1, \dots, A_s\}$ .
2. *Minimality.* No subset of  $\{R_1, \dots, R_k; A_1, \dots, A_s\}$  has property 1.

In some cases, the key of a relationship takes a special form. Consider the relationship WORKSIN between entities of type PROFESSOR and type DEPARTMENT. It is reasonable to assume that each department has several professors but that each professor works for at most one department. Because any given PROFESSOR entity can occur in at most one relationship of type WORKSIN, the role Professor is a key of WORKSIN. While there is no universally accepted representation for relationship keys in E-R diagrams, a relationship key that consists of just one role (a **single-role key**) can be conveniently expressed by drawing this role as an arrow pointing in the direction of the relationship's diamond. Observe that there can be several roles each of which forms a key, and so an E-R diagram can have several arrows pointing toward the same diamond. For instance, in Figure 4.2 both {Husband} and {Wife} are keys of the relationship type MARRIEDTO, so each of these roles is represented as an arrow.

Keys that consist of more than one role or attribute are usually represented textually, next to the diamonds that represent the corresponding relationship type. When representing such keys, we first list the roles and then the attributes. For example, in the last diagram of Figure 4.2 one key could be {Customer, Product; Date}. If such a key is declared, it would signify that there can be at most one sales transaction involving a given customer entity and a given product entity on a given date.

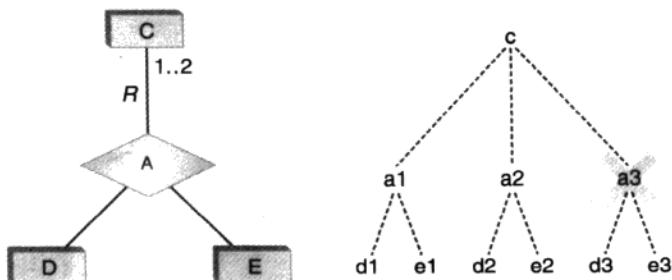
In many situations, however, the relationship has only one key that is the set of all roles. In such a case, we do not specify the key in the diagram.

**Brain Teaser:** Is it possible to have two distinct relationships of the same type that relate the same entities in the same roles and have the same values of attributes?

**Cardinality constraints.** Single-role key constraints, which are drawn as arrows, can be generalized using the notion of a *cardinality constraint*.

Let  $C$  be an entity type and  $A$  be a relationship type that is connected to  $C$  via a role,  $R$ . A **cardinality constraint** on the role  $R$  is a statement of the form  $\text{min..max}$  attached to  $R$ ; it restricts the number of relationship instances of type  $A$  in which

**FIGURE 4.3** Cardinality in the E-R model.



a single entity of type C can participate in role  $R$  to be a number in the interval  $\text{min} \dots \text{max}$  (with end points included).

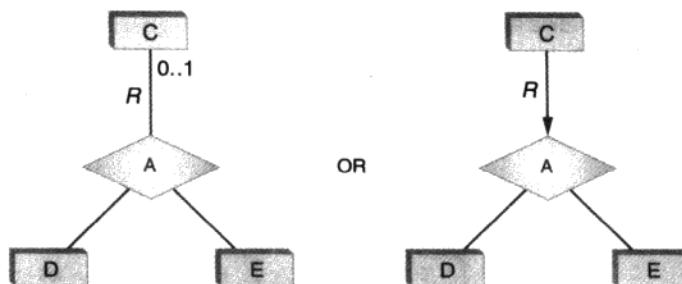
Figure 4.3 shows a diagram with a cardinality constraint on role  $R$ . On the right side it shows a valid instance of this diagram where the entity  $c$  participates in two relationships of type A. The relationship  $a3$  is crossed out because it would violate the cardinality bounds 1..2 on the role  $R$ .

More generally, the E-R model supports cardinality constraints of the form  $\text{min} \dots \text{max}$ , where  $\text{min}$  is a number greater than or equal to 0,  $\text{max}$  is a number greater than 0, and  $\text{min} \leq \text{max}$ . In addition,  $\text{max}$  can be the \* symbol, which represents infinity. Thus, a constraint of the form  $3 \dots *$  on a role  $R$  that connects an entity type, C, with a relationship type, A, means that every entity of type C *must* participate in role  $R$  in *at least* three relationships of type A (with no upper limit). A constraint of the form  $1 \dots 3$  means that every such entity must participate in at least one but no more than three relationships. A constraint of the form  $0 \dots 2$  means that an entity does not have to participate in any relationship of type A in role  $R$ . However, if it does, then it must not participate in more than two relationships. Finally, we note that cardinality constraints of the form  $N \dots N$  (where  $\text{min} = \text{max}$ ) are often abbreviated to just  $N$ , and that \* stands for  $0 \dots *$ .

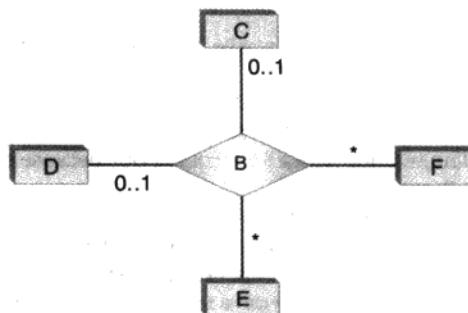
**Brain Teaser:** What does the constraint  $0 \dots *$  mean?

Cardinality constraints generalize the notion of a single-role key, which we earlier represented using arrows. Indeed, representing a role,  $R$ , using an arrow is tantamount to giving it a cardinality constraint  $0 \dots 1$ , as shown in Figure 4.4. Database designers also find it useful to talk about one-to-one, many-to-one, one-to-many, and many-to-many correspondences. These concepts refer to the correspondences between pairs of entity types *implied* by relationships of higher degree. Figure 4.5 illustrates these notions using a relationship type, B, of degree 4. The type of the relationship is determined by the cardinality constraints on the roles of the relationship. For instance, the relationship B implies a **one-to-one** correspondence between the entity types C and D. This means that an entity of type C can be associated

**FIGURE 4.4** Two ways to represent single-role key constraints.



**FIGURE 4.5** Many-to-one, one-to-one, and many-to-many correspondences.



with at most one entity of type D, and vice versa. At the same time, B implies **one-to-many** correspondences of E to C and D (and of F to C and D). This means that an entity of type E can be associated with any number (including zero) of entities of types C and D, but, for example, an entity of type C can be associated with at most one E-entity. Note that the one-to-many correspondence is not symmetric; the inverse correspondence (for example, the correspondence of C to E) is called **many-to-one**. Finally, the correspondence between E and F is said to be **many-to-many**, meaning that an E-entity can be associated with any number of F-entities, and vice versa.

## 4.4 Advanced Features in Conceptual Data Modeling

In this section we introduce a number of more advanced modeling concepts, such as type hierarchies, participation constraints, and the part-of relationship.

### 4.4.1 Entity Type Hierarchies

When modeling an enterprise with the E-R approach, you may find that some entity types are subtypes of others. For instance, every entity of type STUDENT is also a member of the type PERSON. Therefore, all of the attributes of the PERSON type

are applicable to student entities. Students can also have attributes that are not applicable to a typical PERSON entity (e.g., Major, StartDate, GPA). In this case, we say that the entity type STUDENT is a subtype of the entity type PERSON.

Formally, a statement that an entity type R is a **subtype** of the entity type R' is a constraint with the following meaning:

1. Every entity instance in R is also an entity instance in R'.
2. Every attribute in R' is also an attribute in R.

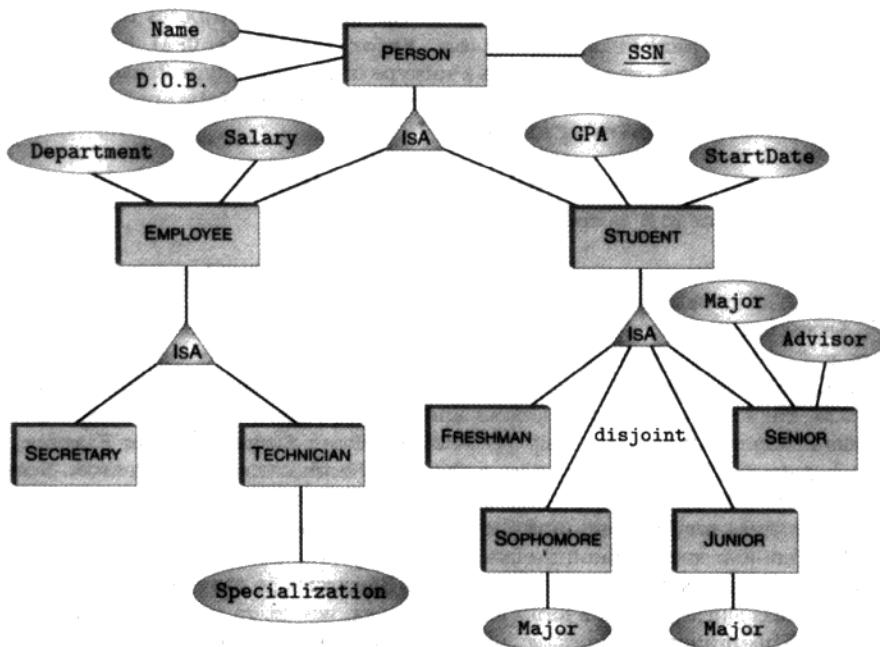
One important consequence of this definition is that any key of a supertype is also a key of all of its subtypes.

Subtyping is not only a constraint but also a relationship between the supertype and its subtype with roles Sub(type) and Super(type). It is often called the **IsA relationship**. For instance, in the IsA relationship type that relates STUDENT and PERSON, the role Sub refers to STUDENT and the role Super refers to PERSON. A particular instance of this relationship type could be *(Homer Simpson, Homer Simpson)*, which states that Homer Simpson is an element of both STUDENT and PERSON. Note that the two entities involved in an IsA relationship are always identical (although the entity types are different), and the names of the roles are fixed.

So what is so special about the IsA relationship? The answer lies in the fact that subtype constraints introduce a **classification hierarchy** in the conceptual model. For instance, FRESHMAN is a subtype of STUDENT, which in turn is a subtype of PERSON. This property is *transitive*, which means that FRESHMAN is also a subtype of PERSON. The transitive property gives us a way to draw diagrams in a more concise and readable manner. Because of property 2 of subtyping, every attribute of PERSON is also an attribute of STUDENT and, by transitivity, is also an attribute of FRESHMAN. This phenomenon is often expressed by saying that STUDENT **inherits** attributes from PERSON and that FRESHMAN inherits attributes from both PERSON and STUDENT.

Note that the inherited attributes (SSN, Name, etc.) are not shown explicitly in Figure 4.6 for the entity types STUDENT and FRESHMAN, and yet they are considered valid attributes because of the IsA relationship. In addition to the inherited attributes, STUDENT and FRESHMAN might have attributes of their own, which their corresponding supertypes might not have. Figure 4.6 illustrates this idea. As STUDENT is a subtype of PERSON, this entity type inherits all of the attributes specified for PERSON, and so there is no need to repeat the attributes Name and D.O.B. (date of birth) for the STUDENT type. Similarly, FRESHMAN, SOPHMORE, and so forth, are subtypes of STUDENT, and so we do not need to copy the attributes of STUDENT and PERSON over to these subtypes. The EMPLOYEE branch of the IsA tree provides another example of attribute inheritance. Every EMPLOYEE entity has attributes Department and Salary. The relationship EMPLOYEE IsA PERSON says that, in addition, every EMPLOYEE entity is also a PERSON entity and, as such, has the attributes Name, SSN, and so forth.

Note that each IsA triangle in Figure 4.6 represents several relationship types. For instance, the upper triangle represents the relationship types STUDENT IsA PERSON



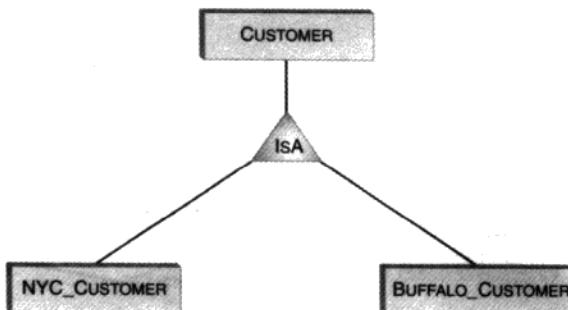
**FIGURE 4.6** Example of an E-R diagram with an ISA hierarchy.

and EMPLOYEE ISA PERSON. Although this notation makes the representation of the ISA relationship different from the representation of other kinds of relationships, it is used because there are a number of constraints associated with entity type hierarchies that can be naturally expressed using such notation.

For instance, the union of the entities that belong to the entity types FRESHMAN, SOPHOMORE, JUNIOR, and SENIOR might be equal to the set of entities of type STUDENT (for example, in a four-year college). This constraint, called the *covering constraint*, can be associated with the lower right ISA triangle. In addition, these entity types might always be disjoint (they are in most American universities), and such a *disjointness constraint* can also be associated with the lower right triangle. There is no universally accepted way of representing covering and disjointness constraints in the E-R diagrams—the most straightforward way is to write the word “*disjoint*” directly on the diagram.

Formally, a group of ISA relationships,  $C_1 \text{ ISA } C; \dots; C_k \text{ ISA } C$ , satisfies the **disjointness constraint** if the sets of entity instances of  $C_1, \dots, C_k$  are disjoint. This group satisfies the **covering constraint** if the union of the sets of instances of  $C_1, \dots, C_k$  equals the set of instances of  $C$ .

**FIGURE 4.7** Using IsA for data partitioning.



**Entity type hierarchies and data partitioning.** While discussing the IsA relationship, we have been focusing on conceptual organization and attribute inheritance. However, these hierarchies are also a good way to approach the issue of physical **data partitioning**. The need for data partitioning often arises in distributed environments, where multiple geographically diverse entities must access a common database. Banking is a typical example because banks often have many branches in different cities.

The problem that arises in such distributed enterprises is that of network delay: accessing a database in New York City from a bank branch in Buffalo can be prohibitive for frequently running transactions. However, the bulk of the data needed by a local bank branch is likely to be of mostly local interest, and it might be a good idea to distribute fragments of such information among databases maintained at the individual branches. This approach is taken in distributed databases.

To see how data partitioning can be addressed at the database design stage, consider an entity type, CUSTOMER, which represents the information about all customers of a bank. For each branch, we can create subtypes, such as NYC\_CUSTOMER or BUFFALO\_CUSTOMER, that are related to CUSTOMER as described in Figure 4.7.

Observe that the constraints associated with type hierarchies provide considerable expressive power in specifying how data might be partitioned. For instance, Figure 4.7 could be interpreted as a requirement that the New York City and Buffalo data must be stored locally. It does not say that the New York City database and the Buffalo customer database must be disjoint, but this can be specified using the disjointness constraint introduced earlier. In addition, we can add the covering constraint to specify that the combined customer information at the branches includes all customers.

#### 4.4.2 Participation Constraints

Suppose that while developing an E-R diagram for your university you have introduced a relationship type, WORKSIN, between the entity types PROFESSOR and DEPARTMENT. Each department has several professors but each professor is a member of a single department, so the role Professor is a key of WORKSIN.

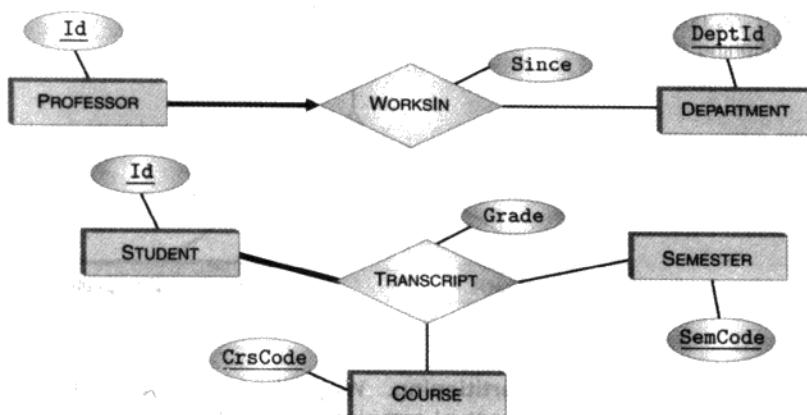


FIGURE 4.8 Participation constraints.

This key constraint ensures that no professor can occur in more than one relationship of type **WORKSIN**. However, it does not guarantee that each professor occurs in *some* relationship of this type. In other words, the key constraint does not rule out professors who do not work for any department (and possibly get away without teaching any courses!). To close this loophole, the designer can use **participation constraints**.

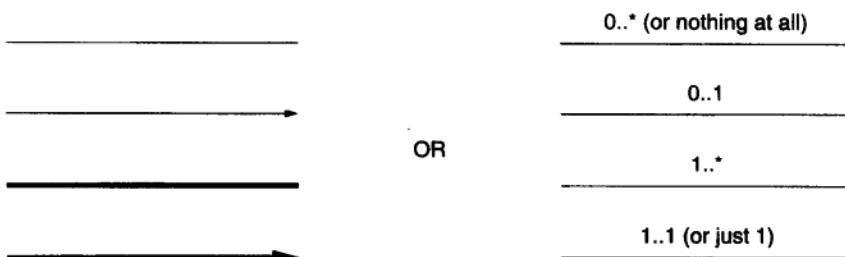
Given an entity type,  $E$ , a relationship type,  $R$ , and a role,  $R$ , a **participation constraint** of  $E$  in  $R$  in role  $R$  states that for every entity instance  $e$  in  $E$ , there is a relationship  $r$  in  $R$  such that  $e$  participates in  $r$  in role  $R$ .

Clearly, requiring that the entity type **PROFESSOR** participates in the relationship type **WORKSIN** in role **Professor** ensures that every professor works in some department.

For another example, we may want to ensure that every student takes at least one course. To this end, we can assume that there is a ternary relationship type, **TRANSCRIPT**, which relates **STUDENT**, **COURSE**, and **SEMESTER**. Our goal can be achieved by imposing a participation constraint on the **Student** role that connects the **STUDENT** entity type to the relationship **TRANSCRIPT**.

One common way of representing participation constraints in an E-R diagram is to draw a thick line for the role that connects the participating entity with the corresponding relationship, as in Figure 4.8. The thick arrow connecting **PROFESSOR** to **WORKSIN** indicates both that each professor participates in at least one relationship (denoted by the thick line) and that each professor can participate in at most one relationship (denoted by the arrow). Hence, a one-to-one mapping between **PROFESSOR** entities and **WORKSIN** relationships exists.

Alternatively, participation constraints can be represented using cardinality constraints. Participation of an entity type,  $E$ , in a relationship type,  $R$ , in role  $R$



**FIGURE 4.9** Line-based representation vs. cardinality constraints.

can be represented using the constraint of the form  $1..*$  placed on the role  $R$ . A participation constraint combined with the single-role key (represented using a thick arrow) can be expressed as a cardinality constraint of the form  $1..1$  (or simply  $1$ ). Figure 4.9 summarizes the correspondence between the two representations.

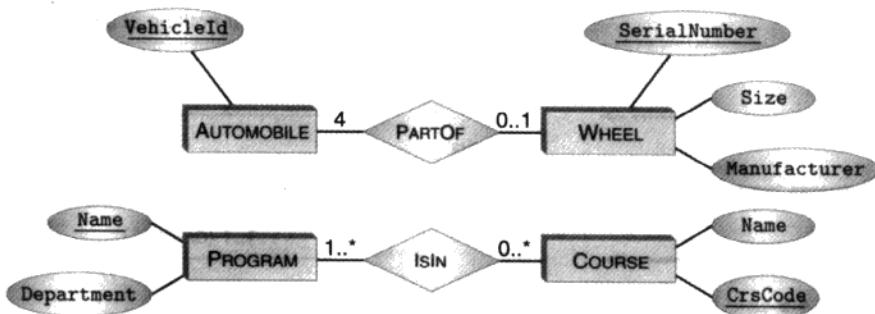
### 4.4.3 The Part-of Relationship

The collective experience of database design suggests that, alongside the **IsA**, **part-of** is a useful kind of relationship. For example, a wheel entity can be part of an automobile entity.

There are two kinds of part-of relationships. In one, the subpart of the whole can exist independently even if the whole is destroyed. This type of part-of relationship is **non-exclusive**. A good example of a non-exclusive part-of relationship is the relationship between an automobile entity and the entities representing its parts. When it is no longer feasible to keep repairing an automobile, it might be brought to a junk yard and taken apart. The automobile no longer exists, but its wheels, transmission, and camshaft may continue their independent existence and even find new life as part of another automobile. In certain cases, the same entity can even be part of several other objects. For instance, the same course may be an integral part of two or more programs of study in a university curriculum.

Another kind of part-of relationship is when the subpart has no existence outside of the whole: when the whole object is destroyed, the subpart goes as well. It is usually further assumed that the subpart cannot be shared, that is, it can belong to *exactly one* whole. For instance, **PROGRAM** of study (biology, physics, etc.) is part of **UNIVERSITY**. If a university is dissolved, its programs no longer exist. Similarly, in a payroll database, **DEPENDENT** can be part of the information associated with the **EMPLOYEE** entity type. When an employee leaves, the information about his dependents is also erased. This type of part-of relationship is **exclusive**.

Non-exclusive part-of relationships do not have special representation in the E-R model. They are treated as regular relationships, and cardinality constraints are used to state whatever is appropriate in each particular situation. In a non-exclusive relationship, a subpart (e.g., an automobile wheel) can exist without



**FIGURE 4.10** Non-exclusive part-of relationship in E-R.

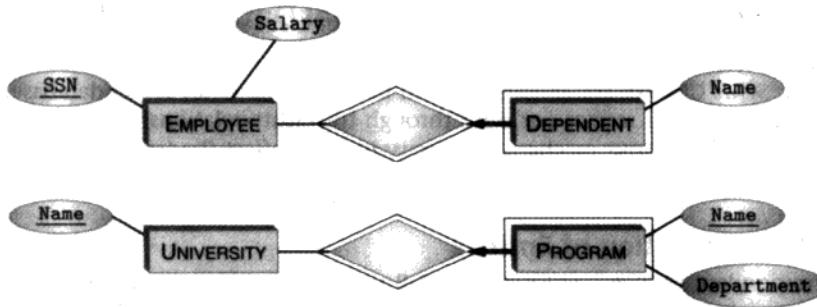
being part of a concrete automobile. Therefore, there need not be a participation constraint between subparts and parts. Likewise, there need not be a many-to-one correspondence between the subpart entity type and the type of the whole part, because the same subpart (e.g., course) can be part of several different wholes (e.g., programs of study). Figure 4.10 shows examples of E-R diagrams representing non-exclusive part-of relationships. This is indicated by the minimum cardinality of 0 for entity types WHEEL and COURSE.

In contrast to non-exclusive part-of relationships, the exclusive ones are represented within the E-R approach using the special machinery of **weak entity types** and **identifying relationships**. Weak entities represent subparts and identifying relationships are the corresponding exclusive part-of relationships. In our examples, PROGRAM and DEPENDENT are weak entity types that are related through identifying relationships to their *master* entity types (which represent whole entities), UNIVERSITY and EMPLOYEE. In E-R diagrams, weak entities and their identifying relationships are represented using double boxes and double diamonds, respectively.

Observe that weak entities always participate in their identifying relationships and that each such entity is related to a single master object. Thus, there is always a thick arrow going from a weak entity to its identifying relationship, as shown in Figure 4.11.

Sometimes designers choose to strip weak entity types of their key attributes and have the entities identified through their relationship with the master entity. For example, the DEPENDENT entity might have the Name attribute, but not the SSN attribute. To find a dependent entity, one would have to first find the corresponding master (an EMPLOYEE entity) and then follow the identifying relationship.

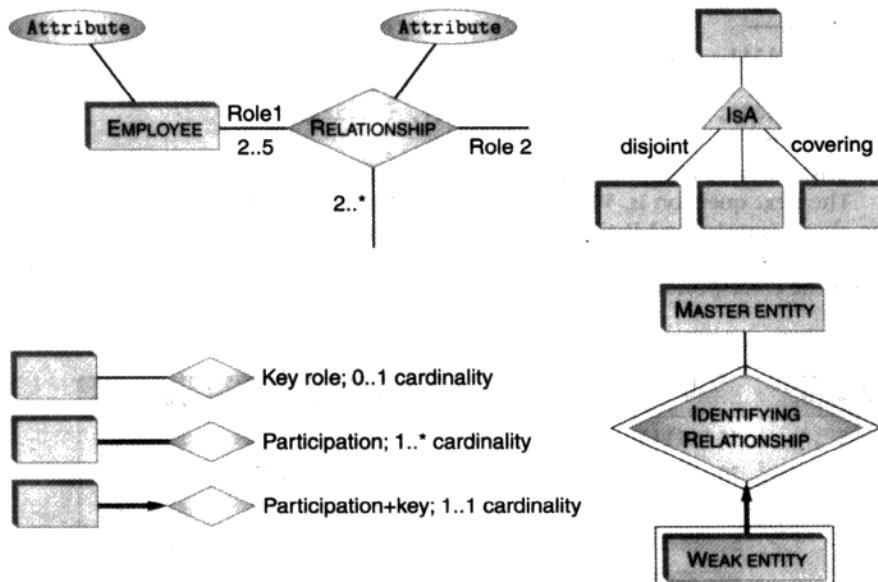
Note that the identifying relationship is not the only kind of relationship in which a weak entity can take part. For instance, PROGRAM in Figure 4.11 can be related to another entity type, EMPLOYEE, via the relationship PROGRAMDIRECTOR. This relationship would not be identifying for PROGRAM (since a program might not have a director), and thus it would be represented using a single diamond. It is also conceivable (although not common) that a weak entity type can participate



**FIGURE 4.11** Exclusive part-of relationship in E-R: weak entities.

in a *ternary* (or higher-degree) relationship type with several master entity types. For instance, certain educational programs might be jointly run by universities and companies. In this case, PROGRAM takes part in a ternary identifying relationship with entities of the types UNIVERSITY and COMPANY, and destruction of either of these entities implies the destruction of the program entity.

Figure 4.12 summarizes the notation used in E-R diagrams.



**FIGURE 4.12** Summary of the E-R notation.

## 4.5 From E-R Diagrams to Relational Database Schemas

Conceptual design is only a means to an end: producing a relational schema and, ultimately, SQL's `CREATE` statements that can be used to build an actual database. In this section, we will retrace our steps through the entity-relationship approach and show how the various mechanisms discussed so far affect the translation into the relational model and SQL.

### 4.5.1 Representation of Entities

The correspondence between entities in the E-R model and relations in the relational model is straightforward. Each entity type is converted into a relation, and each of its attributes is converted into an attribute of the relation.

This simple set of rules might seem suspicious in view of the fact that entities can have set-valued attributes while relations cannot. How can a set-valued attribute of an entity be turned into a single-valued attribute of the corresponding relation without violating the property of data atomicity (defined in Section 3.2) of the relational model?

The answer is that, although each set-valued attribute of an entity type is represented as a single-valued attribute in the resulting relation, each entity instance is represented in the translation by a *set* of tuples—one for each member in the set of the attribute's values. To illustrate, suppose that the entity type PERSON of Figure 4.1 is populated by the following entities:

---

(1111111111, John Doe, 123 Main St., {Stamps, Coins})  
 (5556667777, Mary Doe, 7 Lake Dr., {Hiking, Skating})  
 (987654321, Bart Simpson, Fox 5 TV, {Acting})

---

In translation, we obtain the relation depicted in Figure 4.13.

The next question is, What are the candidate keys of the relation obtained by the above translation? If the entity type does not have set-valued attributes, the answer is simple. Each key (which is a set of attributes) of the entity type becomes a key of the corresponding relation schema. However, if one of the entity attributes is

PERSON	SSN	Name	Address	Hobby
	1111111111	John Doe	123 Main St.	Stamps
	1111111111	John Doe	123 Main St.	Coins
	5556667777	Mary Doe	7 Lake Dr.	Hiking
	5556667777	Mary Doe	7 Lake Dr.	Skating
	987654321	Bart Simpson	Fox 5 TV	Acting

FIGURE 4.13 Translation of entity type PERSON into a relation.

set-valued, determining the keys is a bit more involved. In the entity type PERSON, the attribute SSN is a key because no two PERSON entities can have the same Social Security number. However, in the PERSON relation of Figure 4.13, both John Doe and Mary Doe are represented by a pair of tuples, and their Social Security numbers occur twice. Therefore, SSN is not a key of that relation. What is the problem here?

Clearly, the set-valued attribute Hobby is the troublemaker: to obtain a key of the relation in question, we must include this attribute. Thus, the key of the PERSON relation in Figure 4.13 is {SSN, Hobby}.

The following CREATE TABLE statement defines the schema for the PERSON relation.

---

```
CREATE TABLE PERSON (
    SSN      INTEGER,
    Name     CHAR(20),
    Address  CHAR(50),
    Hobby    CHAR(10),
    PRIMARY KEY (SSN, Hobby) )
```

---

4.1

Even though we have identified a key, a careful examination of the above table leaves us uneasy. It does not seem right that the Hobby attribute should have anything to do with identifying tuples in the PERSON relation. Furthermore, in the original entity type PERSON, any concrete value of SSN is known to uniquely identify the value of Name and Address. In the translation of Figure 4.13, we see that this property still holds, but it is not captured by the primary-key constraint, which states that in order to uniquely determine a tuple, we must specify the value of *both* SSN and Hobby. In contrast, in the entity type PERSON the value of Hobby is not required to determine the value of Name and Address. This important constraint has been lost in the translation!

The preceding example is the first indication that the E-R approach alone does not guarantee good relational design. Chapter 6 will provide a host of objective criteria that can help database designers evaluate the relational schema obtained by converting E-R diagrams into relations. In particular, the problem with the relation in Figure 4.13 is that it is not in a certain *normal form*. Chapter 6 proceeds to develop algorithms that can automatically rectify the problem by splitting the offending relations into smaller relations that are in a desired normal form. For instance, in our case, the PERSON relation would be split into two: one with the attributes SSN, Name, and Address, and another with the attributes SSN and Hobby.

**Algorithm for converting entities into relations.** We can now summarize the algorithm for translating entity types into the relational schema:

- Each entity type becomes a relation.
- Each attribute of the entity becomes an attribute of that relation.

- If attributes  $K_1, K_2, \dots, K_n$  form a key of the entity, then the attributes  $K_1, K_2, \dots, K_n, S_1, \dots, S_k$  form a candidate key of the relation. Here  $S_1, \dots, S_k$  is a list of all set-valued attributes of the entity.

### 4.5.2 Representation of Relationships

The algorithm that maps relationship types into relation schemas can be summarized as follows:

1. Determine the attributes of the relation schema for the relationship type.
2. Determine the candidate keys of the schema.
3. Determine the foreign-key constraints.

We discuss each step of the algorithm in turn.

1. *Attributes of the relation schema derived from a relationship type R.* The attributes of the relation schema are
  - (a) The attributes of R itself.
  - (b) For each role in R, the primary key of the associated entity type. These attributes will become foreign keys referencing the corresponding entity types (as explained below).
  - (c) Each attribute in these primary keys must be declared as NOT NULL unless this is already implied by the PRIMARY KEY constraint.

Note that in (b) we use the primary key of the entity type—not the primary key of the relation schema constructed out of that entity type—because the goal is to uniquely identify the entity involved in the relationship. Thus, for example, in the case of a role associated with the entity type PERSON we use SSN and omit Hobby.

Also, spend a moment to contemplate the reason for the NOT NULL requirement in step (c): By definition, a relationship, for example, of degree four must have exactly four entities involved and none of them can be missing—otherwise, it will not be a relationship of degree four. Therefore, a relationship must provide references to all of its participating entities, and none of the corresponding attributes can be NULL.

While this sounds simple enough, there are two small problems: the primary keys of different roles can have identically named attributes that mean different things or different attributes that mean the same thing. For example, in the MARRIAGE relationship, the primary key of each of the roles, Husband and Wife, could be (FirstName, FamilyName). Assuming that couples use the same family name, the relation schema derived from this relationship will have two pairs of identically named attributes. In the first pair, the two occurrences of FirstName mean different things and must be renamed (e.g., to HusbandFirstName and WifeFirstName). In the second pair, the two occurrences of FamilyName mean the same thing—the family name of a married couple. In this case we can simply delete the second occurrence. We can also imagine situations where the keys can

have differently named attributes that mean the same thing. In this case we can also delete duplicate occurrences of such attributes.

2. *Candidate keys of the relation schema.* In most cases, the keys of the relation schema are obtained by direct translation from the keys of  $R$  itself. That is, if a role,  $R$ , of  $R$  belongs to the key of  $R$ , then the attributes of the primary key,  $\mathcal{K}$ , of the entity type associated with  $R$  must belong to the candidate key of the relation schema derived from  $R$ .

A slight problem arises when  $R$  has set-valued attributes. In that case, we resort to an earlier trick that was used for converting entity keys into relation keys: all set-valued attributes must be included in the candidate key of the relation (see the PERSON entity-to-relation translation in (4.1)). Note that roles are always single-valued, so this special treatment of set-valued attributes does not apply to roles.

3. *Foreign-key constraints of the relation schema.* Because, in the E-R model, a role always refers to some entity (which is mapped to a relation), roles translate into foreign-key constraints. The foreign keys of the relation schema derived from  $R$  are constructed as follows.

Let  $R$  be a role in  $R$  that connects  $R$  to an entity type,  $E$ . We use  $\text{rel}(R)$  and  $\text{rel}(E)$  to denote the relational schemas derived from  $R$  and  $E$ , respectively.

For each such role, the primary key,  $\mathcal{K}$ , of  $E$  (which, by construction, is included among the attributes of  $\text{rel}(R)$ ) becomes a foreign key of the schema  $\text{rel}(R)$  that references  $\text{rel}(E)$ , provided that  $\mathcal{K}$  is also the primary key of  $\text{rel}(E)$ .

The reason for the caveat in this definition is that, as we have seen, the primary key of an entity type need not be the primary key of the corresponding relational schema. For instance, in the case of the PERSON entity type, SSN is the primary key of the entity type, but not of the corresponding relation schema (which has {SSN, Hobby} as its primary key). This type of problem is eliminated by the relational normalization theory, to be discussed in Chapter 6.

Figure 4.14 shows the CREATE TABLE commands that define the schemas corresponding to some of the relationships in Figure 4.2 on page 74. Observe that, in the MARRIEDTO relation, we did not define the foreign-key constraint: although SSNhusband and SSNwife clearly reference the SSN attribute of the PERSON relation, SSN is not a candidate key for that relation, as explained earlier. The UNIQUE constraint in the schema guarantees that SSNwife is a candidate key. The NOT NULL constraints in WORKSIN and MARRIEDTO comes from step 1(c) of the above algorithm: it ensures that each relationship has both of its entities present.

Note that when E-R diagrams are translated into tables, some of these tables describe entities and others describe relationships. Thus, the first E-R diagram of Figure 4.2 would translate into three tables: one to describe the entity type PROFESSOR, one to describe the entity type DEPARTMENT, and one to describe the relationship type that links professors to departments.

**FIGURE 4.14** Translations of some relationships.

```

CREATE TABLE WORKSIN (
    Since      DATE,
    ProfId     INTEGER,
    DeptId     CHAR(4) NOT NULL,
    PRIMARY KEY (ProfId),
    FOREIGN KEY (ProfId) REFERENCES PROFESSOR (Id),
    FOREIGN KEY (DeptId) REFERENCES DEPARTMENT )

CREATE TABLE MARRIEDTO (
    Date       DATE,
    SSNhusband INTEGER,
    SSNwife    INTEGER NOT NULL,
    PRIMARY KEY (SSNhusband),
    UNIQUE (SSNwife) )

CREATE TABLE SOLD (
    Price      INTEGER,
    Date       DATE,
    ProjId    INTEGER,
    SupplierId INTEGER,
    PartNumber INTEGER,
    PRIMARY KEY (ProjId, SupplierId, PartNumber, Date),
    FOREIGN KEY (ProjId) REFERENCES PROJECT,
    FOREIGN KEY (SupplierId) REFERENCES SUPPLIER (Id),
    FOREIGN KEY (PartNumber) REFERENCES PART (Number) )

```

---

### 4.5.3 Representing IsA Hierarchies in the Relational Model

There are several ways to represent the IsA relationship using relational tables. First we present a general way of dealing with IsA in the relational model, and then we show two other techniques, which may have advantages in certain situations.

1. *General representation.* Choose a candidate key for all entity types related by the IsA hierarchy. Add the attributes of this key to each entity type in the hierarchy, and then convert the resulting entities into relations, as discussed in Section 4.5.1. The choice of such a key is possible because, as was observed in Section 4.4.1, a key of a supertype is also a key of each subtype. Therefore, the required key is the key of the top entity type in the hierarchy.

For instance, in Figure 4.6 we can choose {SSN} as the common key of all entity types in the hierarchy, so SSN will be added to STUDENT, EMPLOYEE, etc. The next step in the translation process will yield the following relation schemas.

```

PERSON(SSN, Name, D.O.B.)
STUDENT(SSN, StartDate, GPA)
FRESHMAN(SSN)
SOPHOMORE(SSN, Major)
JUNIOR(SSN, Major)

```

**SENIOR(SSN, Major, Advisor)**  
**EMPLOYEE(SSN, Department, Salary)**  
**SECRETARY(SSN)**  
**TECHNICIAN(SSN, Specialization)**

In addition, inclusion dependencies pointing from sub-entity types to parent entity types are needed. This ensures that every entity in a subtype also belongs to the supertype. In our particular case, since SSN is a key in the supertypes, the inclusion dependencies can be specified as foreign-key constraints. For instance, the relations STUDENT and EMPLOYEE will have the constraint

---

**FOREIGN KEY (SSN) REFERENCES PERSON**

---

Similarly, the relations corresponding to the sub-entity types FRESHMAN, . . . , SENIOR will have the constraint

---

**FOREIGN KEY (SSN) REFERENCES STUDENT**

---

Finally, the relations SECRETARY and TECHNICIAN will have the constraint

---

**FOREIGN KEY (SSN) REFERENCES EMPLOYEE**

---

2. *Representation for disjoint ISA relationships.* If a group of ISA relationships,  $C_1$  IsA  $C$ ; . . . ;  $C_k$  IsA  $C$ , satisfies the disjointness constraint, then the following representation can be used. All entities that participate in the relationship are stored in a single relation whose attribute set is the union of the attribute sets of all entity types involved (i.e.,  $\text{attributes}(C) \cup_{i=1}^k \text{attributes}(C_i)$ ). One extra attribute is added to indicate the original entity type of each tuple in the relation. We should also add the common key inherited from the top type in the ISA hierarchy, as in the general translation algorithm discussed earlier. Tuples that come from the entity types that do not have certain attributes are padded with NULLs over such attributes. (For instance, tuples from  $C$  that are not in any of the  $C_i$ s are likely to have such NULLs.)

As an example, consider the part of the hierarchy below the STUDENT entity type. We can create a single relation schema with the attributes SSN, GPA, StartDate, Major, Advisor (SSN is the key attribute inherited from the supertype Person) plus the new attribute Status with the domain {Freshman, Sophomore, Junior, Senior}. This attribute is used to indicate the original entity type of every tuple. For instance, a FRESHMAN entity will be represented by a tuple that has normal values in the attributes SSN, GPA, StartDate, and Status, and NULL in the attributes Major and Advisor. A student who does not belong to any of the four subtypes of STUDENT will have a NULL also in the Status attribute.

**Brain Teaser:** Name one advantage and one disadvantage of this representation compared to the general representation for the ISA relationship.

3. *Representation for covering Isa relationships.* If a group of Isa relationships,  $C_1$  Isa  $C$ ;  $\dots$ ;  $C_k$  Isa  $C$ , satisfies the covering constraint, we can use the following translation: create one relation schema per each subtype of the Isa relationship. The attribute set of the relation associated with a subtype,  $C_i$ , is the union of the attribute sets for the subtype and the supertype (i.e.,  $\text{attributes}(C_i) \cup \text{attributes}(C)$ ) plus the key inherited from the top entity in the hierarchy, which was used in all previous translations.

For instance, assuming that the only people described in our database are employees and students, the Isa relationship that connects EMPLOYEE and STUDENT to their supertype PERSON satisfies the covering constraint and can be represented by the following pair of relation schemes:

---

```
EMPLREL(SSN, Name, D.O.B., Department, Salary)
STUDREL(SSN, Name, D.O.B., GPA, StartDate)
```

---

An advantage of this representation over the general one is that attributes such as Name and Salary are in the same relation and thus queries of the form "What is John's salary?" can be answered more efficiently. On the other hand, this representation causes redundant information to be stored if, for example, John is both an employee and a student. In that case, John's name, date of birth, and SSN will be stored both in the tuple that represents John in the EMPLREL relation and in the tuple for John in STUDREL.

#### 4.5.4 Representation of Participation Constraints

Conceptually, representing participation constraints in the relational model is easy. We have already seen, in Figure 4.14 on page 90, the CREATE TABLE statement for the WORKSIN relationship. So all it takes to enforce the participation constraint of PROFESSORS in WORKSIN is to specify an inclusion dependency (refer back to Section 3.2.2 for the definition) that states

---

```
PROFESSOR(Id) references WORKSIN(ProfId)
```

---

Since Id is a foreign key (because ProfId is a key of WORKSIN), we can state the participation constraint in SQL by simply declaring the Id attribute of PROFESSOR as a foreign key.

---

```
CREATE TABLE PROFESSOR (
    Id      INTEGER,
    Name    CHAR(20),
    PRIMARY KEY (Id),
    FOREIGN KEY (Id) REFERENCES WORKSIN (ProfId) )
```

---

Note that the foreign-key constraint does not rule out the possibility that Id can be NULL, which means that in general a NOT NULL constraint for Id would be in order. However, in our case, Id is declared as a primary key of PROFESSOR, so the NOT NULL

constraint is implicit. Also observe that the `DeptId` attribute is missing (in contrast to Figure 3.5). It was not included because the above schema for `PROFESSOR` was derived from the E-R diagram in Figure 4.8, and `DeptId` is not one of the attributes of the `PROFESSOR` entity type there. Instead, the connection between professors and departments is represented by the `WORKSIN` relation.

We can do a better translation by noticing that `Id` is a key of `PROFESSOR` and also of `WORKSIN` (indirectly, through the foreign-key constraint). Thus, we can merge the attributes of `WORKSIN` into the `PROFESSOR` relation and identify the attribute `ProfId` with `Id`. This is possible because the common key of these tables guarantees that each `PROFESSOR` tuple has *exactly* one corresponding `WORKSIN` tuple, so no redundancy is created by concatenating such related tuples. This yields the table `PROFESSORMERGEDWITHWORKSIN`.

---

```
CREATE TABLE PROFESSORMERGEDWITHWORKSIN (
    Id      INTEGER,
    Name    CHAR(20),
    DeptId CHAR(4) NOT NULL,
    Since   DATE,
    PRIMARY KEY (Id)
    FOREIGN KEY DeptId REFERENCES DEPARTMENT )
```

---

Note one subtle point about this merge—the `NOT NULL` clause in the `DeptId` attribute. One might conjecture that `DeptId` simply inherited `NOT NULL` from the `WORKSIN` relation during the merge. However, this is not a sufficient reason, for if professors could exist in the database without working for any department, then the `DeptId` attribute in `PROFESSORMERGEDWITHWORKSIN` should be allowed to accept a null value. In reality, the `NOT NULL` specification follows from two facts: the `NOT NULL` clause in the `DeptId` attribute on `WORKSIN` and the participation constraint by `PROFESSOR` in `WORKSIN`, which ensures that professors cannot exist outside of a department.

Although conceptually the representation of participation constraints in the relational model amounts to nothing more than specifying an inclusion dependency, the actual representation in SQL is not always as simple as the previous examples might suggest. The reason is that not all inclusion dependencies are foreign-key constraints (see Section 3.2.2), and expressing such constraints in SQL requires the heavier machinery of assertions or triggers, which can negatively affect the performance.

An example of this situation is the constraint on the participation of `STUDENT` entity type in the `TRANSCRIPT` relationship, depicted in Figure 4.8. The translation of `TRANSCRIPT` to SQL is

---

```
CREATE TABLE TRANSCRIPT (
    StudId  INTEGER,
    CrsCode CHAR(6),
    Semester CHAR(6),
    Grade   CHAR(1),
```

```
PRIMARY KEY (StudId, CrsCode, Semester),
FOREIGN KEY (StudId) REFERENCES STUDENT (Id),
FOREIGN KEY (CrsCode) REFERENCES COURSE (CrsCode),
FOREIGN KEY (Semester) REFERENCES SEMESTERS (SemCode))
```

As before, the foreign-key constraints specified for the TRANSCRIPT table do not guarantee that every student takes a course. To ensure that every student participates in some TRANSCRIPT relationship, the STUDENT relation must have an inclusion dependency of the form

---

STUDENT(Id) references TRANSCRIPT(StudId)

---

However, since StudId is not a candidate key in TRANSCRIPT, this inclusion dependency is not a foreign-key constraint. In Chapter 3, we illustrated how inclusion dependencies can be defined using the CREATE ASSERTION statement (see (3.4) on page 52).

Unfortunately, verifying general assertions is often significantly more costly than verifying foreign-key constraints, so the use of constraints such as (3.4) should be carefully weighed against the potential overhead. For instance, if it is determined that including such an assertion slows down crucial database operations, the designer might opt for checking the inclusion dependency as part of a separate, periodically run transaction and forgo the real-time check.

#### 4.5.5 Representation of the Part-of Relationship

We distinguish three cases: two deal with various forms of non-exclusive part-of relationships and one with the exclusive case.

- *Non-exclusive part-of: subpart can exist independently and be shared between different wholes.* In this case, translation into the relational model is done as if part-of were a regular relationship with no special properties, that is, it translates into a separate relation.
- *Non-exclusive part-of: subpart can exist independently but can be part of at most one whole.* This type of relationship would be represented by a diagram that has a thin arrow leading from the subpart entity to the part-of relationship (or an edge adorned with a cardinality constraint 0..1). Since every subpart can participate in at most one part-of relationship, we can represent both the subpart type and the relationship type using a single relation similarly to the merge of the relations PROFESSOR and WORKSIN into PROFESSORMERGEDWITHWORKSIN on page 93. A foreign key in the merged relation will reference the whole entity that contains the subpart and, in this way, the relationship between the subpart and the whole will be preserved.

Since subparts do not need to be part of a whole, those that do not will have a null value in the fields of that foreign key. Therefore, the NOT NULL clause should *not* be attached to the attributes of that foreign key. For example, if AUTOMOBILE has {VehicleId} as its key and WHEEL has the attributes

`SerialNumber`, `Size`, and `Manufacturer`, then both the `WHEEL` entity type and the `PARTOF` relationship (see Figure 4.10) can be represented as

---

```
CREATE TABLE WHEELMERGEDWITHPARTOF (
    SerialNumber INTEGER,
    Size          CHAR(10),
    Manufacturer CHAR(20),
    VehicleId    CHAR(20)
    PRIMARY KEY (SerialNumber),
    FOREIGN KEY (VehicleId) REFERENCES AUTOMOBILE )
```

---

In line with the previous discussion, we did *not* declare `VehicleId` to be NOT NULL. So, if the car is disembodied and the wheel is sold separately, then `VehicleId` can be set to NULL.

- *Exclusive part-of.* In this case, the subpart is a weak entity and the part-of relationship is its identifying relationship. A weak entity participates in one and only one identifying relationship, and a thick arrow must exist between the subpart and the relationship. Therefore, this case is translated according to the rules for participation and key constraints. The main difference with respect to the previous case is that now the attributes of the foreign key that point to the master entity must have NOT NULL attached to them. An example of this kind of translation was given before (see `PROFESSOR`, `WORKSIN`, and `PROFESSORMERGEDWITHWORKSIN`).

## 4.6 UML: A New Kid on the Block \*

Unified Modeling Language (UML) [Booch et al. 1999] is a culmination of a long process, which led to unifying and generalizing a number of methodologies in software engineering, business modeling and management, database design, and others. The E-R approach was just one of the many inputs that have influenced the final product. Because UML designers tried to capture every known aspect of the design activity, the approach ended up with every complication a modeling language can possibly have. Nevertheless, perhaps actually *due* to its Swiss Army knife model, UML is gaining in popularity in many areas of design, including database design. In this section, we will introduce UML *class diagrams*—a subset of UML that is suitable for conceptual modeling of databases.

We should mention that other parts of UML are also useful for modeling various aspects of database applications. Thus, in Chapter 14 we employ UML **use case diagrams** to describe user interactions with the Student Registration System and UML **sequence diagrams** to model the dynamic aspects of those **use cases**. In Chapter 15, we use UML **state diagrams** to describe the behavior of various objects in that system. In addition UML **activity diagrams** can be used to show how activities are coordinated, and **collaboration diagrams** can be used to describe interactions (i.e., message exchange) among the different objects that comprise a complex system. And then there are **component diagrams**, **deployment diagrams**, and more.

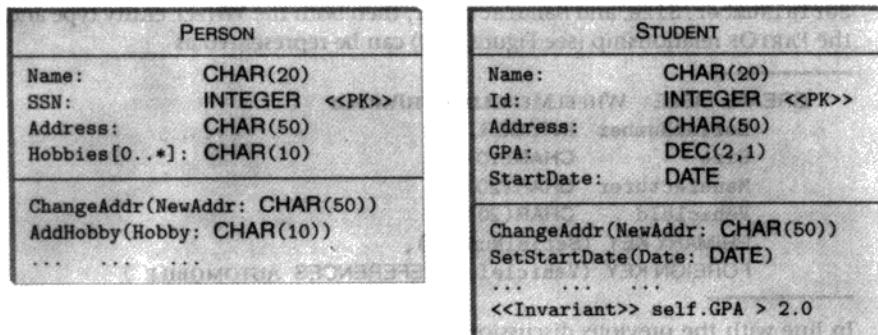


FIGURE 4.15 Examples of UML classes.

### 4.6.1 Representing Entities in UML

UML is an object-oriented modeling language, and, not surprisingly, entities are called **classes** there. Classes are depicted as boxes—as in the E-R case—and appear in UML **class diagrams**. The main visual difference between class diagrams and E-R diagrams is that entity attributes in E-R diagrams are shown inside ovals attached to the box, while in class diagrams attributes appear directly inside the box. UML classes corresponding to the entities PERSON and STUDENT are shown in Figure 4.15. Note that class attributes in UML can be set-valued, as in the E-R model. This is specified by means of a *multiplicity constraint* on the corresponding attribute. In the figure, the attribute Hobbies has the multiplicity [0..\*], meaning that it can have any number of values (including none). Other multiplicities, such as [0..3] or [5..\*], are also possible.

UML classes extend E-R entities in several ways. First, UML classes can include methods that operate on the objects (i.e., entities) that belong to these classes. Some methods that operate on STUDENT objects and on PEOPLE objects are shown in the bottom portion of Figure 4.15. The inclusion of methods allows the designer to specify operations that can be performed on entities that populate each class. This has a particular advantage for object-oriented databases (where the main building blocks are objects rather than relations), but even in relational databases we can use methods to represent transactions that are deemed to be closely associated with particular tables.

Second, UML 2.0 will include the **Object Constraint Language** (or OCL), which can be used to specify certain kinds of constraints directly in the UML diagrams. These constraints can impose restrictions on a single class or on several classes at once, and in this way they are analogous to CHECK and ASSERTION constraints of SQL.

Third, UML has extensibility mechanisms, which can be used to add additional features to the language and make it more suitable for database design. Unfortunately, database-specific extensions are not quite there yet. UML was designed to

model software, not data, and it shows. For instance, while OCL is quite a complex language, it does not have the expressive power of assertions in SQL and thus it is lacking an important functionality required for data modeling. We therefore do not discuss OCL here and instead just show an example of a simple OCL constraint in Figure 4.15. This constraint, indicated with the <<Invariant>> flag, requires that all members of the STUDENT class have a grade point average higher than 2.

At present, UML is lacking even such basic data modeling features as standard ways for specifying primary keys. Not everything is lost, however, since the designers of UML provided for a way to extend the language with new features using **stereotypes**. A **stereotype** is a symbol enclosed in double-angled brackets, such as <<PK>> and <<Invariant>>—see Figure 4.15. Stereotypes have no set meaning within UML. Instead, their meaning is defined by conventions. For example, you and your coworkers may agree that expressions tagged with the stereotype <<Invariant>> represent constraints. Database designers, as a community, may agree that attributes tagged with the stereotype <<PK>> form a primary key of a class. Such sets of conventions within an organization or a community are called **UML profiles**. Clearly, a great deal of data semantics can be specified using stereotypes because one can put any phrase inside the double-angled brackets and then develop a set of conventions around the new stereotype. However, no universally agreed-upon data modeling profile has been adopted by the database community so far. (However, Rational Software [<http://www.rational.com/>], now a division of IBM, has put forward one proposal for a UML data modeling profile.)

## 4.6.2 Representing Relationships in UML

In UML, relationships are called **associations**, and relationship types are known as **association types**. In general, UML diagrams attach more semantics to associations than is normally found in E-R diagrams, and we will examine some of these mechanisms later in this and the next section.

**Associations without attributes.** As in the E-R approach, objects (i.e., entities) that are related to each other by associations (i.e., relationships) may play different roles in those associations. When ambiguity can arise or when greater clarity is desired, the roles can be given explicit names. For binary association types, UML simply uses a line to connect the classes involved in the association. When more than two classes are involved, UML uses a diamond, as in the E-R approach. Figure 4.16 shows the UML version of some of the relationship types previously depicted in Figure 4.2 using the E-R approach.

**Association classes.** Note that the relationships WORKSIN and SOLD have attributes, but the corresponding associations in Figure 4.16 do not, because UML does not offer this facility. Can a designer represent the information contained in those attributes? The answer is provided by **association classes**. An association class is like a regular class, but it is attached to an association in a special way (using a dashed line), which is intended to say that the attributes of the class are intended

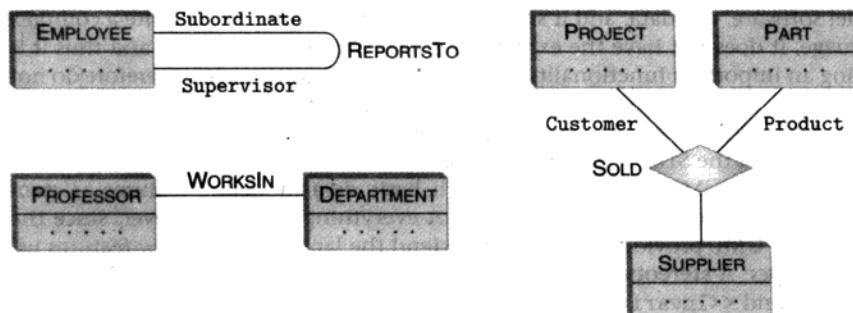


FIGURE 4.16 UML associations.

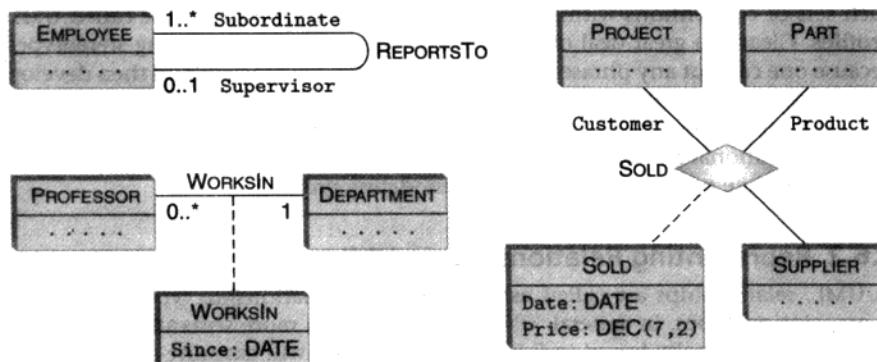


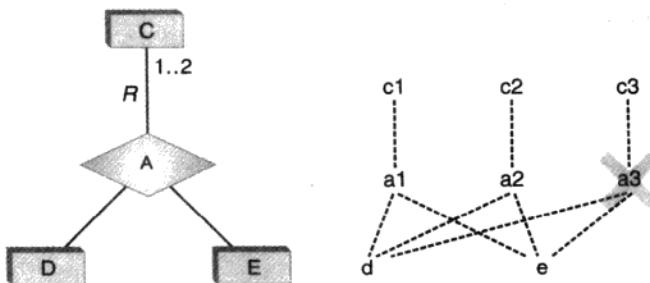
FIGURE 4.17 UML associations with association classes.

to describe the association. The relationships **WORKSIN** and **SOLD** with attached association classes are depicted in Figure 4.17.

**Multiplicity constraints on roles.** Recall that relationship keys can be specified in the E-R diagrams by drawing arrows (see Figure 4.2) or by explicitly writing down the attributes and roles that comprise those keys. In UML, arrows (and even more general constraints) are represented using *multiplicity constraints*.

A **multiplicity constraint** on a role,  $R$ , that connects an association type,  $A$ , with a class,  $C$ , is a range specification of the form  $n..m$  attached to  $R$ , where  $n \geq 0$  is a nonnegative integer and  $m$  is either the  $*$  symbol or an integer  $\geq n$ . The range gives the lower and upper bounds on the number of objects of class  $C$  that can be connected by means of an association of type  $A$  to any given set of objects that are attached to the other ends of the association (one object for each end).

**FIGURE 4.18** The meaning of the multiplicity constraint in UML.



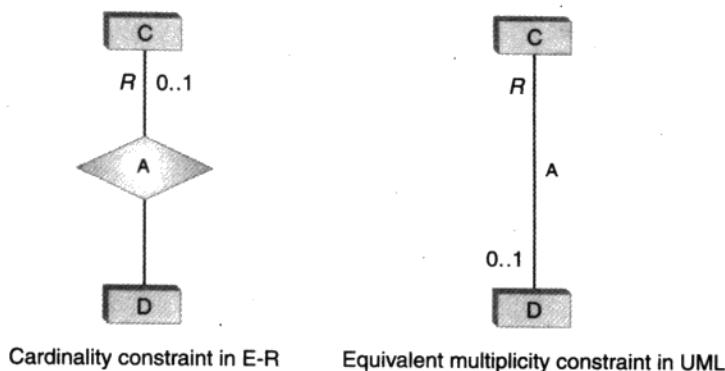
To better understand this concept, take a look at Figure 4.18. In the picture, the class C is connected via the role R to the association type A, and other roles connect the association to classes D and E. The multiplicity constraint on R is 1..2. Therefore, each pair of objects,  $d \in D$  and  $e \in E$ , must be connected by associations of type A to at least one and at most two objects of class C. In the figure, having just two associations,  $a_1$  and  $a_2$ , is legitimate. Adding  $a_3$  would violate the upper bound of the multiplicity constraint because this would allow three objects of type C to be connected to a particular pair of objects ( $d$  and  $e$ ) of classes D and E, respectively. Likewise, it would be a violation of the lower bound of the constraint if  $d$  and  $e$  were not connected to *any* object of class C by an association of type A.

According to this semantics, the range  $5..*$  attached to a role,  $R$ , which connects the association type A to class C, means that at least five C-objects (\* means no upper limit) must participate in role  $R$  in associations of type A with each distinct set of objects attached to the other roles of these associations. The range  $*..*$  means  $0..*$  and the range  $3..3$  means  $3..3$  (i.e., exactly 3). Figure 4.17 shows several uses of the multiplicity constraint in UML. The EMPLOYEE/REPORTSTO example illustrates the assignment of ranges to each role of an association. In this case each range is interpreted separately. The range on the Supervisor role says that an employee can have zero or one supervisor; the range on the Subordinate role says that a supervisor can supervise several employees (but at least one). The ranges in the PROFESSOR/WORKSIN example say that every professor works in exactly one department, but a department can have any number of professors including none.

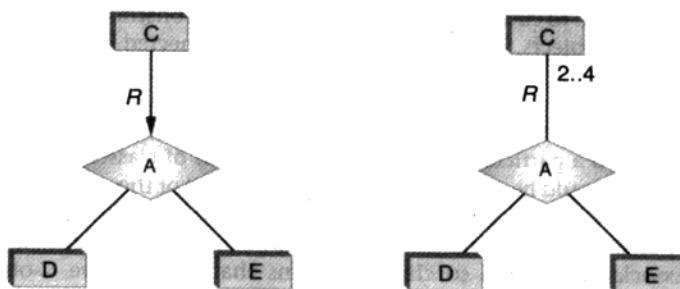
**UML multiplicity vs. E-R cardinality constraints.** On the surface, the notion of multiplicity appears to be similar to cardinality constraints in the E-R approach. However, they are quite different. To see this, compare Figure 4.3 on page 77 with Figure 4.18. In these figures, the diagrams on the left are identical, but their interpretations are different. The valid instance of the UML diagram (on the right side of Figure 4.18) is nothing like the valid instance of the E-R diagram (on the right side of Figure 4.3).

In fact, it can be gleaned from Figures 4.18 and 4.3 that the multiplicity constraint in UML has, in a way, the opposite meaning to that of the cardinality

**FIGURE 4.19** Cardinality vs. multiplicity.



**FIGURE 4.20** Cardinality constraints in E-R that cannot be represented using multiplicity in UML.



constraint in E-R. This is certainly true in the case of binary relationships and associations: Figure 4.19 shows an E-R diagram with a cardinality constraint on a binary relationship and an equivalent UML diagram with the corresponding multiplicity constraint. In both cases, the diagrams say that each entity of type C can be associated with at most one entity of type D. We see that the range specification in UML and E-R appear on the opposite ends of the association/relationship.

For binary relationships, multiplicity and cardinality constraints have equivalent expressive power, although their interpretations are exactly the opposite of each other. For ternary and higher-degree relationships, the difference is greater: the two types of constraints have different, incomparable expressive power. For instance, it is unclear how one can use multiplicity to express the constraints shown in the E-R diagrams in Figure 4.20.

The apparent similarity and the not-so-apparent differences between the notions of cardinality and multiplicity can be an endless source of confusion.

**Key constraints in associations.** Recall that in the E-R model, an arrow that leads from an entity type, C, to a relationship type, A, specifies a key constraint. It says

that an element of C can participate in at most one relationship, and this implies that the primary key of the entity type is also a candidate key of the relationship. In UML, for *binary associations* the same constraint can be specified using multiplicity. If A is an association type connecting classes C and D, then placing the range 0..1 on the role that connects A with D enforces the same constraint: it says that each instance of C can be associated with at most one instance of D. This is illustrated in Figure 4.19. Figure 4.17 shows more examples of the use of multiplicity constraints to specify keys in association types.

As mentioned earlier, multiplicity cannot imitate certain constraints in ternary (and higher-degree) relationships, such as the ones in the E-R diagrams of Figure 4.20. General key constraints cannot be expressed using multiplicity constraints either. For example, a constraint on an association A relating classes C, D, and E that asserts that a particular pair of elements from C and D can participate in at most one association of type A cannot be expressed. However, UML allows just about any text to be placed inside curly braces on the diagram. Such text is intended to be understood as a UML constraint, but of course, it is up to the designer to interpret the meaning of such annotations. For instance, in Figure 4.21 we have annotated the association SOLD with the constraint {Key: Customer, Product; Date}. By itself, this notation means nothing in UML, but the design team and the programming team might adopt internal conventions, which would make such notation meaningful.

**Foreign-key constraints.** As with primary keys, UML does not have a standard way of representing foreign keys. Typically, database designers use the stereotype <<FK>> for that purpose. Figure 4.21 shows examples of the use of this stereotype. This technique, although very common, requires that related attributes in different classes have the same name. Otherwise, it would not be possible to determine which primary keys are referred to by the foreign keys. To overcome this limitation, more expressive stereotypes are needed. For instance, if the identity of a professor is established via the Id attribute (as it has been in all previous examples), then the following stereotype (modeled after SQL) could be used in the WORKSIN association class: <<FK PROFESSOR(Id)>> ProfId: INT. This means that the attribute ProfId in WORKSIN refers to the Id attribute of PROFESSOR.

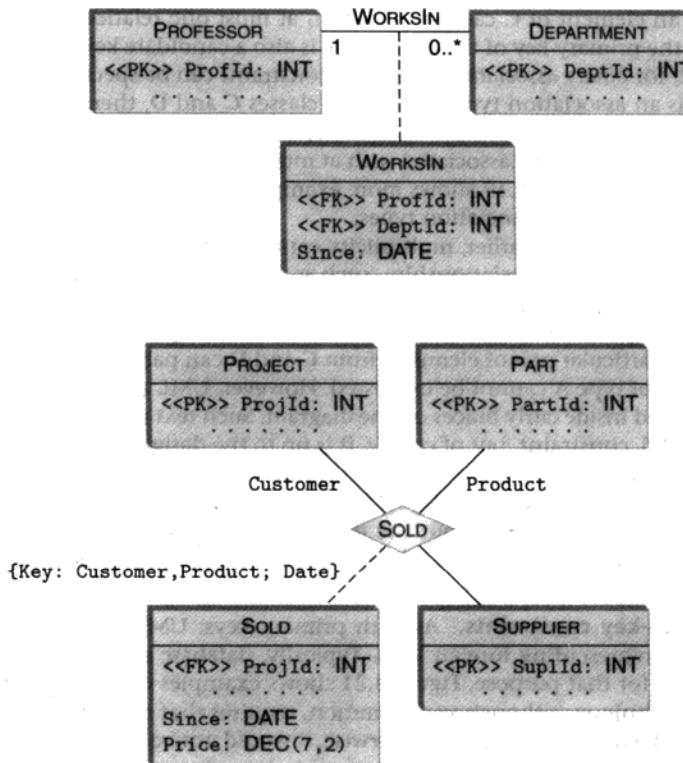
### 4.6.3 Advanced Modeling Concepts in UML

We will now discuss the UML representation of the advanced modeling concepts, which were studied in Section 4.4 in the context of the E-R approach.

**Class hierarchies.** In UML, the ISA relationship is called **generalization**. It is represented as a solid arrow with a large hollow head leading from a subclass to a superclass. An example of a generalization is shown in Figure 4.22.

As in E-R diagrams, several ISA/generalization relationships can be combined, as depicted on the right side of the figure. Covering and disjointness constraints

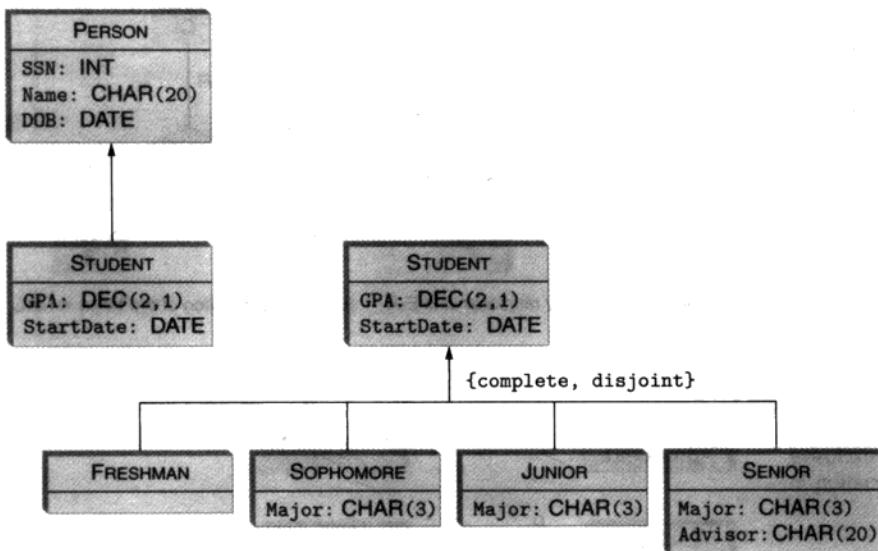
**FIGURE 4.21** Foreign keys in UML.



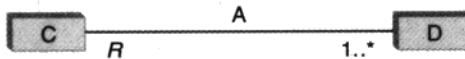
are noted directly on the UML diagrams. In the figure, the covering constraint (which says that any student must belong to one of the four categories: FRESHMAN, SOPHOMORE, etc.) is indicated with the keyword “complete,” and the disjointness constraint with the keyword “disjoint.” They are written inside curly braces, as required by the UML conventions for constraints.

**Participation constraints.** It might seem natural to try to model participation constraints using multiplicity. However, it turns out that the problem is not so simple. Participation in binary association types can, indeed, be modeled using multiplicity constraints, but this cannot be done for associations of higher degree.

Recall the duality principle for binary relationships in E-R and UML shown in Figure 4.19. It says that any cardinality constraint, expressed as a range  $n \dots m$  on role  $R$  of the relationship  $A$  in E-R can be equivalently represented in UML by imposing the same range on the opposite end of the association  $A$ . Since participation constraints in E-R can be specified using the range  $1 \dots *$ , as discussed earlier, expressing the same



**FIGURE 4.22** IsA (or generalization) hierarchies in UML.

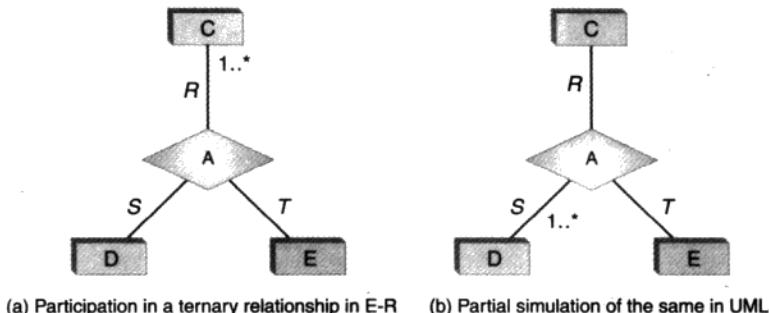


**FIGURE 4.23** UML representation of the participation constraint for class **C** in binary association type **A**.

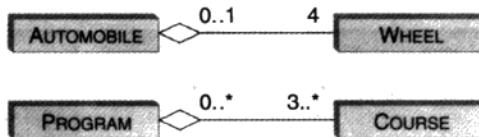
constraints in UML should be obvious. Figure 4.23 shows a UML diagram where class **C** participates in a binary association **A**.

For ternary and other associations, the issue is more involved. One might think that the duality principle can work here as well, and it should be possible to represent the participation constraint in E-R in Figure 4.24(a) using the multiplicity constraint in the UML diagram in Figure 4.24(b). However, the two constraints are not the same. The E-R participation constraint says: “For every entity  $c \in C$  there are entities  $d \in D$  and  $e \in E$  that participate in a relationship  $a \in A$  with  $c$ .” In contrast, the (UML) multiplicity constraint says: “For every pair of objects  $c \in C$  and  $e \in E$  there is at least one object  $d \in D$  that participates in a relationship  $a \in A$  with  $c$  and  $e$ .” The multiplicity constraint is stronger in some respects and weaker in others. Indeed, unless class **C** is empty, the multiplicity constraint implies that *every* **E**-object should be related to at least one **D** object. The participation constraint does not require this. On the other hand, if class **E** is empty, the multiplicity constraint is vacuously

**FIGURE 4.24** Participation constraints for ternary relationships.



**FIGURE 4.25** Aggregation: non-exclusive part-of association in UML.



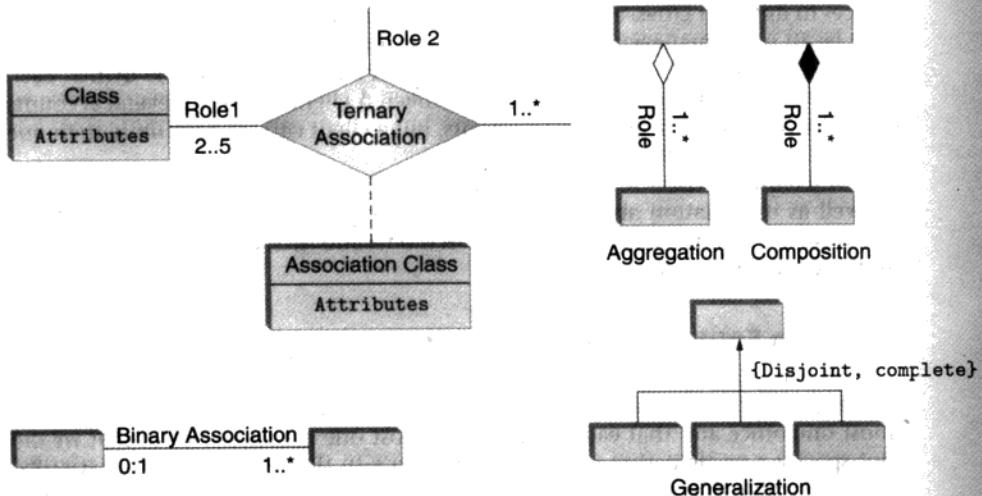
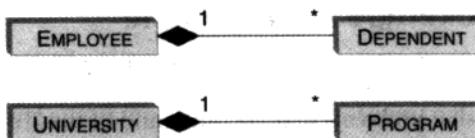
satisfied, while the corresponding E-R diagram does not permit either class D or E to be empty.

Of course, since UML allows any constraint to be specified inside braces, one can simply attach the annotation `{participates}` to the role *R* in the UML diagram. However, this type of annotation requires that all parties to the design understand what this means since this annotation does not have any built-in semantics in UML.

**Part-of relationship.** In UML, the non-exclusive part-of relationship (where sub-parts can have independent existence) is called **aggregation**. UML aggregation has special notation—a line with a hollow diamond—as shown in Figure 4.25. (Recall that E-R does not use special notation for this kind of relationship.) Aggregation in UML is often accompanied by appropriate multiplicity constraints. For instance, the multiplicity constraints in Figure 4.25 indicate that each automobile must have four wheels and that a wheel can be part of at most one automobile. Similarly, the multiplicity constraints between programs in a university and courses indicate that a course can be associated with any number of programs (including none) but any particular program must include at least three courses.

The exclusive part-of relationship is called **composition** in UML; it is viewed as a special kind of aggregation. Compositions are represented using lines with filled diamonds at one end; Figure 4.26 shows two examples of composition analogous to

**FIGURE 4.26** Composition: exclusive part-of association in UML.



**FIGURE 4.27** Summary of the UML notation.

the examples of Figure 4.11 on page 85 for the E-R model. As in those examples, we assume that a **PROGRAM** object is destroyed if its master object of type **UNIVERSITY** is destroyed and that a **DEPENDENT** object is destroyed upon the destruction of the corresponding **EMPLOYEE** object.

Figure 4.27 summarizes the notation used in the UML diagrams.

#### 4.6.4 Translation to SQL

Due to the close correspondence between the basic components of the E-R model and those of UML, translation of UML class diagrams into the relational model is done the same way as in the E-R case. The main problem is how to adequately translate the constraints that might exist in the diagram. In general, this is a complicated matter, which requires the use of **CHECK** and **ASSERTION** constraints. The following sections illustrate some of these issues.

## 4.7 A Brokerage Firm Example

So far we have been using the Student Registration System to illustrate the various issues in database conceptual modeling. In this section, we use a different example to illustrate design problems that are not found in the Student Registration System enterprise.

The Pie-in-the-Sky Securities Corporation (PSSC) is a brokerage firm that buys and sells stocks for its clients. Thus, the main actors are *brokers* and *clients*. PSSC has offices in different cities, and each broker works in one of these offices. A broker can also be an office manager (for the office she works in).

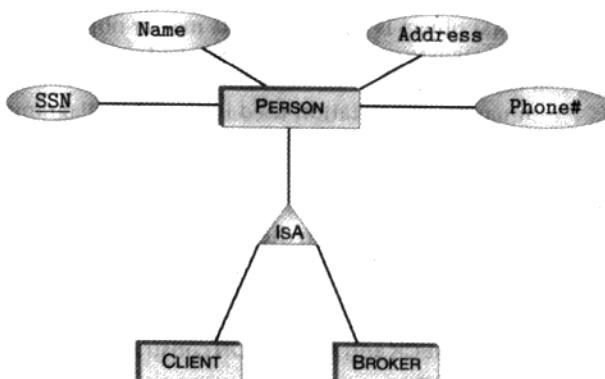
Clients own accounts, and any account can have more than one owner. Each account is also managed by at most one broker. A client can have several accounts and a broker can manage several accounts, but a client cannot have more than one account in a given office.

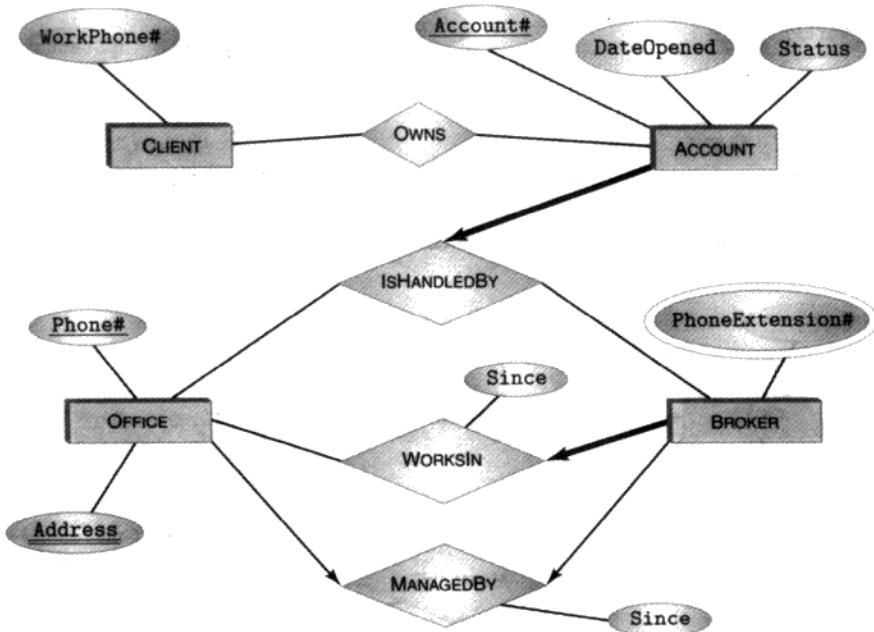
The requirement is to design a database for maintaining the above information as well as information about the trades performed in each account. We will first show two alternative designs using the E-R model and then discuss what is different in the UML representation.

### 4.7.1 An Entity-Relationship Design

The information about brokers and clients is shown with the diagrams in Figures 4.28 and Figure 4.29. Here we make additional assumptions that a broker can manage at most one office and that each office has at most one manager. Notice that we did not specify a participation constraint for OFFICE in the relationship MANAGEDBY, so it is possible that an office might not have a manager (e.g., if the manager quits and the position remains vacant). Since each account must be maintained in exactly one office and by at most one broker, Figure 4.29 shows a participation constraint of entity ACCOUNT in the relationship ISHANDLEDBY by the thick arrow leading

**FIGURE 4.28** The IsA hierarchy of the PSSC enterprise.





**FIGURE 4.29** Client/broker information: first attempt.

from ACCOUNT to ISHANDLEDBY. Thus, {Account} is a key of IsHANDLEDBY. Notice that the attributes that form keys of entity types are underlined and different keys are underlined differently. Thus, for instance, OFFICE has two keys: {Phone#} and {Address}.

Unfortunately, the E-R diagram in Figure 4.29 has problems. First, it requires every account to have a broker. This was not part of our requirements. Second, the requirement that a client cannot have two separate accounts in the same office is not represented in the diagram.

We might try to rectify these problems using the diagram depicted in Figure 4.30. Here we take a slightly different approach and introduce a ternary relation HASACCOUNT with {Client, Office} as a key. Since BROKER is not involved in this relationship, it does not require that an account have a broker. The participation constraint on ACCOUNT says that each account has to be associated with at least one client-office pair and the key of HASACCOUNT guarantees that a client can have at most one account in a given office. In addition, we modify the relationship ISHANDLEDBY so that it involves accounts and brokers only; it does not require that every account has a broker.

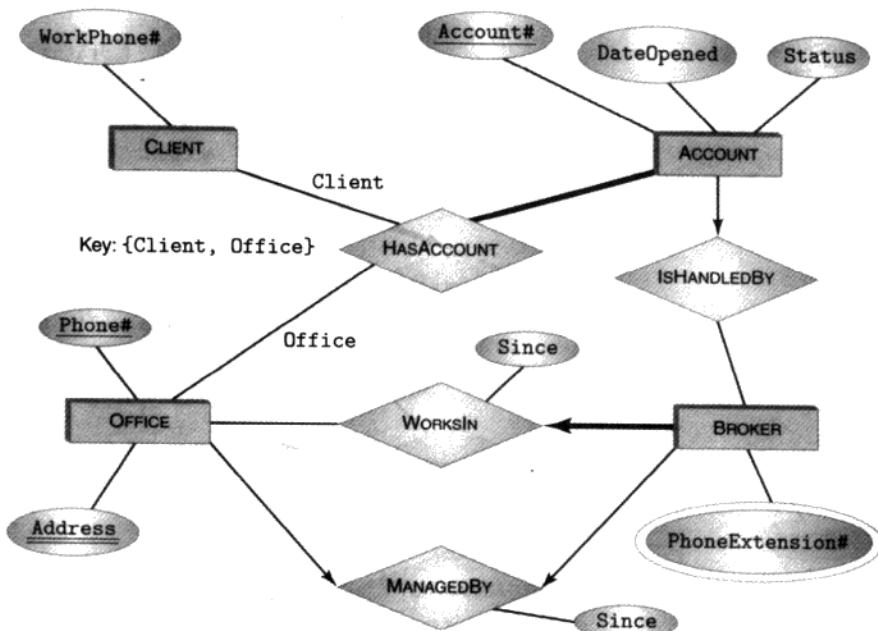
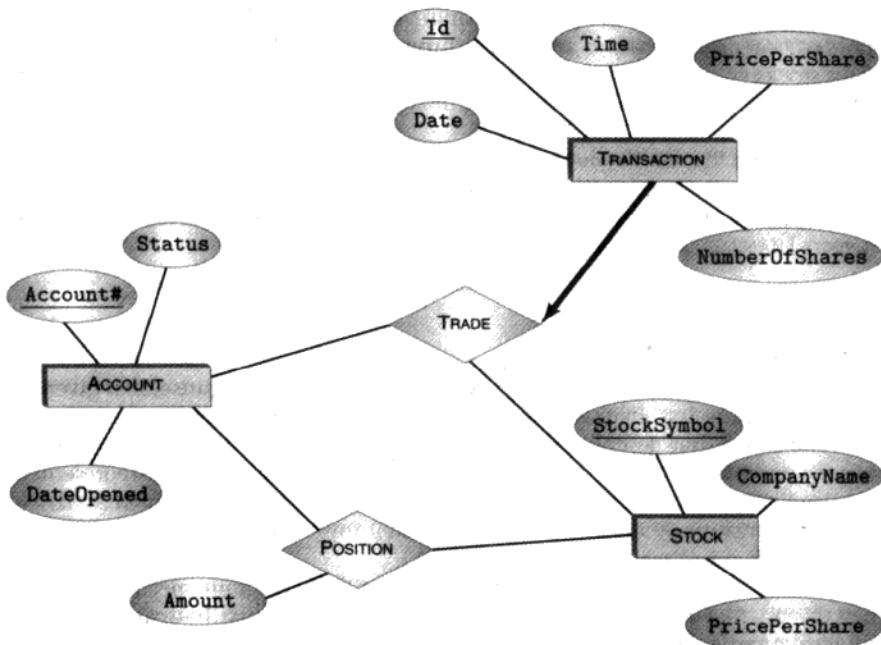


FIGURE 4.30 Client/broker information: second try.

Unfortunately, even this diagram has problems. First, notice that the edge that connects **ACCOUNT** and **HASACCOUNT** does not have an arrow. Such an arrow would have made the role **Account** a key of the relationship **HASACCOUNT**, which contradicts the requirement that an account can have multiple owners. However, our new design introduces a different problem: the constraint that each account is assigned to exactly one office is no longer represented in the diagram. The participation constraint of **ACCOUNT** in **HASACCOUNT** says that each account must be assigned to at least one office (and at least one customer), but nothing here says that such an office must be unique. Furthermore, we cannot solve this problem by adding an arrow to this participation constraint because this would imply that each account has at most one owner.

There is one more problem with our new design (which, in fact, was also present in our original design in Figure 4.29). Suppose that we have the following relationships:

- $(\text{Client1}, \text{Acct1}, \text{Office1}) \in \text{HASACCOUNT}$
- $(\text{Acct1}, \text{Broker1}) \in \text{ISHANDLEDBY}$
- $(\text{Broker1}, \text{Office2}) \in \text{WORKSIN}$



**FIGURE 4.31** Trading information in the PSSC enterprise.

What is there to ensure that `Office1` and `Office2` are the same (i.e., `Account1`'s office is the same as that of the broker who manages `Account1`)? This last problem is known as a **navigation trap**: starting with a given entity, `Office1`, and moving along the triangle formed by the three relationships `HASACCOUNT`, `ISHANDEDBY`, and `WORKSIN`, we might end up with a different entity, `Office2`, of the same type. Navigation traps of this kind are particularly difficult to avoid in the E-R model because doing so requires the use of participation constraints in combination with *functional dependencies* (introduced in Section 6.3), but these constraints are supported by the E-R model only in a very limited way: as keys and participation constraints.

Note that we can avoid the navigation trap by removing the relationship `HASACCOUNT` completely and reintroducing the `OWNS` relationship between clients and accounts. However, this brings back the problem that the constraint that a client cannot have more than one account in any given office is no longer represented.

After these vain attempts to achieve a perfect design for this part of the database, we now turn our attention to the part that deals with stock trading. On a bigger canvas, Figures 4.30 and 4.31 would be connected through the entity type `ACCOUNT`.

Trading information is specified using three entities: ACCOUNT, STOCK, and TRANSACTION. These entities are linked through the relationship POSITION, which relates stocks with the accounts they are held in, and the relationship TRADE, which represents the actual buying and selling of stocks. This information is depicted in Figure 4.31.

Notice that the role **Transaction** is a key of the relationship TRADE and that every transaction must be involved in some TRADE relationship. These constraints are expressed using a thick arrow, which states that there is a one-to-one correspondence between TRANSACTION entities and TRADE relationships.

### 4.7.2 A UML Design\*

The UML diagram for the part of the PSSC enterprise shown in Figure 4.30 is given in Figure 4.32. Let us first acknowledge some obvious differences. Attributes are shown inside the boxes that describe classes, primary and candidate keys are represented using the stereotypes <<PK>> and <<UNIQUE>>, arrows and thick lines are represented by multiplicity constraints, and attributes attached to relationships (such as **Since** in the relationships WORKSIN and MANAGEDBY) now require separate association classes, which are attached with dotted lines.

Focusing on the multiplicity constraints, the UML diagram specifies, among other things, that there is no limit on the number of clients that can be associated with a particular account and office, that each account is handled by exactly one broker, and that an arbitrary number of brokers can work in each office. The multiplicity constraint assigned to the **Account** role of HASACCOUNT asserts that a particular client-office pair can be associated with an arbitrary number of accounts, but this problem is redeemed by the key attached to the association, which guarantees that any given client-office pair can be associated with at most one account through the HASACCOUNT association.

Recall that the E-R representation in Figure 4.30 did not capture the intended semantics completely because the constraint that an account can be associated with exactly one office did not follow from that diagram. Instead, this constraint is approximated in Figure 4.30 by the participation constraint of ACCOUNT in HASACCOUNT, which says only that an account can be maintained in at least one office. Since, as we know, participation constraints on ternary relationships cannot be captured in UML, Figure 4.32 approximates the above participation constraint as suggested in Figure 4.24. Interestingly, this approximation brings us closer to expressing the original constraint (that each account is associated with exactly one office) than the participation constraint. Indeed, the multiplicity constraint on the **Office** role of HASACCOUNT now says that any account object *together* with a client object uniquely determines the office. This constraint is weaker than what is required, because it allows the same account to be associated with different offices for different clients. However, the E-R diagram in Figure 4.30 does not capture even this much.

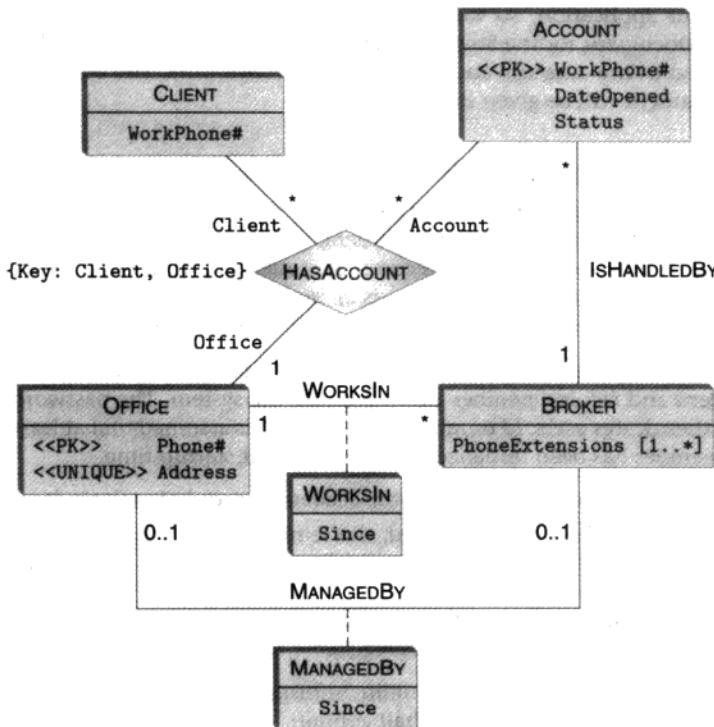


FIGURE 4.32 Client/broker information in UML.

## 4.8 Case Study: A Database Design for the Student Registration System

In this section, we carry out a conceptual design for the database part of the Student Registration System. The outcome of this design—an E-R or a UML diagram and a set of definitions for tables and constraints—is typically included in the Design Document for the entire application. The Design Document itself will be discussed in more detail in Section 15.1.

Before we can start designing an application, we need to write down *precisely* what the system is supposed to do so that we know what it is that we want to design. The document that contains this description is called the **Requirements Document**. In this section, we are interested only in the conceptual design of the

database part of our application, so we will focus on only the relevant parts of the Requirements Document for the Student Registration System: namely, the data items to be included in the database and the associated constraints. The complete Requirements Document will be given in Section 14.2.

### **4.8.1 The Database Part of the Requirements Document**

**I. Information to be contained in the system.** The information to be stored in the system includes four major categories of data: personal information about students and faculty members, academic records of students, information about courses and course offerings, and teaching records of faculty members. Information about classrooms and other auxiliary data is also stored in the system.

- A. *Personal records.* The system shall contain a name, an Id number, and a password for each student and faculty member allowed to use the system. The password and the Id authenticates users. Id numbers are unique. It is assumed that at least one faculty member has been initialized as a valid user at startup time.
- B. *Academic records.* The system shall contain the academic record of each student.
  1. Each course the student has completed, the semester the student took the course, and the grade the student received (all grades are in the set {A, B, C, D, F, I}).
  2. Each course for which the student is enrolled this semester.
  3. Each course for which the student has registered for next semester.
- C. *Course information.* The system shall contain information about the courses offered, and for each course the system shall contain
  1. The course name, the course number (must be unique), the department offering the course, the textbook, and the credit hours.
  2. Whether the course is offered in spring, fall, or both.
  3. The prerequisite courses (there can be an arbitrary number of prerequisites for each course).
  4. The maximum allowed enrollment, the number of students who are enrolled (unspecified if the course is not offered this semester), and the number of students who have registered (unspecified if the course is not offered next semester).
  5. If the course is offered this semester, the days and times at which it is offered; if the course is offered next semester, the days and times at which it will be offered. The possible values shall be selected from a fixed list of weekly slots (e.g., MWF10).
  6. The Id of the instructor teaching the course this semester and next semester (Id unspecified if the course is not offered in the specified semester; it must be specified before the start of the semester in which the course is offered).
  7. The classroom assignment of the course for this semester and next semester (classroom assignment unspecified if the course is not offered in the speci-

#### **4.8 Case Study: A Database Design for the Student Registration System**

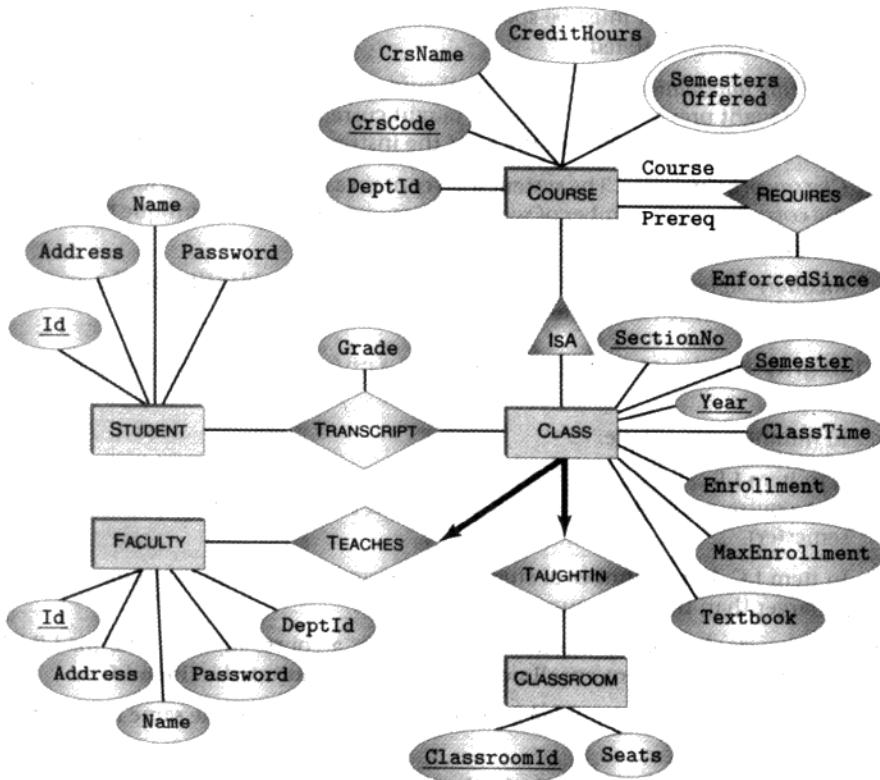
- fied semester; it must be specified before the start of the semester in which the course is offered).
- D. *Teaching information.* The system shall contain a record of all courses that have been taught previously or are being taught currently, including the semester in which they were taught and the Id of the instructor.
  - E. *Classroom information.* The system shall contain a list of classroom identifiers and the corresponding number of seats. A classroom identifier is a unique three-digit integer.
  - F. *Auxiliary information.* The system shall contain the identity of the current semester and the next semester (e.g., F1997, S1998).

**II. Integrity constraints.** The database shall satisfy the following integrity constraints.

- A. Id numbers are unique.
- B. If in item I.B.2 (or I.B.3), a student is listed as enrolled (registered) for a course, that course must be indicated in item I.C.2 as offered this semester (or next semester).
- C. In item I.C.4, the number of students registered or enrolled in a course cannot be larger than the maximum enrollment.
- D. The count of students enrolled (registered) in a course in item I.B.2 (or I.B.3) must equal the current enrollment (registration) indicated in item I.C.4.
- E. An instructor cannot be assigned to two courses taught at the same time in the same semester.
- F. Two courses cannot be taught in the same room at the same time in a given semester.
- G. If a student is enrolled in a course, the corresponding record must indicate that the student has completed all prerequisite courses with a grade of at least C.
- H. A student cannot be registered (enrolled) in two courses taught at the same hour.
- I. A student cannot be registered for more than 20 credits in a given semester.
- J. The room assigned to a course must have at least as many seats as the maximum allowed enrollment for the course.
- K. Once a letter grade of A, B, C, D, or F has been assigned for a course, that grade cannot later be changed to an I.

#### **4.8.2 The Database Design**

We will now present the conceptual design of the database and the corresponding relational representation.



**FIGURE 4.33** An E-R diagram for the Student Registration System.

**A conceptual design diagram.** The first step in the design process is to construct a diagram using the E-R approach or UML. The E-R diagram is shown in Figure 4.33. It is a model of the student registration enterprise as described in the sections of the Requirements Document that we just discussed. Note from the diagram that

- STUDENT is related to CLASS through TRANSCRIPT, which signifies that a student is registered, is enrolled, or has completed a class in some semester.
- FACULTY is related to CLASS through TEACHES, meaning that a faculty member teaches a class in some semester and every class is taught by exactly one faculty member.
- COURSE is related to itself through the relationship REQUIRES; that is, a course can be a prerequisite for another course in some semester. The attribute EnforcedSince specifies the date when the prerequisite was established.

- CLASS is related to CLASSROOM through TAUGHTIN; that is, a class is taught in a classroom in some semester.
- A class (i.e., a particular offering of a course) can use at most one textbook since the attribute Textbook is single-valued in the entity type CLASS.

**UML representation.\*** As far as UML is concerned, the Student Registration System does not introduce any new or interesting issues beyond what was shown in the E-R diagram. (Exercise 4.13 involves redrawing Figure 4.33 using UML conventions.)

**Relational representation.** On the basis of this E-R diagram and the list of integrity constraints given in the Requirements Document, the next step is to produce the schema shown in Figures 4.34 and 4.35. The translation was done in a straightforward manner using the techniques described in this chapter. Note that we did not create tables for the TEACHES and TAUGHTIN relationships: because of the participation and key constraints involving these relationships (i.e., a CLASS entity participates in exactly one TEACHES and one TAUGHTIN relationship), their corresponding tables can *both* be merged with the table for the entity CLASS, as explained in Section 4.5.2. The result of the merge is that the schema of the CLASS table includes the attributes ClassroomId, which identifies the room where the class is taught, and InstructorId of the faculty member who teaches it—see the corresponding CREATE TABLE statement in Figure 4.35.

Note that the simpler integrity constraints among those that are specified in the Requirements Document can and are defined within CREATE TABLE statements with the help of the CHECK clause (specifically the constraints that involve a single relation, such as constraints A, C, E, and F). In the complete schema design, other integrity constraints are defined as separate CREATE ASSERTION statements plus one trigger. However, this part of the design requires SQL constructs that we have not yet discussed, and so we will complete the schema design in Section 15.7 after we cover these constructs.

We selected this particular design for inclusion in the book because it is straightforward. In practice, such a design might be a starting point for a number of enhancements whose goal is to capture more features and increase the efficiency of the final implementation.

**Alternatives.** Let us consider a few possible enhancements and alternatives. Consider course dependencies. One obvious omission in our schema is the *co-requisite* relationship and all of the constraints entailed by it. More subtly, university curricula change all the time: new courses are introduced, old courses are removed, and prerequisite dependencies between courses evolve in time. Thus, the REQUIRES relationship in Figure 4.33 might need two additional attributes, Start and End, to designate the period when the prerequisite relationship is effective. Even more interesting is the possibility that a particular prerequisite relationship might exist at different times. For instance, course A might be a prerequisite for course B between 1985 and 1990 and again between 1999 and the present. (Modeling this situation is left to Exercise 4.11.)

**FIGURE 4.34** A schema for the Student Registration System—Part 1.

```

CREATE TABLE STUDENT (
    Id           CHAR(9),
    Name         CHAR(20) NOT NULL,
    Password     CHAR(10) NOT NULL,
    Address      CHAR(50),
    PRIMARY KEY (Id) )

CREATE TABLE FACULTY (
    Id           CHAR(9),
    Name         CHAR(20) NOT NULL,
    DeptId       CHAR(4) NOT NULL,
    Password     CHAR(10) NOT NULL,
    Address      CHAR(50),
    PRIMARY KEY (Id) )

CREATE TABLE COURSE (
    CrsCode      CHAR(6),
    DeptId       CHAR(4) NOT NULL,
    CrsName      CHAR(20) NOT NULL,
    CreditHours  INTEGER NOT NULL,
    PRIMARY KEY (CrsCode),
    UNIQUE (DeptId, CrsName) )

CREATE TABLE WHENOFFERED (
    CrsCode      CHAR(6),
    Semester     CHAR(6),
    PRIMARY KEY (CrsCode, Semester),
    CHECK (Semester IN ('Spring','Fall')) )

CREATE TABLE CLASSROOM (
    ClassroomId CHAR(3),
    Seats        INTEGER NOT NULL,
    PRIMARY KEY (ClassroomId) )

```

Another interesting enhancement is to account for the possibility that certain highly popular courses might be restricted to certain majors only. In this situation, the E-R diagram and the schema have to include information about the subjects in which each student is majoring, the majors allowed in a particular course (both are set-valued attributes), and a constraint to ensure that the restriction is enforced. (This enhancement is left to Exercise 4.12.)

Enhancements to express more complex requirements are one source of modifications to the proposed design. Another source is the vast range of possible alter-

#### 4.8 Case Study: A Database Design for the Student Registration System

**FIGURE 4.35** A schema for the Student Registration System—Part 2.

```
CREATE TABLE REQUIRES (
    CrsCode      CHAR(6),
    PrereqCrsCode CHAR(6),
    EnforcedSince DATE      NOT NULL,
    PRIMARY KEY (CrsCode, PrereqCrsCode),
    FOREIGN KEY (CrsCode) REFERENCES COURSE(CrsCode),
    FOREIGN KEY (PrereqCrsCode) REFERENCES COURSE(CrsCode) )
```

```
CREATE TABLE CLASS (
    CrsCode      CHAR(6),
    SectionNo    INTEGER,
    Semester     CHAR(6),
    Year         INTEGER,
    Textbook     CHAR(50),
    ClassTime    CHAR(5),
    Enrollment   INTEGER,
    MaxEnrollment INTEGER,
    ClassroomId  CHAR(3),          -- from TAUGHTIN
    InstructorId CHAR(9),          -- from TEACHES
    PRIMARY KEY (CrsCode,SectionNo,Semester,Year),
    CONSTRAINT TIMECONFLICT
        UNIQUE (InstructorId,Semester,Year,ClassTime),
    CONSTRAINT CLASSROOMCONFLICT
        UNIQUE (ClassroomId,Semester,Year,ClassTime),
    CONSTRAINT ENROLLMENT
        CHECK (Enrollment <= MaxEnrollment AND Enrollment >= 0),
    FOREIGN KEY (CrsCode) REFERENCES COURSE(CrsCode),
    FOREIGN KEY (ClassroomId) REFERENCES CLASSROOM(ClassroomId),
    FOREIGN KEY (CrsCode, Semester)
        REFERENCES WHENOFFERED(CrsCode, Semester),
    FOREIGN KEY (InstructorId) REFERENCES FACULTY(Id) )
```

```
CREATE TABLE TRANSCRIPT (
    StudId       CHAR(9),
    CrsCode      CHAR(6),
    SectionNo    INTEGER,
    Semester     CHAR(6),
    Year         INTEGER,
    Grade        CHAR(1),
    PRIMARY KEY (StudId,CrsCode,SectionNo,Semester,Year),
    FOREIGN KEY (StudId) REFERENCES STUDENT(Id),
    FOREIGN KEY (CrsCode,SectionNo,Semester,Year)
        REFERENCES CLASS(CrsCode,SectionNo,Semester,Year),
    CHECK (Grade IN ('A','B','C','D','F','I') ),
    CHECK (Semester IN ('Spring','Fall') ) )
```

## CHAPTER 4 Conceptual Modeling with E-R Diagrams and UML

native designs, which might have implications for the overall performance of the system. We discuss one such alternative and its implications.

Consider the attribute `SemestersOffered` of entity `COURSE` in Figure 4.33. Because it is a set-valued attribute, we translate it using a separate table, `WHENOFFERED`. We chose this particular design because it makes it easy to express the constraint that the semester in which any particular class is taught must be one of the allowable semesters. For instance, it should not be possible for course `CS305` to be offered only in spring semesters but for a certain class of this course to be taught in fall 2004. However, this should be allowed if `CS305` is offered in both spring and fall semesters.

In our design, this requirement is expressed as a foreign-key constraint attached to table `CLASS`.

---

```
FOREIGN KEY (CrsCode, Semester)
    REFERENCES WHENOFFERED(CrsCode, Semester)
```

---

This constraint says that if a class of a course with code `abc` is offered during a semester, `sem`, then `(abc, sem)` should be a tuple in the relation `WHENOFFERED`; that is, `sem` must be one of the allowed semesters for the course.

Despite the simplicity of this design, one might feel that creating a separate relation for such a trivial purpose is unacceptable overhead. A separate relation requires an extra operation for certain queries and additional storage.<sup>2</sup> An alternative is to define a new SQL domain with three values in it:

---

```
CREATE DOMAIN SEMESTERS CHAR(6)
    CHECK ( VALUE IN ('Spring', 'Fall', 'Both') )
```

---

The set-valued attribute `SemestersOffered` of the entity `COURSE` is now single-valued, but it ranges over the domain `SEMESTERS`. The advantage is that the translation into the relational model is more straightforward and there is no need for the extra relation `WHENOFFERED`. However, it is now more difficult to specify the constraint that a class can be taught only in the semesters when the corresponding course is offered. (Details are left to Exercise 4.14.)

Finally, let us consider the possible alternatives for representing the current and the next semesters, as required in item I.F. In fact, our design has no obvious place for this information. One simple way to tell which semester is current or next is to create a separate relation to store this information. However, this entails that any reference to the current or the next semester would require a database query—an expensive way to obtain such simple information. The right way to do this type of thing is to use the function `CURRENT_DATE` provided by SQL and the function `EXTRACT` to extract particular fields from that date. For instance, the following calls

<sup>2</sup> In our particular case, none of these disadvantages seems to apply: in all likelihood, the relation `WHENOFFERED` will be used to verify the above foreign-key constraint, and having a separate relation for course-semester pairs provides efficient support for such verification.

---

```
EXTRACT(YEAR FROM CURRENT_DATE)
EXTRACT(MONTHS FROM CURRENT_DATE)
```

---

return the numeric values of the current year and month. This should be sufficient to determine whether any given semester is current or next.

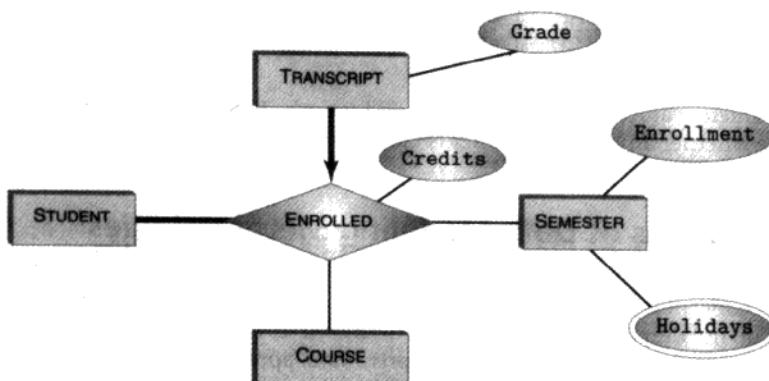
## 4.9 Limitations of Data Modeling Methodologies

We have now seen two case studies where the entity-relationship model and UML were used for conceptual database design. If conceptual design still is not completely clear to you, do not despair. Although we have discussed several concepts that might provide general guidance in organizing enterprise data, applying these concepts in any concrete situation requires a great deal of experience, intuition, and some black magic. There is considerable freedom in deciding whether a particular datum should be an entity (or class), a relationship (or association), or an attribute. Furthermore, even after these issues are settled, the various relationships that exist among entities can be expressed in different ways. This section discusses some of the dilemmas that are often faced by database designers. For concreteness, we use the E-R model in our examples, but the discussion equally applies to UML.

**Entity or attribute?** In Figure 4.8 on page 82, semesters are represented as entities. However, we could as well make TRANSCRIPT into a binary relation and turn SEMESTER into one of its attributes. The obvious question is which representation is best (and in which case).

To some extent, the decision about whether a particular datum should be represented as an entity or an attribute is a matter of taste. Beyond that, the representation might depend on whether the datum has an internal structure of its own. If the datum has no internal data structure, keeping it as a separate entity makes the E-R diagram more complex and, more important, adds an extra relation to your database schema when you convert the diagram into the relational model. On the other hand, if the datum has attributes of its own, it is possible that these attributes cannot be represented if the datum itself is demoted to the status of an attribute.

For instance, in Figure 4.8 the entity type SEMESTER does not have descriptive attributes apart from the identifying semester code, so representing the semester information as an entity appears to be an overkill. However, it is entirely possible that the Requirements Document might state that the following additional information must be available for each semester: *Start\_date*, *End\_date*, *Holidays*, *Enrollment* (which represents total enrollment in all courses during the semester). In such a case, the semester information cannot be an attribute of the TRANSCRIPT relationship as there would then be no place to specify the information about the key dates and the enrollment associated with semesters (e.g., the total enrollment in the university during a semester is not an attribute of any particular transcript relationship).

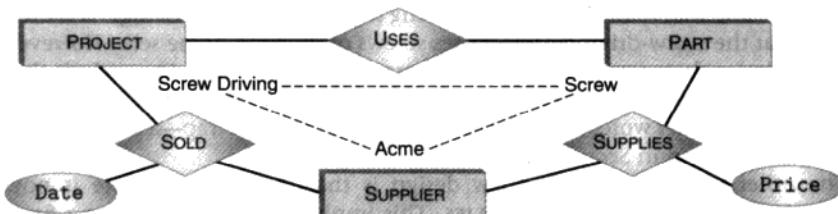


**FIGURE 4.36** An alternative representation of the transcript information.

**Entity or relationship?** Consider once again the diagram in Figure 4.8 on page 82, where we treat transcript records as relationships between STUDENT, COURSE, and SEMESTER entities. An alternative to this design is to represent transcript records as entities and use a new relationship type, ENROLLED, to connect them. This alternative is shown in Figure 4.36. Here we incorporate some of the attributes for the entity SEMESTER, as discussed earlier. We also add an extra attribute, Credits, to the relationship ENROLLED, which means that the same course (e.g., thesis research) can be taken for a variable number of credits. Clearly, the two diagrams represent the same information (except for the extra attributes added to Figure 4.36), but which one is better?

As with the “entity vs. attribute” dilemma, the choice largely depends on your taste. However, a number of points are worth considering. For instance, it is a good idea to keep the total number of entities and relations as small as possible because it is directly related to the number of relations that will result when the E-R diagram is converted to the relational model. Generally, it is not too serious a problem if two relations are lumped together at this stage because the relational design theory presented in Chapter 6 will help identify the relation schemas that must be split. On the other hand, it is much harder to spot the opposite problem: needless decomposition of one relation into two or more.

Coming back to Figure 4.36, we notice that there is a participation constraint for the entity TRANSCRIPT in the relationship type ENROLLED. Moreover, the arrow leading from TRANSCRIPT to ENROLLED indicates that the Transcript role forms a key of the ENROLLED relationship. Therefore, there is a one-to-one correspondence between the relationships of type ENROLLED and the entities of type TRANSCRIPT. This means that relationships of type ENROLLED can be viewed as superfluous because TRANSCRIPT entities can be used instead to relate the entities of types STUDENT, COURSE, and SEMESTER. All that is required (in order not to lose information) is to



**FIGURE 4.37** Replacing the ternary relationship SOLD of Figure 4.2 with three binary relationships.

transfer the descriptive attributes of ENROLLED to TRANSCRIPT after converting the latter into a relationship.

This discussion leads to the following rule:

Consider a relationship type,  $R$ , that relates the entity types  $E_1, \dots, E_n$ , and suppose that  $E_1$  is attached to  $R$  via a role that (by itself) forms a key of  $R$ , and that a participation constraint exists between  $E_1$  and  $R$ . Then it might be possible to collapse  $E_1$  and  $R$  into a new relationship type that relates the entity types  $E_2, \dots, E_n$ .

Note that this rule is only an indication that  $E_1$  can be collapsed into  $R$ , not a guarantee that this is possible or natural. For instance,  $E_1$  might be involved in some other relationship,  $R'$ . In that case, collapsing  $E_1$  into  $R$  leaves an edge that connects two relationship types,  $R$  and  $R'$ , which is not allowed by the construction rules for E-R diagrams. Such is the situation of the BROKER and ACCOUNT entities in Figure 4.30: The above rule suggests that BROKER can be collapsed into WORKSIN, and ACCOUNT can be collapsed into ISHANDLEDBY. However, both BROKER and ACCOUNT are involved in two different relationships, and each such collapse leaves us with a diagram where two relationships, WORKSIN and ISHANDLEDBY or HASACCOUNT and ISHANDLEDBY, are directly connected by an edge. On the other hand, the TRANSACTION entity type in Figure 4.31 *can* be collapsed into the TRADE relationship type.

**Information loss.** We have seen examples where the degree of a relationship might change by demoting an entity to an attribute or by collapsing an entity into a relationship. In all of these cases, however, the transformations obviously preserve the information content of the diagrams. Now we are going to discuss some typical situations where seemingly innocuous transformations cause **information loss**; that is, they lead to diagrams with subtly changed information content.

Consider the PART/SUPPLIER/PROJECT diagram of Figure 4.2, page 74. Some designers do not like ternary relationships, preferring to deal with multiple binary relationships instead. Such a decision might lead to the diagram shown in Figure 4.37.

Although superficially the new diagram seems equivalent to the original, there are several subtle differences. First, the new design introduces a navigation trap of the kind we saw in the stock-trading example: It is possible that a supplier, Acme,

sells "Screw" and that Acme has sold something to project "Screw Driving." It is even possible that the screw-driving project uses screws of the kind Acme sells. However, from the relationships represented in the diagram it is not possible to conclude that it was Acme who sold these screws to the project. All we can tell is that Acme *might* have done so. In other words, we have introduced a navigation trap—a problem that we have already seen in Section 4.7.

The other problem with the new design is that the price attribute is now associated with the relationship SUPPLIES. This implies that a supplier has a fixed price for each item regardless of the project to which that item is sold. In contrast, the original design in Figure 4.2 supports different pricing for different projects. Similarly, the new design allows only one transaction between a particular supplier and project on any given day because each sale is represented as a triple (project, supplier; date) in the SOLD relationship. So there is no way to represent different transactions between the same parties on the same day. The original design, on the other hand, allows several such deals, provided that different parts were involved.

Having realized the problem posed by navigation traps, one might become inclined to use higher-degree relationships whenever possible. For instance, in Figure 4.30 we might want to try eliminating the navigation trap caused by the relationships HASACCOUNT, WORKSIN, and ISHANDLEDBy by collapsing these three relationships into one. However, this transformation introduces more problems than it solves. For instance, if this transformation keeps the arrow that connects BROKER and WORKSIN, we unwittingly introduce the constraint that a broker can have at most one account and at most one client. If we do not keep this arrow, we lose the constraint that each broker is assigned to exactly one office. This transformation also makes it impossible to have brokers who have no accounts and accounts that have no brokers.

**Conceptual design and object databases.** Although we will not discuss object databases until Chapter 16, we briefly mention here that some of the difficult issues involved in translating the conceptual design diagrams into schemas become easier for object databases.

- In Section 4.2, we discussed the issues involved in representing entities with set-valued attributes in a relational database. The objects stored in an object database can have set-valued attributes, so the representation of such entities in the schema of the object database is considerably easier.
- In Section 4.4, we discussed the issues involved in representing the ISA relationship in a relational database. Object databases allow a direct representation of the ISA relationship within the schema, so, again, representation of such relationships is considerably easier.
- UML class diagrams allow methods to be specified along with attributes. These methods can be directly translated into the methods supported by object-oriented databases.

From these examples, it should be apparent that not only is it generally easier to go from conceptual design to object-oriented schemas, but for many applications, object databases support a much more intuitive model of the enterprise than do relational databases.

## BIBLIOGRAPHIC NOTES

The entity-relationship approach was introduced in [Chen 1976]. Since then it has received considerable attention and various extensions have been proposed (see, for example, research papers in [Spaccapietra 1987]). Conceptual design using the E-R model has also been advanced significantly. The reader is referred to [Teorey 1999; Batini et al. 1992; Thalheim 1992] for comprehensive coverage.

The Unified Modeling Language [Booch et al. 1999] was a product of a long line of research on object-oriented modeling and design. Precursors of UML include OMT [Rumbaugh et al. 1991], the methods developed in [Booch 1994], and the methodology for modeling software through use cases [Jacobson 1992]. While UML was primarily motivated by the needs of software engineering, it borrows many ideas from the E-R model and extends it in the direction of object-oriented modeling. In particular, it provides means to model not only the structure of the data but also the behavioral aspects of programs and how large applications are to be deployed in complex computing environments. A succinct introduction to UML can be found in [Fowler and Scott 2003].

A number of tools exist to help the database designer with E-R and UML modeling. These tools guide the user through the process of specifying the diagrams, attributes, constraints, and so forth. When all is done, they map the conceptual model into relational tables. Such tools include *ERwin* from Computer Associates, *ER/Studio* from Embarcadero Technologies, and *Rational Rose* from Rational Software. In addition, DBMS vendors provide their own design tools, such as *Oracle Designer* from Oracle Corporation and *PowerDesigner* from Sybase.

## EXERCISES

- 4.1 Suppose that you decide to convert IsA hierarchies into the relational model by adding a new attribute (such as *Status* in the case of *STUDENT* entities, as described on page 91—the second option for representing IsA hierarchies). What kind of problems exist if subentities are not disjoint (e.g., if a secretary can also be a technician)? What problems exist if the covering constraint does not hold (e.g., if some employees are not classified as either secretary or technician)?
- 4.2 Construct your own example of an E-R or UML diagram whose direct translation into the relational model has an anomaly similar to that of the *PERSON* entity (see the discussion regarding Figure 4.13 on page 86).
- 4.3 Represent the IsA hierarchy in Figure 4.6, page 80, in the relational model. For each IsA relationship discuss your choice of the representation technique. Discuss

the circumstances in which an alternative representation (to the one you have chosen) would be better.

- 4.4 Suppose, in Figure 4.8, the PROFESSOR entity did not participate in the relationship WORKSIN, but the arrow between them was still present. Would it make sense to merge PROFESSOR with WORKSIN during translation into the relational model? What kind of problems can arise here? Are they serious problems?
- 4.5 Translate the brokerage example of Section 4.7 into an SQL schema. Use the necessary SQL machinery to express all constraints specified in the E-R model.
- 4.6 Identify the navigation traps present in the diagram of Figure 4.29, page 107.
- 4.7 Consider the following database schema:
  - SUPPLIER(SName, ItemName, Price)—supplier SName sells item ItemName at Price
  - CUSTOMER(CName, Address)—customer CName lives at Address.
  - ORDER(CName, SName, ItemName, Qty)—customer CName has ordered Qty of item ItemName from supplier SName.
  - ITEM(ItemName, Description)—information about items.
    - (a) Draw the E-R diagram from which the above schema might have been derived. Specify the keys.
    - (b) Suppose now that you want to add the following constraint to this diagram: *Every item is supplied by some supplier.* Modify the diagram to accommodate this constraint. Also show how this new diagram can be translated back to the relational model.
    - (c) Repeat parts (a) and (b) in UML.
- 4.8 Perform conceptual design of the operations of your local community library. The library has books, CDs, tapes, and so forth, which are lent to library patrons. The latter have accounts, addresses, and so forth. If a loaned item is overdue, it accumulates penalty. Some patrons are minors, so they must have sponsoring patrons who are responsible for paying penalties (or replacing a book in case of a loss).
  - a. Use the E-R approach.
  - b. Use UML.
- 4.9 A real estate firm keeps track of the houses for sale and customers looking to buy houses. A house for sale can be *listed* with this firm or with a different one. Being “listed” with a firm means that the house owner has a contract with an agent who works for that firm. Each house on the market has price, address, owner, and a list of features, such as the number of bedrooms, bathrooms, type of heating, appliances, size of garage, and the like. This list can be different for different houses, and some features can be present in some houses but missing in others. Likewise, each customer has preferences that are expressed in the same terms (the number of bedrooms, bathrooms, etc.). Apart from these preferences, customers specify the price range of houses they are interested in. Perform conceptual design for this enterprise.
  - a. Use the E-R approach.
  - b. Use UML.
- 4.10 A supermarket chain is interested in building a decision support system with which they can analyze the sales of different products in different supermarkets

at different times. Each supermarket is in a city, which is in a state, which is in a region. Time can be measured in days, months, quarters, and years. Products have names and categories (produce, canned goods, etc.).

- a. Design an E-R diagram for this application.
  - b. Do the same in UML.
- 4.11** Modify the E-R diagram for the Student Registration System in Figure 4.33 on page 114 to include co-requisite and prerequisite relationships that exist over multiple periods of time. Each period begins in a certain semester and year and ends in a certain semester and year, or it continues into the present. Modify the translation into the relational model appropriately.

- 4.12** Modify the E-R diagram for the Student Registration System in Figure 4.33 on page 114 to include information about the student majors and the majors allowed in courses. A student can have several majors (which are codes of the various programs in the university, such as CSE, ISE, MUS, ECO). A course can also have several admissible majors, or the list of admissible majors can be empty. In the latter case, anyone is admitted into the course. Express the constraint that says that a course with restrictions on majors can have only those students who hold one of the allowed majors.

Alas, in full generality this constraint can be expressed only as an SQL assertion (introduced in Section 3.3) that uses features of Section 5.2 (which we have yet to study). However, it is possible to express this constraint under the following simplifying assumption: when a student registers for a course, she must declare the major toward which the course is going to be taken, and this declared major is checked against the admissible majors.

Modify the relation schema in Figures 4.34 and 4.35 to reflect this simplifying assumption and then express the aforesaid integrity constraint.

- 4.13** Redo the E-R diagram of the Student Registration System (Figure 4.33) in UML.
- 4.14** Make the necessary modifications to the schema of the Student Registration System to reflect the design that uses the SQL domain SEMESTERS, as discussed at the end of Section 4.8. Express the constraint that a class can be taught only during the semesters in which the corresponding course is offered. For instance, if the value of the attribute SemestersOffered for the course CS305 is Both, then the corresponding classes can be taught in the spring and the fall semesters. However, if the value of that attribute is Spring then these classes can be taught only in the spring.
- 4.15** Design an E-R model for the following enterprise. Various organizations make business deals with various other organizations. (For simplicity, let us assume that there are only two parties to each deal.) When negotiating (and signing) a deal, each organization is represented by a lawyer. The same organization can have deals with many other organizations, and it might use different lawyers in each case. Lawyers and organizations have various attributes, like address and name. They also have their own unique attributes, such as specialization and fee, in the case of a lawyer, and budget, in the case of an organization.
- Show how information loss can occur if a relationship of degree higher than two is split into a binary relationship. Discuss the assumption under which such a split does not lead to a loss of information.

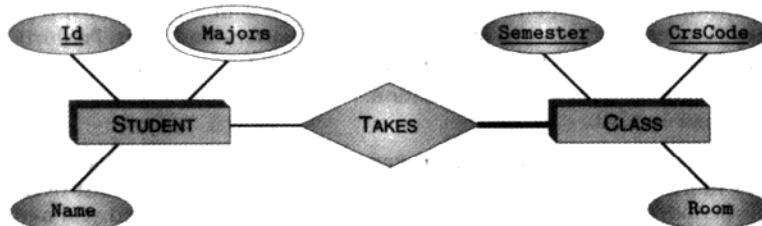


FIGURE 4.38 E-R diagram for Exercise 4.17.

- 4.16 Design an E-R model for the library system described in Exercise 3.15. Do the same with UML.
- 4.17 Consider the E-R diagram depicted in Figure 4.38. Write down the corresponding relational schema using SQL. Include all keys and other applicable constraints.
- 4.18 Consider the partial translation of an E-R participation constraint into UML (shown in Figure 4.24). Show that this is, indeed, an imprecise translation in that there are instances of the database (i.e., collections of objects and associations) that comply with the E-R constraint in Figure 4.24(b) such that it does not satisfy the multiplicity constraint in UML.

# 5

## Relational Algebra and SQL

Now that we know how to create a database, the next step is to learn how to query it to retrieve the information needed for some particular application. Before relational databases, database querying was a dreadful task. To pose even a simple query (by today's standards), one would use a conventional programming language to write a program that could include multiple nested loops, error handling, and boundary condition checking. In addition, the programmer would deal with numerous details of the internal physical schema—in those days, data independence was only on the wish list.

A **database query language** is a special-purpose programming language designed for retrieving information stored in a database. The relational query language in which we are most interested is SQL (Structured Query Language). It is quite different from conventional programming languages. In SQL, you specify the properties of the information to be retrieved but not the detailed algorithm required for retrieval. For example, a query in the Student Registration System might specify retrieval of the names and IDs of all professors who have taught a particular course in a particular semester, but it would not provide a detailed procedure (involving while loops, if statements, pointer variables, etc.) to traverse the various database tables and retrieve the specified names. Thus, SQL is said to be *declarative*, as its queries "declare" what information the answer should contain, not how to compute it. Contrast this to conventional programming languages (e.g., C or Java), which are said to be *procedural* because programs written in them describe the exact actions to be performed to compute the answer.

As with other computer languages, a programmer can design simple queries after only a brief introduction to SQL, but the design of the complex queries needed in real applications requires a more detailed knowledge of the language and its semantics. Therefore, before we introduce SQL we study the *relational algebra*, which is another relational query language that is used by the DBMS as an intermediate language into which SQL statements are translated before they are optimized.

## 5.1 Relational Algebra: Under the Hood of SQL

**Relational algebra** is called an algebra because it is based on a small number of **operators**, which operate on relations (tables). Each operator operates on one or more relations and produces another relation as a result. A query is just an expression involving these operators. The result of the expression is a relation, which is the answer to the query.

While SQL is a declarative language, meaning that it does not specify the algorithm used to process queries, relational algebra is procedural. A relational expression can be viewed as a specification of such an algorithm (although at a much higher level than the algorithms specified using traditional programming languages).

Thus, even when programmers use SQL to specify their queries, DBMSs use relational algebra as an intermediate language for specifying query evaluation algorithms. The DBMS parses the SQL query and translates it into an expression in relational algebra, which usually leads to a rather simplistic, inefficient algorithm. The **query optimizer** then converts this algebraic expression into one that is *equivalent* but that (hopefully) takes less time to execute.<sup>1</sup> On the basis of the optimized algebraic expression it produced, the query optimizer prepares a **query execution plan**, which is then transformed into executable code by the code generator within the DBMS. Because algebraic expressions have precise mathematical semantics, the system can verify that the resulting “optimized” expression is equivalent to the original. The semantics also makes it possible to compare different proposed query evaluation plans. A schematic view of query processing is shown in Figure 5.1.

The relational algebra is the key to understanding the inner workings of a relational DBMS, which in turn is essential in designing SQL queries that can be processed efficiently.

### 5.1.1 Basic Operators

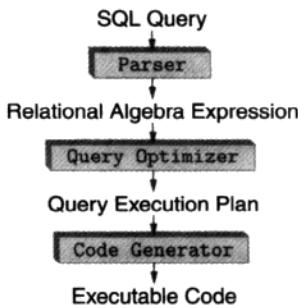
Relational algebra is based on five basic operators:

1. *Select*
2. *Project*
3. *Union*
4. *Set difference*
5. *Cartesian product* (also known as *cross product*)

each of which we consider in turn. In addition, there are three *derived* operators (i.e., they can be represented as expressions involving the basic operators): *intersection*, *division*, and *join*. We also discuss the *renaming* operator, which is useful in conjunction with Cartesian products and joins.

<sup>1</sup>Query optimizers do not really “optimize” (in the sense of producing the *most efficient* query evaluation algorithm) because this is generally an impossible task. Instead, they use heuristics known to produce equivalent expressions that are generally cheaper to evaluate.

**FIGURE 5.1** Schematic view of query processing.



**Select operator.** One of the most frequent operations performed on relations is **selection** of a subset of tuples (i.e., selection of some subset of the rows in a table). For instance, you might want a list of the professors in the CS department. Surely they are all listed in the PROFESSOR relation, but this relation might be large and a manual scan for the tuples of interest might be difficult. Using the select operator, this query can be expressed as

---


$$\sigma_{\text{DeptId} = \text{'CS'}}(\text{PROFESSOR})$$


---

The query reads as “Select all tuples from the PROFESSOR relation that satisfy the condition `DeptId = 'CS'`.”

The general syntax of the select operator is

---


$$\sigma_{\text{selection-condition}}(\text{relation-name})$$


---

We will see later that the argument of the select operator can be more general than simply a name that identifies a relation. It can also be an expression that evaluates to a relation.

The selection condition can have one of the following forms:

- *simple-selection-condition* (explained below)
- *selection-condition AND selection-condition*
- *selection-condition OR selection-condition*
- *NOT (selection-condition)*

A simple selection condition can be any one of the following:

- *relation-attribute oper constant*
- *relation-attribute oper relation-attribute*

where *oper* can be any one of the following comparison operators:  $=$ ,  $\neq$ ,  $>$ ,  $\geq$ ,  $<$ , and  $\leq$ . Each attribute that occurs in a comparison must be one of the attributes in the *relation-name* argument of the selection operator.

It is important to realize that the above syntactic rules *must be followed*, in the same way that you follow the rules of syntax in any other programming language. For instance, a common syntax error is to write a query such as

---


$$\sigma_{\text{ProfId}=\text{PROFESSOR.Id} \text{ AND } \text{PROFESSOR.DeptId}=\text{'CS'}}(\text{TEACHING})$$


---

to list all courses taught by computer science professors. However natural this expression might seem, it is *syntactically incorrect* because the selection condition uses attributes of the PROFESSOR relation whereas the syntax rules permit only the attributes of the TEACHING relation—the relation specified in the argument of the selection.

By itself, the expression  $\sigma_{\text{selection-condition}}(R)$  is meaningless—it is just a string of characters. However, in a concrete database it can have a *value*, which represents the *meaning* of the expression in the context of that database. Thus, we always assume that we are working in the context of some concrete database, which associates a concrete relation instance with each relation name.

Suppose that  $r$  is such a relation instance associated with relation schema  $R$ . We define the *value* of the above expression with respect to  $r$ , denoted  $\sigma_{\text{selection-condition}}(r)$ , to be the relation that consists of the set of all tuples in  $r$  that satisfy *selection-condition*. Thus, the value of the select operator applied to a relation is another relation with the same set of attributes as the original.

For instance, the value of  $\sigma_{\text{DeptId}=\text{'CS'}}(\text{PROFESSOR})$  with respect to the database of Figure 3.5 on page 39 is the relation CSPROF.

CSPROF	Id	Name	DeptId
	101202303	Smyth, John	CS
	555666777	Doe, Mary	CS

It should be clear what it means for a tuple in  $r$  to satisfy a selection condition. For instance, if the condition is  $A > c$ , where  $A$  is an attribute and  $c$  is a constant, a tuple,  $t$ , satisfies the condition if (and only if) the value of attribute  $A$  in  $t$  is greater than  $c$ .<sup>2</sup> When the selection condition is more complex (e.g.,  $cond_1 \text{ AND } cond_2$ ), satisfaction is defined recursively. It is satisfied by  $t$  if and only if  $t$  satisfies both  $cond_1$  and  $cond_2$ . The condition  $cond_1 \text{ OR } cond_2$  is similar, except that  $t$  needs to satisfy only one of the subconditions. Analogously,  $t$  satisfies  $\text{NOT}(cond)$  if  $t$  violates  $cond$ .

For instance, tuple 2 in the relation CSPROF satisfies the complex condition  $\text{Id} > 111222333 \text{ AND NOT } (\text{DeptId} = \text{'EE'})$  because it satisfies both of the following:

- $\text{Id} > 111222333$ , since  $555666777 > 111222333$ .
- $\text{NOT } (\text{DeptId} = \text{'EE'})$ , since ' $CS$ '  $\neq$  ' $EE$ ' and thus the tuple violates the condition  $\text{DeptId} = \text{'EE'}$ .

<sup>2</sup> We assume the existence of some ordering on the domain of  $A$ . In the case of numeric domains, the order is clear; in the case of domains of strings, we assume lexicographic order.

In contrast, tuple 1 in that relation violates the above complex condition because it violates the first term,  $Id > 111222333$ , of that condition (since  $101202303 \not> 111222333$ ).

Here is a more complex selection:

---


$$\sigma_{\text{StudId} \neq 111111111 \text{ AND } (\text{Semester} = 'S1991' \text{ OR Grade} < 'B')}( \text{TRANSCRIPT})$$


---

where the comparison among strings ( $\text{Grade} < 'B'$ ) assumes lexicographic order in which ' $A$ ' < ' $B$ ', etc. The value of this expression in the context of the database of Figure 3.5 is the relation

SUBTRANSCRIPT	StudId	CrsCode	Semester	Grade
	666666666	MGT123	F1994	A
	666666666	EE101	S1991	B
	123454321	CS315	S1997	A
	123454321	CS305	S1996	A
	023456789	CS305	S1996	A

One obvious generalization of the selection condition is to allow conditions of the form  $expression_1$  oper  $expression_2$ , where  $expression$  can be either an arithmetic expression that involves attributes (which act as variables) and constants, or a string expression (e.g., pattern matching, string concatenation).

Examples of such expressions are  $\text{Emp1Salary} > (\text{MngrSalary} * 2)$  and  $(\text{DeptId} + \text{CrsNumber}) \text{ LIKE } \text{CrsCode}$  (here  $+$  denotes string concatenation and  $\text{LIKE}$  denotes pattern matching). The utility of such extended selections is obvious, and they are extensively used in database languages.

**Project operator.** When discussing selection, we often refer to values of certain attributes in a tuple. In relational algebra, such references are very common, so we introduce special notation for them. Let  $A$  denote an attribute of a relation,  $r$ , and let  $t$  be a tuple in  $r$ . Then  $t.A$  denotes the component of tuple  $t$  that corresponds to the attribute  $A$ . For instance, if  $t$  denotes tuple 1 in relation SUBTRANSCRIPT, then  $t.\text{CrsCode}$  is MGT123.

Also, we often need to extract a *subtuple* from a tuple. A **subtuple**,  $t$ , is a sequence of values extracted from  $t$  in accordance with some list of attributes. It is denoted as

$$t.\{A_1, \dots, A_n\}$$

where  $A_1, \dots, A_n$  are attributes.

For instance, if  $t$  is tuple 1 in SUBTRANSCRIPT, then  $t.\{\text{Semester}, \text{CrsCode}\}$  is the tuple ( F1994, MGT123 ). Note that the order of attributes here does not

(and need not) follow the order in which these attributes are listed in the relation **SUBTRANSCRIPT**. Moreover, sometimes it is convenient to allow duplicate attributes in the list. Thus,  $t.\{\text{Semester, CrsCode, Semester}\}$  is  $\{\text{F1994, MGT123, F1994}\}$ .

Now we are ready to define the projection operator, whose general syntax is

---

$\pi_{\text{attribute-list}} (\text{relation-name})$

---

For example, if **R** is a relation name and  $A_1, \dots, A_n$  are *some* (or all) of the attributes in **R**, then  $\pi_{A_1, \dots, A_n}(\textbf{R})$  is called the **projection** of **R** on attributes  $A_1, \dots, A_n$ . In other words, projection picks some subset of the columns in a table. (Sometimes it is convenient to view  $A_1, \dots, A_n$  as a list with possible repetition of attributes, but we will not need this generality here.)

As in the case of selection, the above expression can be assigned a value in the context of a concrete database. Suppose that **r** is a relation instance corresponding to **R** in such a database. Then the *value* of  $\pi_{A_1, \dots, A_n}(\textbf{R})$ , denoted  $\pi_{A_1, \dots, A_n}(\textbf{r})$ , is the set of *all* tuples of the form  $t.\{A_1, \dots, A_n\}$ , where  $t$  ranges over all tuples in **r**.

For instance,  $\pi_{\text{CrsCode, Semester}} (\text{TEACHING})$  is the relation

OFFERINGS	CrsCode	Semester
	MGT123	F1994
	EE101	S1991
	CS305	F1995
	CS315	S1997
	MAT123	S1996
	EE101	F1995
	CS305	S1996
	MAT123	F1997
	MGT123	F1997

Observe that the original **TEACHING** relation of Figure 3.5 (page 40) has 11 tuples, while **OFFERINGS** has only 9. What happened to the other tuples? The answer becomes apparent if we examine the original relation more closely. It is easy to see that tuples 1 and 4 are identical in their **CrsCode** and **Semester** attributes. The only difference between them is in the value of the **ProfId** attribute. The same is true of tuples 10 and 11. Applying the projection operator to tuples 1 and 4 (and to tuples 10 and 11) yields identical tuples because the attribute **ProfId** is eliminated. Relations are sets and so do not allow duplicates; thus, only one copy in each group of identical tuples is kept.

The purpose of the projection operator is to help us focus on the relationships of interest and ignore the attributes that are irrelevant to a particular query. For instance, suppose that we wish to know which courses are offered in which semesters.

The **OFFERINGS** relation shows this information clearly without diluting the answer with data that we did not ask for (i.e., **Prof Id**). Projection also eliminates duplicates, which can save us time on analyzing the answer.

Now that we have seen two relational operators, we can construct **relational expressions** out of them. This is not just an abstract mathematical exercise. Expressions are a general way of constructing queries in relational databases. For instance, the relation **CSPROF**, which contains the tuples corresponding to computer science professors, is the result of applying the selection operator to **PROFESSOR**. We could further request just the names of those professors using the projection operator:  $\pi_{\text{Name}}(\text{CSPROF})$ . The advantage of the algebra is that operators can be combined just as in high-school algebra, so we write

$$\underline{\pi_{\text{Name}}(\sigma_{\text{DeptId} = \text{'CS'}}(\text{PROFESSOR}))}$$

without specifying the intermediate result, **CSPROF**, and without creating a temporary relation.

**Set operations.** The next two operators are the familiar set operators **union** and **set difference**. Clearly, since relations are sets, set operators are applicable to relations. The syntax is  $R \cup S$  and  $R - S$ . If  $r$  and  $s$  are the relations corresponding to  $R$  and  $S$ , then

- The value of  $R \cup S$  is  $r \cup s$ , the set of all tuples that belong to either  $r$  or  $s$ .
- The value of  $R - S$  is  $r - s$ , the set of all tuples in  $r$  that *do not* belong to  $s$ .

We can also use the **intersection** operator,  $R \cap S$  (whose value on  $r$  and  $s$  is, naturally,  $r \cap s$ ), but this operator is not independent of the rest. It can be represented as an expression built out of the basic five operators mentioned at the beginning of this section.

Unfortunately, we are not done yet. While the union of two sets is always a set, we cannot say the same about arbitrary relations. Consider the union of **PROFESSOR** with **TRANSCRIPT**. One problem is that relations are *tables* where all rows have the same number of items. However, the tuples in the **PROFESSOR** relation have three items each, while the tuples in the **TRANSCRIPT** relation have four. Since they have different arities, the collection of all of these tuples is a set (all right), but it does not constitute a relation.

Even when the arities are the same, in order for the union to be meaningful all items in the corresponding columns must belong to the same domain. Consider the set-theoretic union of **PROFESSOR** and **TEACHING**. If we match the columns, then **Id**, **Name**, and **DeptId** in **PROFESSOR** will correspond to **ProfId**, **CrsCode**, and **Semester**. In the union, the values in the first column are members of the same domain, so no problem there. However, the second and the third columns clearly do not belong to the same domain (e.g., people's names and course codes). Again, the union does not make sense.

CrsCode	Semester
CS305	F1995

$$\begin{aligned} & \pi_{\text{CrsCode}, \text{Semester}}(\sigma_{\text{Grade}='C'}(\text{TRANSCRIPT})) \\ & - \pi_{\text{CrsCode}, \text{Semester}}(\sigma_{\text{CrsCode}='MAT123'}(\text{TEACHING})) \end{aligned}$$

CrsCode	Semester
CS305	F1995
MAT123	S1996
MAT123	F1997

$$\begin{aligned} & \pi_{\text{CrsCode}, \text{Semester}}(\sigma_{\text{Grade}='C'}(\text{TRANSCRIPT})) \\ & \cup \pi_{\text{CrsCode}, \text{Semester}}(\sigma_{\text{CrsCode}='MAT123'}(\text{TEACHING})) \end{aligned}$$

CrsCode	Semester
MAT123	S1996

$$\begin{aligned} & \pi_{\text{CrsCode}, \text{Semester}}(\sigma_{\text{Grade}='C'}(\text{TRANSCRIPT})) \\ & \cap \pi_{\text{CrsCode}, \text{Semester}}(\sigma_{\text{CrsCode}='MAT123'}(\text{TEACHING})) \end{aligned}$$

FIGURE 5.2 Examples of relational expressions that involve set operators.

To overcome these problems, we limit the scope of the union operator and apply it only to *union-compatible* relations. Relations are **union-compatible** if their schemas satisfy the following rules:

- Both relations have the same number of columns.
- The names of the attributes are the same in both relations.
- Attributes with the same name in both relations have the same domain.

We also require union-compatibility for the difference and intersection operators.

**Example 5.1.1 (Complex Relational Expressions).** Figure 5.2 illustrates some non-trivial uses of set operators combined with select and project operators. The first query retrieves all course offerings *other than* MAT123, where some student received the grade C. The second query is a bit contrived but is a good illustration; it yields all course offerings where either somebody got a C or the offered course was MAT123. The third query lists all offerings of MAT123 where somebody got a C.

One interesting aspect of these examples is that the original relations, TRANSCRIPT and TEACHING, are not union-compatible. However, they become compatible after the incompatible attributes are projected out. All expressions in this figure are evaluated in the context of our running example of Figure 3.5 on page 39. ■

Id	Name
111223344	Smith, Mary
023456789	Simpson, Homer
987654321	Simpson, Bart

A subset of  $\pi_{\text{Id}, \text{Name}}(\text{STUDENT})$

Id	DeptId
555666777	CS
101202303	CS

A subset of  $\pi_{\text{Id}, \text{DeptId}}(\text{PROFESSOR})$

STUDENT.Id	Name	PROFESSOR.Id	DeptId
111223344	Smith, Mary	555666777	CS
111223344	Smith, Mary	101202303	CS
023456789	Simpson, Homer	555666777	CS
023456789	Simpson, Homer	101202303	CS
987654321	Simpson, Bart	555666777	CS
987654321	Simpson, Bart	101202303	CS

Their Cartesian product

FIGURE 5.3 Two relations and their Cartesian product.

**The Cartesian product and renaming.** The **Cartesian product** (also known as **cross product**),  $R \times S$ , is close to the cross product operation on sets. If  $r$  and  $s$  are relational instances corresponding to  $R$  and  $S$ , respectively, the *value* of this expression, denoted  $r \times s$ , is the set of all tuples,  $t$ , that can be obtained by concatenation of a tuple  $r \in r$  and a tuple  $s \in s$ .<sup>3</sup>

Figure 5.3 shows a Cartesian product of a subset of  $\pi_{\text{Id}, \text{Name}}(\text{STUDENT})$  and a subset of  $\pi_{\text{Id}, \text{DeptId}}(\text{PROFESSOR})$ . To make it clear which parts of each tuple in the product come from which relation, we have marked the boundary between the parts with a double line.

**Brain Teaser:** What is  $r \times s$  when  $s$  is an empty relation?

<sup>3</sup> A slight difference between the usual set-theoretic cross product operation on relations and the relational cross product operation defined above is that in the former the result is a set of pairs of tuples of the form  $(\langle a, b \rangle, \langle c, d \rangle)$  while in the latter it is a set of concatenated tuples (i.e.,  $\langle a, b, c, d \rangle$ ).

We are now forced to address the problem of attribute naming in the results of the relational expressions, which so far we have conveniently ignored. The relations that are arguments to the algebraic expressions have their schema defined in the system catalog, so the names of their attributes are known. In contrast, the relations produced by evaluating the expressions are created on the fly, and their schema is not explicitly defined. For some operations, such as  $\sigma$  and  $\pi$  (when projection list does not include repeated attributes), this does not present a problem as we can simply reuse the schema of the argument relation. For  $\cup$ ,  $\cap$ , and  $-$ , we do not have a naming problem either because the relations involved in these operations are union-compatible and so have the same schema, which, again, can be reused for the query answer.

The Cartesian product is the first time we must deal with the naming problem. Observe how some attributes in the product relation in Figure 5.3 have mysteriously changed names. This is because the relations involved in the operation,  $\pi_{\text{Id}, \text{Name}}(\text{STUDENT})$  and  $\pi_{\text{Id}, \text{DeptId}}(\text{PROFESSOR})$  have an identically named attribute,  $\text{Id}$ , which would otherwise appear twice in the product. The relational model does not allow different columns to have the same name within the same schema. To overcome this problem, we rename the attributes by prefixing them with the relation of their origin.

In this Cartesian product example, the problem of attribute name clashes was conveniently solved through a simple renaming convention, which we will continue to use in the future whenever possible. Unfortunately, this convention does not always work—for instance, it breaks down in the case of a cross product of two instances of the same PROFESSOR relation. Rather than trying to invent increasingly complex renaming schemes, we will place the burden on the programmer, who now becomes responsible for the renaming. To this end, we introduce the **renaming operator**, which does not belong to the core of the algebra and has no standard notation. We choose the following simple notation:

$$\text{expression}[A_1, \dots, A_n]$$

where *expression* is an expression in relational algebra and  $A_1, \dots, A_n$  is a list of names to be used for the attributes in the result of that expression.

We assume that  $n$  represents the number of columns in the result of the expression. Moreover, we assume that there is some standard order of columns in the relation produced by evaluating the expression. For example, in the results of  $\pi$ ,  $\sigma$ ,  $\cup$ ,  $\cap$ , and  $-$ , the order is the same as that in which the attributes are listed in the schema of the argument relations. For  $R \times S$ , the attributes should be listed as in Figure 5.3. The attributes of  $R$  followed by the attributes of  $S$ . For instance,

---


$$(\pi_{\text{Id}, \text{Name}}(\text{STUDENT}) \times \pi_{\text{Id}, \text{DeptId}}(\text{PROFESSOR}))[\text{StudId}, \text{StudName}, \text{ProfId}, \text{ProfDept}]$$


---

renames the attributes of the product relation in Figure 5.3 to  $\text{StudId}$ ,  $\text{StudName}$ ,  $\text{ProfId}$ ,  $\text{ProfDept}$  from left to right. In particular, the two occurrences of  $\text{Id}$  are renamed  $\text{StudId}$  and  $\text{ProfId}$ , respectively.

The renaming operator can also be applied to subexpressions. The following example is similar to the previous one except that we renamed the attributes of the PROFESSOR relation before applying other operators.

---


$$\pi_{\text{Id}, \text{Name}}(\text{STUDENT}) \times \pi_{\text{ProfId}, \text{ProfDept}}(\text{PROFESSOR} [\text{ProfId}, \text{ProfName}, \text{ProfDept}])$$


---

The result is the same as before, but the names of the attributes (from left to right) are now `Id`, `Name`, `ProfId`, `ProfDept`. Also note that we had to change the attributes in the rightmost  $\pi$  operator because the attributes in the argument relation were changed as a result of the renaming.

The Cartesian product holds the distinction of being the most computationally expensive operator in the whole of relational algebra. Consider  $R \times S$ , and suppose that  $R$  has  $n$  tuples and  $S$  has  $m$  tuples. The Cartesian product has  $n \times m$  tuples. In addition, each tuple in the product is larger in size. For concreteness, let both  $R$  and  $S$  have 1,000 tuples with 100 bytes per tuple. Then  $R \times S$  has 1,000,000 tuples with 200 bytes per tuple. Thus, while the total size of the original relations is 200 kilobytes, the product has 200 megabytes. The cost of just writing out such a relation on disk can be prohibitive. This is just a small example. We will soon see that it is not uncommon for a query to involve three or more relations. Even in the case of four tiny relations of 100 tuples with 100 bytes per tuple, the Cartesian product has 100,000,000 tuples with 400 bytes per tuple—40 gigabytes of data!

In a course on analysis of algorithms, you might have been taught that algorithms with polynomial time complexity are acceptable as long as the degree of the polynomial is not too high. As you can see from the above example, in query processing even quadratic algorithms can be unacceptably expensive if they operate on large volumes of data. Because of the potentially huge costs, query optimizers attempt to avoid cross products if at all possible.

### 5.1.2 Derived Operators

**Joins.** A join of two relations,  $R$  and  $S$ , is an expression of the form

$$R \bowtie_{\text{join-condition}} S$$

The *join condition* is a restricted form of the already familiar selection condition used in the  $\sigma$  operator:

---


$$R.A_1 \text{ oper}_1 S.B_1 \text{ AND } R.A_2 \text{ oper}_2 S.B_2 \text{ AND } \dots \text{ AND } R.A_n \text{ oper}_n S.B_n \quad 5.1$$


---

Here the list  $A_1, \dots, A_n$  is a subset of the attributes of  $R$ , and  $B_1, \dots, B_n$  is a subset of the attributes of  $S$ . Finally,  $\text{oper}_1, \dots, \text{oper}_n$  are the comparison operators  $=, \neq, >$ , and so forth.

These restrictions imply that join conditions can have only the **AND** connective (no ORs or NOTs) and that comparisons between attributes and constants are not allowed.

STUDENT.Id	Name	PROFESSOR.Id	DeptId
111223344	Smith, Mary	555666777	CS
023456789	Simpson, Homer	555666777	CS
023456789	Simpson, Homer	101202303	CS

$$\pi_{\text{Id}, \text{Name}}(\text{STUDENT}) \bowtie_{\text{Id} < \text{Id}} \pi_{\text{Id}, \text{DeptId}}(\text{PROFESSOR})$$

**FIGURE 5.4** Join of relations in Figure 5.3.

Even though we use a new symbol to represent it, join is not a radically new operator. *By definition*, the above join is equivalent to

$$\sigma_{\text{join-condition}'}(\mathbf{R} \times \mathbf{S})$$

However, as joins occur frequently in database queries, they have earned the privilege of having their own symbol.

Notice one subtlety: we use *join-condition'* in  $\sigma_{\text{join-condition}'}$  above rather than *join-condition*, which was used in  $\bowtie_{\text{join-condition}}$  in the definition of the join. This is because  $\mathbf{R}$  and  $\mathbf{S}$  might have identically named attributes, which must be renamed as part of the Cartesian product operation. Thus, *join-condition'* is like *join-condition*, except that it uses the actual renamed attributes of  $\mathbf{R} \times \mathbf{S}$  rather than the qualified attribute names in (5.1) above.

**Example 5.1.2 (Join).** Figure 5.4 is an example of a join of two relations. Note the simplified join condition  $\text{Id} < \text{Id}$ , which we use instead of  $\text{STUDENT.Id} < \text{PROFESSOR.Id}$ . It means that in order to qualify for the join, the value of the  $\text{Id}$  attribute in the  $\text{STUDENT}$  tuple must be less than the value of the  $\text{Id}$  attribute in the  $\text{PROFESSOR}$  tuple. We will continue to use such simplified join conditions when it is clear which attributes come from which relations. ■

Note that the definition of a join involves a Cartesian product as an intermediate step. Therefore, a join is a potentially expensive operation. However, the silver lining is that the Cartesian product is hidden inside the join. In fact, Cartesian products rarely occur on their own in typical database queries. Thus, even though the intermediate result (the product) can potentially be very large, the final result can be manageable, as only a small number of tuples in the product might satisfy the join condition.

For example, the result of the join in Figure 5.4 is half the size of the Cartesian product in Figure 5.3. It is not uncommon for the size of a join to be just a tiny fraction of the size of the corresponding cross product. The tricky part is to compute a join without having to compute the intermediate Cartesian product! This might sound like magic, but there are several ways to accomplish this feat, and query optimizers do so routinely. This subject is discussed in Chapter 10.

The general joins described above are sometimes called **theta-joins** because in antiquity the Greek letter  $\theta$  was used to denote join conditions. While theta-joins

are certainly common in query processing (the query *List all employees who earn more than their managers* involves a theta-join), the more common kind is one where all comparisons are equalities:

$$\underline{\underline{R.A_1 = S.B_1 \text{ AND } \dots \text{ AND } R.A_n = S.B_n}}$$

Joins that utilize such conditions are called **equi-joins**.

Equi-joins are essentially what gives relational databases their “intelligence” because they tie together pieces of disparate information scattered throughout the database. Using equi-joins, programmers can uncover complex relationships hidden in the data with just a few lines of SQL code.

**Example 5.1.3 (More Joins).** Here is how one can find the names of professors who taught a course in the fall of 1994:

$$\underline{\underline{\pi_{\text{Name}} (\text{PROFESSOR} \bowtie_{\text{Id}=\text{ProfId}, \sigma_{\text{Semester}=\text{'F1994}}} \text{TEACHING})}}$$

The inner join lines up the tuples in PROFESSOR against the tuples in TEACHING that describe courses taught by the respective professors in fall 1994. The final projection cuts off the uninteresting attributes. Finding the names of courses and the professors who taught them in the fall of 1995 is almost as easy:

$$\begin{aligned} & \pi_{\text{CrsName}, \text{Name}} ( \\ & \quad ( \text{PROFESSOR} \bowtie_{\text{Id}=\text{ProfId}, \sigma_{\text{Semester}=\text{'F1995}}} \text{TEACHING} ) \\ & \quad \bowtie_{\text{CrsCode}=\text{CrsCode}} \text{ COURSE} \\ & ) \end{aligned}$$

The second query in the above example involves two joins. We placed parentheses around the first join to indicate the order in which the joins are to be computed. However, this was not really necessary because join happens to be an *associative* operation, as can be proved using its definition:

$$\begin{aligned} & R \bowtie_{\text{cond}_1} (S \bowtie_{\text{cond}_2} T) \\ &= \sigma_{\text{cond}_1}(R \times \sigma_{\text{cond}_2}(S \times T)) \quad \text{by definition of } \bowtie \\ &= \sigma_{\text{cond}_1}(\sigma_{\text{cond}_2}(R \times (S \times T))) \quad \text{because } \sigma \text{ and } \times \text{ commute (check!)} \\ &= \sigma_{\text{cond}_2}(\sigma_{\text{cond}_1}(R \times (S \times T))) \quad \text{because two } \sigma \text{'s commute (check!)} \\ &= \sigma_{\text{cond}_2}(\sigma_{\text{cond}_1}((R \times S) \times T)) \quad \text{by associativity of } \times \text{ (check!)} \\ &= (R \bowtie_{\text{cond}_1} S) \bowtie_{\text{cond}_2} T \quad \text{by definition of } \bowtie \end{aligned}$$

The double-join query of Example 5.1.3 illustrates one additional point. Join conditions such as TEACHING.CrsCode = COURSE.CrsCode, which test for equality of attributes with the same name (but in different relations) are quite common. The

main reason for this is that it is considered a good design practice to assign the same name to attributes that denote the same thing but belong to different relations. For instance, the semantics of CrsCode in COURSE, TEACHING, and TRANSCRIPT are the same (which is why we used the same name in all three cases!). Because finding hidden connections in the data often amounts to comparing similar attributes in different relations, the above design practice leads to equi-join conditions that equate identically named attributes.<sup>4</sup>

In fact, this design practice in which the join condition equates *only* identically named attributes yields equi-joins of a special variety. In recognition of their importance, such joins received their very own name: the **natural join**. A natural join actually is a little more than that. First, the join condition equates *all* identically named attributes in the two relations being joined. Second, as the equated attributes really denote the same thing in both relations (as indicated by the identity of their names), there is no reason to keep both of the columns. Thus, one copy is always projected out. In sum, the *natural join* of R and S, denoted  $R \bowtie S$ , is defined by the following relational expression:

$$\pi_{attr-list}(\sigma_{join-cond}(R \times S))$$

where

1. *attr-list* = *attributes*(R)  $\cup$  *attributes*(S); that is, the attribute list used in the project operator contains all the attributes in the union of the argument relations *with duplicate attribute names removed*. Since duplicate attributes are deleted, there is no need to perform attribute renaming.
2. The join condition, *join-cond*, has the form

---


$$\underline{R.A_1 = S.A_1 \text{ AND } \dots \text{ AND } R.A_n = S.A_n}$$


---

where  $\{A_1, \dots, A_n\} = \text{attributes}(R) \cap \text{attributes}(S)$ . That is, it is the list of all attributes that R and S have in common.

Note that the notation for natural joins *omits the join condition* because the condition is implicitly (and uniquely) determined by the names assigned to the attributes of the relations in the join.

A typical example of the use of natural joins is the following query:

---


$$\pi_{\text{StudId}, \text{ProfId}} (\text{TRANSCRIPT} \bowtie \text{TEACHING})$$


---

<sup>4</sup> A natural question is, Why have we used different names for the Id attributes in STUDENT (Id) and TRANSCRIPT (StudId)—in clear violation of the design rule previously mentioned? We did it so that we could squeeze more examples out of a reasonably sized schema. In a well-designed database schema, StudId would be used in both places; likewise, ProfId would be used in both PROFESSOR and TEACHING; or, perhaps, Id would be used in all four places.

which lists all IDs of students who ever took a course along with the IDs of professors who taught them.

To further illustrate the difference between the natural join and the equi-join, it is instructive to compare the following two expressions:

---

$\text{TRANSCRIPT} \bowtie \text{TEACHING}$   
 $\text{TRANSCRIPT} \bowtie_{\text{Cond}} \text{TEACHING}$

---

where the equi-join condition Cond is  $\text{TRANSCRIPT.CrsCode} = \text{TEACHING.CrsCode}$  AND  $\text{TRANSCRIPT.Semester} = \text{TEACHING.Semester}$ . Both expressions are equi-joins, and both use the same join conditions (the natural join uses it implicitly). However, the resulting relations have different sets of attributes.

---

*Natural join:*

$\text{StudId, CrsCode, Semester, Grade, ProfId}$

---

*Equi-join:*

$\text{StudId, TRANSCRIPT.CrsCode, TEACHING.CrsCode,}$   
 $\text{TRANSCRIPT.Semester, TEACHING.Semester, Grade, ProfId}$

---

The two expressions represent essentially the same information. However, the schema of the equi-join has two extra attributes (which are duplicates of other attributes), and the natural join benefits from a simpler attribute-naming convention.

Apart from finding hidden connections in the data, joins can be used for certain counting tasks. Here is how we can find all students who took at least two different courses:

---

```
 $\pi_{\text{StudId}} ($ 
     $\sigma_{\text{CrsCode} \neq \text{CrsCode2}} ($ 
         $\text{TRANSCRIPT} \bowtie$ 
         $\text{TRANSCRIPT}[\text{StudId, CrsCode2, Semester2, Grade2}]$ 
     $)$ 
```

---

One obvious limitation of this technique is that if we want students who had taken fifteen courses, we have to join TRANSCRIPT with itself fifteen times. A better way is to extend the relational algebra with so-called *aggregate* functions, which include the counting operator. We do not pursue this possibility here, but we will return to aggregate functions in the context of SQL in Section 5.2.

The discussion of joins cannot be complete without mentioning that—rather unexpectedly—the intersection operator is a special case of a natural join. Suppose

that  $R$  and  $S$  are union-compatible. It then follows directly from the definitions that  $R \cap S = R \bowtie S$ .

**Brain Teaser:** What is  $R \bowtie S$ , if  $R$  and  $S$  have not even one common attribute?

**Outer joins.** When two relations are joined, tuples that do not match fall by the wayside. The operators *outer join*, *left outer join*, and *right outer join* were introduced for the situations where this particular feature of joins is not wanted.

An **outer join** of two relations  $r$  and  $s$  with join condition  $cond$ , denoted  $r \bowtie_{cond}^{\text{outer}} s$ , is defined as follows. As before, it is a relation over the schema that contains the union of the (possibly renamed) attributes in  $R$  (the schema of  $r$ ) and  $S$  (the schema of  $s$ ). However, the tuples in  $r \bowtie_{cond}^{\text{outer}} s$  consist of three categories:

1. The tuples that appear in the regular join of  $r$  and  $s$ ,  $r \bowtie_{cond} s$ .
2. The tuples of  $r$  that do not join with any tuple in  $s$ . Since these tuples do not have values for the attributes that come from  $S$ , they are padded with NULL over these attributes.
3. The tuples of  $s$  that do not join with any tuple in  $r$ . Again, these tuples are padded with NULL over the attributes of  $R$ .

The outer join is sometimes also called **full outer join**. The **left outer join**,  $r \bowtie_{cond}^{\text{left}} s$ , is like the full outer join, except it does not include the third category of tuples (the tuples from  $S$  that are padded with NULL over  $R$ ). The **right outer join**,  $r \bowtie_{cond}^{\text{right}} s$ , is like the full outer join, except that the second type of tuple is missing. Figure 5.5 illustrates these notions.

Note that the outer joins are not independent operators: they can be expressed using the other relational operators (see Exercise 5.9).

**Example 5.1.4 (Left Outer Join).** Outer joins are useful when the NULL-padded tuples still carry useful information for the task at hand. Suppose we need to compute the average grade for every student (let us assume, for the sake of this example, that the grades are numeric). We could join the STUDENT relation of Figure 3.3 on page 36 with the TRANSCRIPT relation of Figure 3.5 on page 39. Then we could group tuples corresponding to each student Id and compute the average (the relational algebra can be extended with operators that support such computation—see Exercise 5.11).

However, it is easy to see that the student with Id 111223344 has no TRANSCRIPT records. Therefore, we will have no information about the average grade of that student (which is 0). To rectify this problem, we could compute the left outer join instead of the regular join:

---

STUDENT  $\bowtie_{\text{Id}=\text{StudId}}^{\text{left}}$  TRANSCRIPT

---

SupplName	PartNumber	PartNumber	PartName
Acme Inc.	P120	N30	10'' screw
Main St. Hardware	N30	KCL12	2lb hammer
Electronics 2000	RM130	P120	10-ohm resistor

SUPPLIER relation

PartNumber	PartName
N30	10'' screw
KCL12	2lb hammer
P120	10-ohm resistor

PARTS relation

SupplName	PartNumber	PartNumber2	PartName
Acme Inc.	P120	P120	10-ohm resistor
Main St. Hardware	N30	N30	10'' screw
Electronics 2000	RM130	NULL	NULL
NULL	NULL	KCL12	2lb hammer

Full outer join SUPPLIER  $\bowtie_{\text{PartNumber}=\text{PartNumber}}$  <sup>outer</sup> PARTS

SupplName	PartNumber	PartNumber2	PartName
Acme Inc.	P120	P120	10-ohm resistor
Main St. Hardware	N30	N30	10'' screw
Electronics 2000	RM130	NULL	NULL

Left outer join SUPPLIER  $\bowtie_{\text{PartNumber}=\text{PartNumber}}$  <sup>left</sup> PARTS

SupplName	PartNumber	PartNumber2	PartName
Acme Inc.	P120	P120	10-ohm resistor
Main St. Hardware	N30	N30	10'' screw
NULL	NULL	KCL12	2lb hammer

Right outer join SUPPLIER  $\bowtie_{\text{PartNumber}=\text{PartNumber}}$  <sup>right</sup> PARTS

FIGURE 5.5 Outer joins.

Now the tuple with Id 111223344 is part of the result and can take part in the computation of the average grade. In fact, we will see that, in practical languages like SQL, nulls are ignored during the computation of the average, which will give us the desired average grade for our student. ■

**Division operator.** While the join operator brings intelligence to query answering, the division operator holds the distinction of being the most difficult to understand and use correctly.

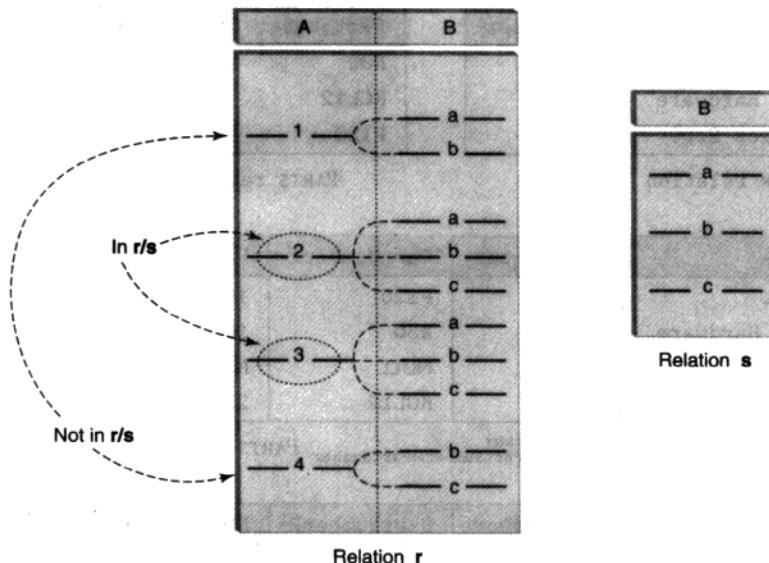


FIGURE 5.6 Division operator.

The **division operator** becomes useful when you feel the urge to find out *which professors taught all courses offered by the Computer Science Department or which students took a class from every professor in the Electrical Engineering Department*. The key here is that we are looking for tuples in one relation that match *all* tuples in another relation.

Here is a precise definition. Let  $R$  be a relation schema with attributes  $A_1, \dots, A_n, B_1, \dots, B_m$  and let  $S$  be a relation schema with attributes  $B_1, \dots, B_m$ . In other words, the set of attributes of  $S$  is a subset of the attributes of  $R$ . The **division** of  $R$  by  $S$  is an expression of the form  $R/S$ . If  $r$  and  $s$  are relation instances corresponding to  $R$  and  $S$  in our database, the **value** of  $R/S$ , denoted  $r/s$ , is a relation over the attributes  $A_1, \dots, A_n$  that consists of all tuples  $\langle a \rangle$  such that, for every tuple  $\langle b \rangle$  in  $s$ , the concatenated tuple  $\langle a, b \rangle$  is in  $r$ . A schematic view of this definition is depicted in Figure 5.6.

An equivalent way to define division is

$$\langle a \rangle \in r/s \text{ if and only if } \{\langle a \rangle\} \times s \subseteq r$$

Here  $\{\langle a \rangle\}$  denotes a relation that contains a single tuple,  $\langle a \rangle$ . Note that, if we view  $\times$  as multiplication, we can view division as multiplication's inverse, which explains the name for this operation.

**Brain Teaser:** What is  $r/s$ , if  $s$  is empty?

**Example 5.1.5 (Division).** Consider the query *List all courses that have been taught by every computer science professor.* Here, by “every professor,” we mean “every professor recorded in the database.” That is, it is assumed that the database describes all we know about the situation. Not only does every tuple in the database assert a fact about the real-world enterprise being modeled, but there are no additional known facts that could be represented as tuples in the database relations. This is known as the **closed-world assumption** and is implicit in relational query processing.

Figure 5.7 shows three relations. The relation

$$\text{PROFCS} = \pi_{\text{Id}}(\sigma_{\text{DeptId}=\text{'CS'}}(\text{PROFESSOR}))$$

contains the Ids of all known computer science professors. The relation

$$\text{PROFCOURSES} = (\pi_{\text{ProfId}, \text{CrsCode}}(\text{TEACHING})) [\text{Id}, \text{CrsCode}]$$

is a relationship between courses and professors who have taught them (at any time); it contains all known relationships of this kind. (Note that we have applied the renaming operator to make sure that the first attribute of PROFCOURSES has the same name as that of the attribute of PROFCS.) The third relation shows PROFCOURSES/PROFCS, the answer to the query. ■

Observe that before applying the division operator in the above example we carefully projected out some attributes (and renamed some others) to make the division operator applicable. The situation when projection must be applied to the operands of the division operator in order to make the division possible is quite typical. Here is another example.

**Example 5.1.6 (Another Division).** Consider the following query: *Retrieve all students who took a course from every professor who ever taught a course.* In the numerator we need a relation that associates students with professors who instructed them in some course. We can get this by taking the natural join of TRANSCRIPT and TEACHING:

$$\text{STUDPROF} = \text{TRANSCRIPT} \bowtie \text{TEACHING}$$

STUDPROF has attributes StudId, CrsCode, Semester, Grade, ProfId, which is more information than we want, so we eliminate unwanted columns using projection to get the numerator:  $\pi_{\text{StudId}, \text{ProfId}}(\text{STUDPROF})$ .

Because we are interested in students who took a course with *every professor*, we need a relation in the denominator that has a tuple for every professor. Professors

PROFCS	Id
	101202303
	555666777

All computer science professors:  $\pi_{\text{Id}}(\sigma_{\text{DeptId}=\text{'CS'}}(\text{PROFESSOR}))$

PROFCOURSES	Id	CrsCode
	783432188	MGT123
	009406321	MGT123
	121232343	EE101
	555666777	CS305
	101202303	CS315
	900120450	MAT123
	101202303	CS305

Who taught what:  $(\pi_{\text{ProfId}, \text{CrsCode}}(\text{TEACHING})) [\text{Id}, \text{CrsCode}]$

CrsCode
CS305

The answer: ProfCourses/ProfCS

**FIGURE 5.7** The anatomy of a query: Courses taught by every computer science professor.

who have taught a course have their Ids inside the tuples of the TEACHING relation. (We do not want to use PROFESSOR here as this would include tuples for professors who have not done any teaching.) Hence, the denominator is  $\pi_{\text{ProfId}}(\text{TEACHING})$ , and the answer to our query is

---

$\pi_{\text{StudId}, \text{ProfId}}(\text{STUDPROF}) / \pi_{\text{ProfId}}(\text{TEACHING})$

---

**5.2**

**Example 5.1.7 (Complex Division).** Find all students who took all courses that were taught by all computer science professors. Here we need double division:

---

$(\pi_{\text{Id}, \text{Name}}(\text{STUDENT})) [\text{StudId}, \text{Name}] \bowtie$   
 $(\pi_{\text{StudId}, \text{CrsCode}}(\text{TRANSCRIPT}) /$   
 $((\pi_{\text{ProfId}, \text{CrsCode}}(\text{TEACHING})) [\text{Id}, \text{CrsCode}] /$   
 $\pi_{\text{Id}}(\sigma_{\text{DeptId}=\text{'CS'}}(\text{PROFESSOR}))) )$

---

**5.3**

The second division above is our friend from Figure 5.7, which yields all courses taught by every computer science professor. Therefore, the last three lines in the expression define the IDs of all students who took all such courses. The natural join in the first line is then used to obtain the names of these students. We had to rename the attributes of the STUDENT relation before taking the join. ■

We conclude by showing how the division operator can be expressed using other relational operators: projection, set difference, and cross product. Let  $R$  be a relation with attributes  $A$  and  $B$ , and let  $S$  be a relation over a single attribute  $B$ . (The construction, below, works even if  $A$  and  $B$  are disjoint lists of attributes.) The expression  $R/S$  can then be computed without the use of division as follows:

$T_1 = \pi_A(R) \times S$  All possible associations between  $A$ -values in  $R$  and  $B$ -values in  $S$ .

$T_2 = \pi_A(T_1 - R)$  All those  $A$ -values in  $R$  that are **not** associated in  $R$  with every  $B$ -value in  $S$ . These are precisely those  $A$ -values that should **not** be in the answer.

5.4

$T_3 = \pi_A(R) - T_2$  *The answer:* All those  $A$ -values in  $R$  that are associated in  $R$  with all  $B$ -values in  $S$ .

## 5.2 The Query Sublanguage of SQL

SQL is the most widely used relational database language. An initial version was proposed in 1974, and it has been evolving ever since. A widely used version, generally referred to as SQL-92, is a standard of the American National Standards Institute (ANSI). The language continues to evolve and recently SQL:1999 and SQL:2003 have been completed. Like any rapidly changing language whose form is influenced by its many users, SQL has become surprisingly complex. The purpose of this section and the next is to introduce you to some of the complexity of the data manipulation sublanguage of SQL. However, you should be aware that a full treatment of this subject is well worth a book of its own. For example, [Melton and Simon 1992, Date and Darwen 1997] are more complete references to SQL-92; [Gulutzan and Pelzer 1999] describes SQL:1999. The unfortunate reality, however, is that commercial databases do not always adhere to the standards, and a vendor-specific reference is almost always a must for serious application development.

SQL can be used interactively by submitting an SQL statement directly to the DBMS from a terminal. However, particularly in transaction processing systems, SQL statements are usually embedded in a larger program that submits the statements to the DBMS at run time and processes the results. The special considerations that relate to such embedding will be discussed in Chapter 8.

### 5.2.1 Simple SQL Queries

Simple SQL queries are easy to design. Need a list of all professors in the Electrical Engineering (EE) Department? Happy to oblige:

---

```
SELECT P.Name
FROM PROFESSOR P
WHERE P.DeptId = 'EE'
```

---

5.5

Note the symbol `P` here, which is called a **tuple variable**. It ranges over the tuples of the relation `PROFESSOR`.<sup>5</sup> The tuple variable is actually unnecessary in this statement, and, if you recall, we tried to avoid its use in our brief introduction to SQL in Chapter 2. In simple queries, such as above, we could have referred to the attributes simply as `Name` and `DeptId` instead of `P.Name` and `P.DeptId`, since there is no ambiguity as to which relation the attributes come from. In some multi-table queries, we could have used `PROFESSOR.Name` and `PROFESSOR.DeptId` in case of ambiguity.

However, as we shall see shortly, in some situations the use of tuple variables is essential. In fact, *not* using tuple variables in `SELECT` statements is considered poor programming practice, which often leads to subtle mistakes. Therefore, from now on we will be pedantically declaring all tuple variables.

Although this statement is rather simple, it is important to understand operationally how a `SELECT` statement might be evaluated. `SELECT` statements can become very complex, and following an operational flow through the statement is often the only way to figure out what is going on. Of course, different DBMSs might adopt different strategies for evaluating the same statement, but since they all produce the same result it is useful to describe a particularly simple strategy.

**An evaluation strategy for simple queries.** The basic algorithm for evaluating SQL queries can be stated as follows:

**Step 1.** *The `FROM` clause is evaluated.* It produces a table that is the Cartesian product of the tables listed as its arguments. If a table occurs more than once in the `FROM` clause, as in query (5.11) on page 151, then this table occurs as many times in the product.

**Step 2.** *The `WHERE` clause is evaluated.* It takes the table produced in step 1 and processes each row individually. Attribute values from the row are substituted for the attribute names in the condition, and the condition is evaluated. The table produced by the `WHERE` clause contains exactly those rows for which the condition evaluates to true.

<sup>5</sup> Mastering SQL often means stuffing one's head with redundant terminology. For instance, SQL variables are also known as *table aliases*.

**Step 3.** *The SELECT clause is evaluated.* It takes the table produced in step 2 and retains only those columns that are listed as arguments. The resulting table is output by the SELECT statement.

As we discuss new features of the language, we will be adding additional steps to this strategy, but the basic idea remains the same. Each clause produces a table, which is the input to the next clause to be evaluated. Certain steps need not be present in a particular evaluation. For example, step 2 might not have to be evaluated because a SELECT statement does not have to have a WHERE clause. Other steps might be trivial. For example, although every SELECT statement must have a FROM clause, if the clause names only a single relation, the Cartesian product is not required and that relation is simply passed on to step 2.

In view of the above algorithm, the relational algebra equivalent of the SQL query (5.5) is

$$\pi_{\text{Name}}(\sigma_{\text{DeptId} = 'EE'} (\text{PROFESSOR}))$$

**Join queries.** Queries that express a join between two relations follow the same pattern as above. The following query, which returns the list of all professors who taught in fall 1994, involves a join.

---

SELECT	P.Name	5.6
FROM	PROFESSOR P, TEACHING T	
WHERE	P.Id = T.ProfId AND T.Semester = 'F1994'	

---

Note that the tuple variables in this example clarify the meaning of the statement because they identify the table from which each attribute is drawn. In this particular case, however, there is no ambiguity as to where the attributes are coming from and so the use of these variables is just a matter of good practice. If, on the other hand, the Id attribute of PROFESSOR were called ProfId, then we would *have* to use tuple variables to distinguish the two references to Id.

Evaluation of the above statement follows the steps outlined above. In this example, in contrast to query (5.5), processing the FROM clause involves taking a Cartesian product. Take the time to convince yourself that query (5.6) is equivalent to the relational algebra expression

---

π <sub>Name</sub> (PROFESSOR ⋈ <sub>Id=ProfId</sub> σ <sub>Semester='F1994'</sub> (TEACHING))	5.7
---	-----

---

where we split the condition in the WHERE clause into a *join condition* and a *selection condition*. The join condition, Id = ProfId, ensures that related tuples in the two tables are combined. It makes no sense to combine a tuple from PROFESSOR that describes a particular professor with a tuple from TEACHING that describes a course taught by a different professor. Hence, the join condition eliminates garbage. The

selection condition `Semester = 'F1994'`, on the other hand, eliminates tuples that are not relevant to the query.

**The relationship between SQL and relational algebra.** The relational algebra expression

$$\pi_{\text{Name}}(\sigma_{\text{Id}=\text{ProfId} \text{ AND } \text{Semester}=\text{'F1994'}}(\text{PROFESSOR} \times \text{TEACHING}))$$

is equivalent to the expression (5.7) and to the SQL query (5.6). Although this expression leads to one of the least efficient ways of evaluating query (5.6), it is simple, uniform, and, from the syntactic point of view, reflects more closely the corresponding SQL statement. More generally, the query template

---

```
SELECT TargetList
FROM   REL1 V1, ..., RELn Vn
WHERE  Condition
```

---

5.8

is roughly equivalent to the algebraic expression

$$\pi_{\text{TargetList}} \sigma_{\text{Condition}} (\text{REL}_1 \times \dots \times \text{REL}_n)$$


---

5.9

"Roughly" means that we have to transform *Condition* into a relational algebra form. To be more concrete, consider the query *Find the names of the courses taught in fall 1995 together with the names of the professors who taught those courses*. In SQL we get

---

```
SELECT C.CrsName, P.Name
FROM   PROFESSOR P, TEACHING T, COURSE C
WHERE  T.Semester = 'F1995' AND
       P.Id = T.ProfId AND T.CrsCode = C.CrsCode
```

---

5.10

The corresponding algebraic expression in the inefficient, but uniform, form described above is

$$\pi_{\text{CrsName}, \text{Name}}(\sigma_{\text{Condition}}(\text{PROFESSOR} \times \text{TEACHING} \times \text{COURSE}))$$


---

where *Condition* denotes the contents of the WHERE clause (modified to suit relational algebra):

---

```
Id = ProfId AND TEACHING.CrsCode = COURSE.CrsCode AND
Semester = 'F1995'
```

---

**Self-join queries.** Let us return to the query that we considered earlier, *Find all students who took at least two courses*. We expressed this in relational algebra as

---

```

 $\pi_{\text{StudId}} \left( \sigma_{\text{CrsCode} \neq \text{CrsCode2}} \left( \text{TRANSCRIPT} \bowtie_{\text{StudId}=\text{StudId}} \text{TRANSCRIPT} [\text{StudId}, \text{CrsCode2}, \text{Semester2}, \text{Grade2}] \right) \right)$ 

```

---

Observe that we have joined the relation TRANSCRIPT with *itself*. To accomplish this in relational algebra, we have to apply the renaming operator to the second occurrence of TRANSCRIPT. In SQL, we must mention the TRANSCRIPT relation twice in the FROM clause and declare two different variables over this relation. Each variable is meant to represent a distinct occurrence of TRANSCRIPT in the join.

---

```

SELECT T1.StudId
FROM TRANSCRIPT T1, TRANSCRIPT T2
WHERE T1.CrsCode <> T2.CrsCode
      AND T1.StudId = T2.StudId

```

---

**5.11**

The symbol  $<\!\!>$  in this query is SQL's way of saying "not equal." Note that we do need *two distinct tuple variables* to range over TRANSCRIPT; if we were to use just one variable, T, the condition  $T.\text{CrsCode} <\!\!> T.\text{CrsCode}$  would never be satisfied, making the answer to the query the empty relation. Therefore, there is no obvious way to do this query without tuple variables.

**Retrieving distinct answers.** We already know from Chapter 3 that relations are sets, so no duplicate tuples are allowed. However, many relational operators can yield **multisets** (i.e., set-like objects that might contain multiple occurrences of identical elements) as an intermediate result of the computation. For instance, if we chop off the attribute Semester from the instance of the relation TEACHING depicted in Figure 3.5 (page 39), we get a list of tuples that contains duplicate occurrences of (009406321, MGT123) and of other tuples as well. As a consequence, in order for the SQL query

---

```

SELECT T.ProfId, T.CrsCode
FROM TEACHING T

```

---

**5.12**

to return a relation (as required by the relational data model), the query processor must perform an additional scan of the query result in order to eliminate duplicates. In many cases, the application programmer is not willing to pay the price for

duplicate elimination. Hence, the designers of SQL decided that, by default, duplicate tuples are not eliminated unless elimination is explicitly requested using the keyword DISTINCT.

---

```
SELECT DISTINCT T.ProfId, T.CrsCode
FROM TEACHING T
```

5.13

Note that this query is missing the WHERE clause, which is *optional* in SQL. When it is missing, the WHERE condition is assumed to be true regardless of the values of tuple variables in the FROM clause.

While the WHERE clause is optional, the SELECT and the FROM clauses are not.

**Comments.** As with every programming language, the programmer might wish to annotate queries with comments. In SQL, comments are strings that begin with the double minus sign, --, and end with a new line. For instance,

---

```
-- An example of SELECT DISTINCT
SELECT DISTINCT T.ProfId, T.CrsCode
FROM TEACHING T    --Look, no WHERE clause!
```

5.14

**Expressions in the WHERE clause.** So far, the conditions in the WHERE clause have been comparisons of attributes against constants or other attributes. For numeric values, SQL provides the following comparison operators: = (equal), <> (not equal), > (greater than), >= (greater than or equal to), < (less than), and <= (less than or equal to).

All of these operators can be applied to numerals and character strings as well. (Strings can also be compared against patterns using the LIKE operator, which we will describe later.) Strings are compared character-wise, from left to right. For the purposes of this text, we limit our attention to ASCII symbols and assume that the ordering of characters in making comparisons between two strings is determined by the ASCII codes assigned to these characters.

The operands of these comparison operators can be **expressions**, not just single attributes or constants. For numeric values, expressions are composed of the usual operators, \*, +, and the like. For strings, the concatenation operator, ||, can be used. Assume, for instance, an appropriate EMPLOYEE relation with the attributes SSN, BossSSN, LastName, FirstName, and Salary. Then the query

---

```
SELECT E.Id
FROM EMPLOYEE E, EMPLOYEE M
WHERE E.BossSSN = M.SSN AND E.Salary > 2 * M.Salary
AND E.LastName = 'Mc' || E.FirstName
```

---

returns all employees whose salary is more than twice that of their bosses' and whose last names are a concatenation of "Mc" and the first name (e.g., Donald McDonald).

**Expressions and special features in the SELECT clause.** The SELECT clause has a number of special features. In Section 2.2, we noted that the asterisk (\*) represents the list of all attributes of all relations in the FROM clause. Note that when it is used and a relation appears twice (or more) in the FROM clause, its attributes appear twice (or more) as well. For instance, the query

---

```
SELECT *
FROM   EMPLOYEE E, EMPLOYEE M
```

---

is the same as

---

```
SELECT E.SSN, E.BossSSN, E.FirstName, E.LastName, E.Salary,
      M.SSN, M.BossSSN, M.FirstName, M.LastName, M.Salary
FROM   EMPLOYEE E, EMPLOYEE M
```

---

SQL permits expressions in the target list, not only in the WHERE clause (which we have seen so far). This feature is illustrated using the following example.

**Example 5.2.1 (Expressions in Target List).** Suppose that an audit office needs a report on salary gaps between employees and their immediate bosses. This can be accomplished with the following query:

---

```
SELECT E.SSN, M.SSN, M.Salary - E.Salary
FROM   EMPLOYEE E, EMPLOYEE M
WHERE  E.BossSSN = M.SSN
```

---

The noteworthy feature here is an arithmetic expression in the last member of the target list. ■

If the above query is used interactively, most DBMSs would display a table where the first two columns are labeled SSN and the last column has no label at all. Obviously, this is not very satisfactory since it requires the user to remember the meaning of the items in the SELECT clause. To alleviate this problem, SQL allows the programmer to change attribute names and assign names where they do not exist. This is accomplished with the help of the keyword AS.

**Example 5.2.2 (Naming Attributes in the Target List).** For example, we can modify the previous query as follows.

---

```
SELECT E.SSN AS EmplId,
      M.SSN AS MngrId,
      M.Salary - E.Salary AS SalaryGap
FROM   EMPLOYEE E, EMPLOYEE M
WHERE  E.BossSSN = M.SSN
```

---

This query is identical to the previous one in all but the form of the output. While the result of the first query will have unnamed columns, the attributes of the last query will all be named and displayed as `EmplId`, `MngrId`, and `SalaryGap`. ■

**Negation.** Any condition in the `WHERE` clause can be negated with `NOT`. For instance, instead of `T1.CrsCode <> T2.CrsCode` in query (5.11), we could have written `NOT (T1.CrsCode = T2.CrsCode)`. The negated condition need not be atomic—it can consist of an arbitrary number of subconditions connected with `AND` or `OR`, and it can even have nested applications of `NOT`, as in the following example.

---

```
NOT (E.BossSSN = M.SSN AND E.Salary > 2 * M.Salary
     AND NOT (E.LastName = 'Mc'|| E.FirstName))
```

---

### 5.2.2 Set Operations

SQL uses the set-theoretic operators from the relational algebra. Here is a simple query where set-theoretic operators can be used: *Find all professors who are working for the CS or EE departments.*

---

```
(SELECT P.Name
   FROM PROFESSOR P
  WHERE P.DeptId = 'CS' )
UNION
(SELECT P.Name
   FROM PROFESSOR P
  WHERE P.DeptId = 'EE' )
```

---

5.15

The query is self-explanatory. It consists of two subqueries: one retrieving all CS professors and the other retrieving all EE professors. The results are collected into a single relation using the `UNION` operator of the algebra. Note that `UNION` removes duplicates from the result.

While this example illustrates the basic use of set-theoretic operators in SQL, the benefits of using `UNION` here are small, since this query can be rewritten without `UNION` and in a more efficient way:

---

```
SELECT DISTINCT P.Name
   FROM PROFESSOR P
  WHERE P.DeptId = 'CS' OR P.DeptId = 'EE'
```

---

5.16

Note that `DISTINCT` is used here because `UNION` removes duplicate tuples.

Our next example, the query *Find all computer science professors and also all professors who ever taught a computer science course*, is more involved, and the advantages of set-theoretic operators there are more substantial.

Let us assume that all course codes in computer science begin with CS. In designing the query, we need to match patterns against strings (course codes). To verify whether a string matches a pattern, SQL provides the LIKE predicate. For instance, T.CrsCode LIKE 'CS%' verifies that the value of T.CrsCode matches the pattern that starts with CS and can have *zero or more* additional characters. SQL patterns are similar to wildcards in UNIX or DOS, although SQL's arsenal for building patterns is somewhat limited: besides %, there is \_, a symbol that matches an arbitrary *single character*.<sup>6</sup>

Without the UNION operator, CS professors or those who taught a CS course can be found as follows:

---

```
SELECT P.Name
  FROM PROFESSOR P, TEACHING T
 WHERE (P.Id = T.ProfId AND T.CrsCode LIKE 'CS%')
       OR (P.DeptId = 'CS')
```

5.17


---

We see that, as the WHERE condition gets longer and more complicated, it becomes harder to read and understand. With the UNION operator, we can rewrite this query in the following way:

---

```
(SELECT P.Name
  FROM PROFESSOR P, TEACHING T
 WHERE P.Id = T.ProfId AND T.CrsCode LIKE 'CS%')
UNION
(SELECT P.Name
  FROM PROFESSOR P
 WHERE P.DeptId = 'CS')
```

5.18


---

Although this is no more succinct than (5.17), it is more modular and easier to understand.

If we want to change our query so that it retrieves all professors who taught a CS course without being a CS professor, we can easily modify (5.18) by replacing UNION with EXCEPT—the SQL counterpart of the MINUS operator in relational algebra.

<sup>6</sup> Suppose that you need to construct a pattern where the special characters % and \_ stand for themselves. For instance, suppose you need to match all strings that start with \_%. This is possible, albeit cumbersome. You have to declare an escape character and then prefix it to the special character to let SQL know that you want these characters to stand for themselves. For instance, C.Descr LIKE '\\_%\\_\\$' ESCAPE '\' compares the value of C.Descr with a pattern that matches all strings that begin with \_, followed by a pair of arbitrary characters, and terminated with the symbol \$. Here \ is declared as an escape character via the ESCAPE clause and then used to "escape" % and \_.

Changing (5.17) to answer the new query is more complex. The WHERE clause of (5.17) would have to be rewritten as

---

```
P.Id = T.ProfId AND T.CrsCode LIKE 'CS%' AND P.DeptId <> 'CS'
```

---

This is not as modular a change as in the case of (5.18).

**Example 5.2.3 (Set Operators Help Simplify Queries).** Suppose that we need to find all students who took both the transaction processing course, CS315, and the database systems course, CS305. As a first try, we might write the following query:

---

```
SELECT S.Name
FROM STUDENT S, TRANSCRIPT T
WHERE S.StudId = T.StudId AND T.CrsCode = 'CS305'
AND T.CrsCode = 'CS315'
```

---

5.19

On closer examination, however, we discover that this SQL query is not what we need because it requires that there be a tuple in TRANSCRIPT such that T.CrsCode is equal to both CS305 and CS315—an unsatisfiable condition. Thus, the formulation (5.19) illustrates one very common mistake—failure to recognize the need for an additional tuple variable. The correct formulation is

---

```
SELECT S.Name
FROM STUDENT S, TRANSCRIPT T1, TRANSCRIPT T2
WHERE S.StudId = T1.StudId AND T1.CrsCode = 'CS305'
AND S.StudId = T2.StudId AND T2.CrsCode = 'CS315'
```

---

5.20

Observe that we used two distinct tuple variables over the relation TRANSCRIPT to express the fact that student S has taken two different courses.

What does this have to do with the original subject of set-theoretic operators of the relational algebra? It turns out that the INTERSECT operator lets us rewrite (5.20) in a more modular and less error-prone way:

---

```
(SELECT S.Name
FROM STUDENT S, TRANSCRIPT T
WHERE S.StudId = T.StudId AND T.CrsCode = 'CS305')
INTERSECT
(SELECT S.Name
FROM STUDENT S, TRANSCRIPT T
WHERE S.StudId = T.StudId AND T.CrsCode = 'CS315')
```

---

5.21

Notice that here we do not need multiple variables to range over the same relation, which somewhat reduces the risk of error. Instead, we write two simple, essentially similar queries and take the intersection of their results. ■

**Set constructor.** Finally, we mention one related feature, the constructor for building finite sets within SQL queries. The syntax of the set constructor is simple:  $(set\_elem_1, set\_elem_2, \dots, set\_elem_n)$ . The operator IN lets us check if a particular element is within a set. For example, consider query (5.16), which, with the help of the set constructor, can be simplified to

---

```
SELECT P.Name
FROM PROFESSOR P
WHERE P.DeptId IN ('CS', 'EE')
```

---

5.22

Note that if we take query (5.19) and replace its WHERE clause with

---

```
S.StudId = T.StudId AND T.CrsCode IN ('CS305', 'CS315')
```

---

we obtain a query with a meaning different from that of (5.19). This issue is further investigated in Exercise 5.13.

**Negation and infix comparison operators.** Earlier we discussed the NOT operator. For some infix operators, such as LIKE and IN, SQL provides two forms of negation: NOT ( $X$  LIKE  $Y$ ) and, equivalently,  $X$  NOT LIKE  $Y$ . Similarly, one can write  $X$  NOT IN  $Y$  instead of the more cumbersome NOT( $X$  IN  $Y$ ).

### 5.2.3 Nested Queries

SQL would be only half as much fun if it offered only one way to do each task. Consider the query *Select all professors who taught in fall 1994*. One way to say this in SQL was given in (5.6) on page 149, but there is (at least) one other, radically different way. First compute the set of all professors who taught in fall 1994 using a nested subquery; then collect their names and produce the result.

---

```
SELECT P.Name
FROM PROFESSOR P
WHERE P.Id IN
    -- A nested subquery
    (SELECT T.ProfId
     FROM TEACHING T
     WHERE T.Semester = 'F1994')
```

---

Note that in this example the nested subquery is evaluated only once, and then each row of PROFESSOR can be tested in the WHERE clause against the result of that evaluation.

The above example illustrates one way in which nested subqueries can be of help—*increased readability*. However, readability alone is not a sufficient reason for using this facility as most query processors cannot optimize nested subqueries well

enough, and thus indulging in subqueries has a performance penalty. A much more important reason for the existence of nested subqueries is that they increase the expressive power of SQL—some queries simply cannot be formulated in a natural way without them.

Consider the query *List all students who did not take any courses*. In English this query sounds deceptively simple, but in SQL it cannot be done without the nested subquery facility (or the EXCEPT operator—try this alternative on your own).

---

```
SELECT S.Name
  FROM STUDENT S
 WHERE S.Id NOT IN
      -- Students who have taken a course
      (SELECT T.StudId
        FROM TRANSCRIPT T)
```

---

5.23

As a final example, a subquery can be used to extract a scalar value from a table. Suppose that you want to know which employees are paid a higher salary than you. Assuming that your Id is 111111111, you might use a subquery to return your salary from EMPLOYEE as follows:

---

```
SELECT E.Id
  FROM EMPLOYEE E
 WHERE E.Salary >
      (SELECT E1.Salary
        FROM EMPLOYEE E1
       WHERE E1.Id = '111111111')
```

---

5.24

The overall query then finds all employees that earn more.

**Correlated nested subqueries.** Although nested subqueries can sometimes improve readability, on the whole they are one of the most complex, expensive, and error-prone features of SQL. To a large extent, this complexity is due to **query correlation**—the ability to define variables in the outer query and use them in the inner subquery. Nesting and correlation are akin to the notion of begin/end blocks in programming languages and the associated idea of the scope of a variable.

To illustrate, suppose that we need to find student assistants for professors who are scheduled to teach in a forthcoming semester (for definiteness, let us say in fall 2004). For each professor, we compute the set of all courses she will be teaching during that semester and then find students who have taken one of these courses (and so are eligible to assist with them). The list of courses taught by professors can be computed in a nested subquery, and associating professors with students can be done in an outer query. Here is a realization of this plan in SQL:

---

```

SELECT    R.StudId, P.Id, R.CrsCode
FROM      TRANSCRIPT R, PROFESSOR P
WHERE     R.CrsCode IN
          -- Courses taught by P.Id in F2004
          (SELECT T1.CrsCode
           FROM   TEACHING T1
           WHERE T1.ProfId = P.Id AND T1.Semester = 'F2004' )

```

---

Here the scope of the variable `T1` is limited to the subquery. In contrast, the variable `P` is visible in both the outer and inner queries. This variable parameterizes the inner query and correlates its result with the tuples of the outer query. For each value of `P.Id`, the inner query is computed independently *as if P.Id were a constant*. Each time an inner subquery is computed, the value of `R.CrsCode` is checked against the result returned. If this value belongs to the result, an output tuple is formed by the outer `SELECT` query.

Observe that, at a minimum, the inner query must be reevaluated for each row of `PROFESSOR`. This contrasts with uncorrelated nested queries and explains the expense associated with query correlation.

Even though nested queries are a challenge for query optimizers, inexperienced database programmers sometimes abuse them, substituting them for the much simpler `ANDs`, `NOTs`, and the like.

**Example 5.2.4 (Abuse of Query Nesting).** Here is an example of how *not* to write the previous query (even though it is semantically correct):

---

```

SELECT    R.StudId, T.ProfId, R.CrsCode
FROM      TRANSCRIPT R, TEACHING T
WHERE     R.CrsCode IN
          -- Courses taught by T.ProfId in F2004
          (SELECT T1.CrsCode
           FROM   TEACHING T1
           WHERE T1.ProfId = T.ProfId AND
          -- Bad style: unreadable and slow!
           T1.ProfId IN (SELECT T2.ProfId
                         FROM   TEACHING T2
                         WHERE T2.Semester = 'F2004' ) )

```

---

The third level of nesting can be avoided here. Not only is it a performance hit, but it is also much harder to understand compared to (5.25). ■

**The `EXISTS` operator.** It is often necessary to check if a nested subquery returns no answers. For instance, we might wish to *Find all students who never took a computer science course*. A way to approach this problem is to compute the set of all computer

courses taken by a student and then list only those students for whom this set is empty. This can be done with the help of correlated nested subqueries and the EXISTS operator, which returns true if a set is non-empty.

Here is one SQL formulation of this query:

---

```
SELECT S.Id
FROM STUDENT S
WHERE NOT EXISTS (
    -- All CS courses taken by S.Id
    SELECT T.CrsCode
    FROM TRANSCRIPT T
    WHERE T.CrsCode LIKE 'CS%'
        AND T.StudId = S.Id )
```

---

5.26

Once again, the variable S is global with respect to the inner subquery; this subquery is evaluated for each value of S.Id, which is treated as a constant during the evaluation. All values of S.Id for which the inner query has no answers constitute the answer to the outer query.

**Expressing the division operator.** We now show that query nesting can help in expressing the relational division operator. For concreteness, consider the query *List the students who have taken all computer science courses*. We can solve the problem by first computing a single-attribute relation that has a row for each CS course (this is the denominator of the division operator). Then, for each student, we check if the student's transcript contains all of these courses.

To make the idea easier to understand, we first realize our plan assuming the availability of a predicate, CONTAINS, which does not actually exist in SQL. As its name suggests, CONTAINS tests if one set contains another. Then we will show how this predicate is expressed using the SQL operators that do exist.

---

```
SELECT S.Id
FROM STUDENT S
WHERE -- All courses taken by S.Id
      (SELECT R.CrsCode
      FROM TRANSCRIPT R
      WHERE R.StudId = S.Id )
CONTAINS
-- All CS courses
( SELECT C.CrsCode
  FROM COURSE C
  WHERE C.CrsCode LIKE 'CS%' )
```

---

Now, observe that  $A \text{ CONTAINS } B$  is equivalent to  $\text{NOT EXISTS } (B \text{ EXCEPT } A)$ . Therefore, we can rewrite the above query as follows using only the available SQL operators. Clearly, the result is much harder to understand and construct than the query that involves the (alas nonexistent) operator  $\text{CONTAINS}$ .

---

```
SELECT S.Id
FROM STUDENT S
WHERE NOT EXISTS (
    (SELECT C.CrsCode
     FROM COURSE C
     WHERE C.CrsCode LIKE 'CS%' )
EXCEPT
    (SELECT R.CrsCode
     FROM TRANSCRIPT R
     WHERE R.StudId = S.Id ) )
```

---

5.27

The following is an example of an even harder query.

**Example 5.2.5 (Complex Nested Query).** Consider the query *Find the students who took a course from every professor in the CS department*. One possible SQL formulation of this query is

---

```
SELECT S.Id
FROM STUDENT S
WHERE
    NOT EXISTS (
        -- CS professors who did not teach S.Id
        (SELECT P.Id -- All CS professors
         FROM PROFESSOR P
         WHERE P.Dept = 'CS')
EXCEPT
    (SELECT T.ProfId -- Professors who have taught S.Id
     FROM TEACHING T, TRANSCRIPT R
     WHERE T.CrsCode = R.CrsCode
       AND T.Semester = R.Semester
       AND S.Id = R.StudId) )
```

---

5.28

The variable  $S$  is global, and the subquery is evaluated for each value of  $S$ . The variable  $R$  ranges over all tuples in  $\text{TRANSCRIPT}$ . It is local to the second subquery where it is related to  $S$  through a condition in the  $\text{WHERE}$  clause. Therefore, the values of  $R.\text{Semester}$  and  $R.\text{CrsCode}$  correspond to all recorded enrollments of student  $S.\text{Id}$ . Similarly  $T.\text{ProfId}$  gets successively bound to all professors who ever taught  $S.\text{Id}$ .

To understand why this SQL expression represents the query at hand, recall that the combination NOT EXISTS/EXCEPT is nothing but the aforesaid CONTAINS predicate over sets. ■

**Set comparison operators.** Suppose that our STUDENT relation has one additional numeric attribute, GPA. We can ask the question *Is there a student in the university whose GPA is higher than that of all junior students?*

It turns out that nested queries are helpful here, too.

---

```
SELECT S.Name, S.Id
FROM STUDENT S
WHERE S.GPA > ALL (SELECT S.GPA
                      FROM STUDENT S
                      WHERE S.Status = 'junior')
```

---

5.29

Here `> ALL` is a comparison operator, which returns true whenever its left argument is greater than *every* element of the set to the right. If we replace `> ALL` with `>= ANY`, we obtain a query about students whose GPA is greater than or equal to the GPA of *some* junior student.

One other point is worth noting about this query. The variable `S` is declared in both the outer and the inner queries. So which one is referred to in the `WHERE` clause of the inner query? The answer, as with `begin/end` blocks, is that the inner declaration is valid within the inner query. (However, excessive reuse of existing variable names in inner queries can be confusing.)

**Nested subqueries in the FROM clause.** As if query (5.28) were not complex enough, SQL has more up its sleeve: you can have nested subqueries in the `FROM` clause! This works as follows: You write a nested subquery (it must not be correlated and hence cannot use global variables). This subquery can be used in the `FROM` clause as if it were a relation name. You can use the keyword `AS` to attach a tuple variable to it. (Actually, `AS` is optional here but is highly recommended for readability.)

To illustrate, consider a query similar to (5.28) but without `NOT EXISTS` (i.e., the required answer would consist of all students who were *not* taught by at least one CS professor). We can formulate an equivalent query by moving the first nested subquery of (5.28) to the `FROM` clause as shown below. (Note that we cannot move the second subquery, because it is correlated.)

---

```
SELECT S.Id
FROM STUDENT S,
     (SELECT P.Id -- All CS professors
      FROM PROFESSOR P
      WHERE P.Dept = 'CS') AS C
```

5.30

---

```

WHERE C.ProfId NOT IN
  (SELECT T.ProfId -- All S.Id's professors
   FROM TEACHING T, TRANSCRIPT R
   WHERE T.CrsCode = R.CrsCode
     AND T.Semester = R.Semester)
   AND S.Id = R.StudId )

```

---

The use of nested subqueries in the **FROM** clause should be avoided if at all possible as it tends to produce queries that are hard to understand and verify. A much better alternative is to use the view mechanism, which will be discussed in Section 5.2.8.

Apart from nested queries, SQL also permits explicit table joins in the **FROM** clause. However, we do not discuss this feature.

#### 5.2.4 Quantified Predicates

Beginning with SQL:1999, the language supports a limited form of explicit universal and existential quantification. Although this new feature does not increase the expressive power of the language, it makes certain queries easier to understand. The basic idea is to include quantified predicates with the following general syntax:

---

```

FOR ALL table-name-or-query (condition)
FOR SOME table-name-or-query (condition)

```

---

A quantified predicate can be used in the **WHERE** clause like any other predicate. It is true if and only if every row (FOR ALL) or some rows (FOR SOME) in the set of tuples represented by *table-name-or-query* satisfy *condition*. The condition in a quantified predicate can be as complex as any **WHERE**-clause condition, and it can refer to the attributes of the tuples in *table-name-or-query*. To make this concrete, consider the following example of a quantified predicate:

---

```

FOR ALL PROFESSOR
  (Id IN (SELECT T.ProfId FROM TEACHING))

```

---

When it appears in a **WHERE** clause (e.g., as a conjunct along with other predicates), it verifies that every professor (identified through the **Id** attribute) is teaching something. For a more complex example, we show how quantified predicates can make expression of the division operator easier to understand.

**Example 5.2.6 (Expressing Division Using Universal Quantification).** Consider the query *List the students who have taken all computer science courses*, which was earlier represented in SQL in quite a convoluted way (see query (5.27)). The preceding discussion on page 160 made it clear that part of the reason for this complexity is the absence of the **CONTAINS** predicate in SQL. Fortunately, **CONTAINS** can be expressed

as a quantified predicate much more naturally than with the NOT EXISTS/EXCEPT combination used in (5.27):

---

```

SELECT  S.Id
FROM    STUDENT S
WHERE
        FOR ALL (SELECT C.CrsCode
                  FROM COURSE C
                  WHERE C.CrsCode LIKE 'CS%' )
                  (CrsCode IN
                    (SELECT R.CrsCode
                      FROM TRANSCRIPT R
                      WHERE R.StudId = S.Id ) )

```

---

While this is not as simple as what would have been possible with CONTAINS, it comes close. ■

The explicit existential quantifier, FOR SOME, is less useful in SQL than the universal quantifier, but it is provided for symmetry. For instance, if we wanted to find out if any professor teaches CS305, we could use the following test in the WHERE clause:

---

```

FOR SOME PROFESSOR
(Id IN (SELECT T.ProfId FROM TEACHING
          WHERE T.CrsCode = CS305 ))

```

---

### 5.2.5 Aggregation over Data

In many instances, it is necessary to compute average salary, maximum GPA, number of employees per department, total cost of a purchase, and so forth. These tasks are performed with the help of **aggregate functions**, which operate on sets of tuples. SQL uses five aggregate functions that are described in Figure 5.8.

Aggregate functions cannot be expressed in pure relational algebra. However, the algebra can be extended to allow their use (these extensions are beyond the scope of this text).

To illustrate the use of aggregate functions, we assume that both STUDENT and PROFESSOR relations have the attribute Age and that the STUDENT relation also has the attribute GPA. We start with a few simple examples.

---

```

-- Average age of the student body
SELECT  AVG(S.Age)
FROM    STUDENT S

```

---

```

-- Minimum age among professors in the Management Department
SELECT  MIN(P.Age)

```

---

```
FROM   PROFESSOR P
WHERE  P.DeptId = 'MGT'
```

---

The above queries find only the average and the minimum ages, not the actual people who have them. If we need to find the youngest professor(s) within the Management Department, we can use a nested subquery.

---

```
-- Youngest professor(s) in the Management Department
SELECT  P.Name, P.Age
FROM    PROFESSOR P
WHERE   P.DeptId = 'MGT' AND
        P.Age = (SELECT  MIN(P1.Age)
                  FROM    PROFESSOR P1
                  WHERE   P1.DeptId = 'MGT' )
```

---

The query (5.29) that returns the names and Ids of juniors with the highest GPA (previously written without aggregates) can be equivalently written with the use of MAX:

---

```
SELECT  S.Name, S.StudId
FROM    STUDENT S
WHERE   S.GPA >= (SELECT  MAX(S1.GPA)
                  FROM    STUDENT S1
                  WHERE   S1.Status = 'Junior')
```

---

**5.31**

COUNT([DISTINCT] Attr)	Count the number of values in column Attr of the query result. The optional keyword DISTINCT indicates that each value should be counted only once, even if it occurs multiple times in different answer tuples.
SUM([DISTINCT] Attr)	Sum up the values in column Attr. DISTINCT means that each value should contribute to the sum only once, regardless of how often it occurs in column Attr.
AVG([DISTINCT] Attr)	Compute the average of the values in column Attr. Again, DISTINCT means that each value should be used only once.
MAX(Attr)	Compute the maximum value in column Attr. DISTINCT is not used with this function, as it would have no effect.
MIN(Attr)	Compute the minimum value in column Attr. Again, DISTINCT is not used with this function.

FIGURE 5.8 SQL aggregate functions.

The use of DISTINCT in aggregate functions can sometimes make subtle differences in the query semantics. For instance,

---

```
SELECT COUNT(P.Name)
FROM PROFESSOR P
WHERE P.DeptId = 'MGT'
```

---

5.32

returns the number of professors in the Management Department. On the other hand,

---

```
SELECT COUNT(DISTINCT P.Name)
FROM PROFESSOR P
WHERE P.DeptId = 'MGT'
```

---

returns the number of distinct *names* of professors in that department, which can be different from the number of professors. Similarly,

---

```
SELECT AVG(P.Age)
FROM PROFESSOR P
WHERE P.DeptId = 'MGT'
```

---

5.33

returns the average age of professors in the Management Department. However, if we write AVG ( DISTINCT P.Age ) in the above SELECT clause, the query result is the average value among *distinct* ages—a statistically meaningless number.

Now that you have seen the good things you can do with aggregates, you should also keep in mind the things you cannot do. It makes no sense to mix an aggregate and an attribute in the SELECT list, as in

---

```
SELECT COUNT(*), S.Id
FROM STUDENT S
WHERE S.Name = 'JohnDoe'
```

---

5.34

This is because the aggregate produces a single value that pertains to the entire set of rows corresponding to John Doe, while the attribute S.Id produces a distinct value for each row (in our case there can be several people named John Doe). In some cases, however, such associations can be made meaningful with the help of the GROUP BY construct, to be defined shortly.

While associating aggregates with attributes in the SELECT clause is not normally very useful, having multiple aggregates does make sense. For instance,

---

```
SELECT COUNT(*), AVG(P.Age)
FROM STUDENT S
WHERE S.Name = 'JohnDoe'
```

---

**5.35**

counts the number of John Does in the student relation and also computes their average age. Finally, one might be tempted to rewrite statement (5.31) as

---

```
SELECT S.Name, S.StudId
FROM STUDENT S
WHERE S.GPA >= (MAX(SELECT S1.GPA
                      FROM STUDENT S1
                     WHERE S1.Status = 'junior'))
```

---

but dare not—it is an invalid construct. Aggregates cannot be applied to the result of a query.

**Aggregation and grouping.** By now we know how to count professors in the Management Department. But what if we need this information for *each* department in the university? Of course, we could construct queries similar to (5.32) for each separate department. Each query would be the same, except that MGT in the WHERE condition would be replaced with other department codes. Clearly, this is not a practical solution, as even in a medium-sized enterprise the number of departments can reach several dozen. Furthermore, each time a new department is created, we have to construct a new query, and when departments change their name we have to do tedious maintenance.

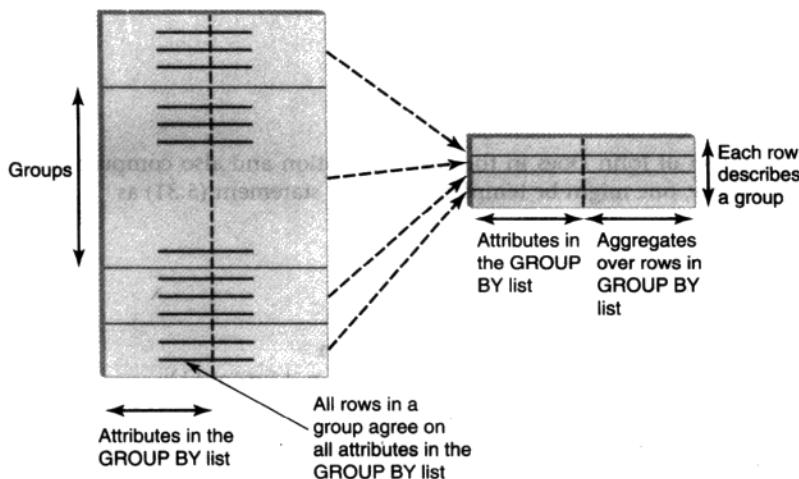
A better solution is provided in the form of the GROUP BY clause, which can be included as a component of a SELECT statement. This clause lets the programmer partition a set of rows into groups whose membership is characterized by the fact that all of the rows in a single group agree on the values in some specified subset of columns. The aggregate function is then applied to each group and yields a single row for each such group, as shown in Figure 5.9. For example, if we group the instance of the relation TRANSCRIPT shown in Figure 3.5, page 39, based on the column StudId, five groups result. In any particular group, all rows have the same value in the StudId column but might differ in other columns.

For instance, the following query

---

```
SELECT T.StudId, COUNT(*) AS NumCrs,
       AVG(T.Grade) AS CrsAvg
  FROM TRANSCRIPT T
 GROUP BY T.StudId
```

---



**FIGURE 5.9** Effect of the GROUP BY clause.

produces the table

TRANSCRIPT	StudId	NumCrs	CrsAvg
	666666666	3	3.33
	987654321	2	2.5
	123454321	3	3.33
	023456789	2	3.5
	111111111	3	3.33

Here is how to determine the number of professors in each department and (why not?) their average age:

---

```

SELECT P.DeptId, COUNT(P.Name) AS DeptSize,
       AVG(P.Age) AS AvgAge
  FROM PROFESSOR P
 GROUP BY P.DeptId
  
```

---

The important point to note in these two queries is that each column in the **SELECT** clause either must be named in the **GROUP BY** clause or must be the result of an aggregate function. Soon (on page 173) we will see why this is needed.

**The HAVING clause.** The **HAVING** clause is used in conjunction with **GROUP BY**. It lets the programmer specify a condition that restricts which groups (specified in the **GROUP BY** clause) are to be considered for the final query result. Groups that

do not satisfy the condition are removed before the aggregates are applied. Suppose that we wish to know the number of professors and the average ages of professors by department, as in the previous query, but this time only if the department has more than 10 professors. This is accomplished as follows:

---

```
SELECT      P.DeptId, COUNT(*) AS DeptSize,
            AVG( P.Age) AS AvgAge
  FROM      PROFESSOR P
 GROUP BY   P.DeptId
 HAVING     COUNT(*) > 10
```

---

The **HAVING** condition (unlike the **WHERE** condition) is applied to groups, *not* to individual tuples. So, for each group created by the **GROUP BY** clause, **COUNT(\*)** counts the number of tuples. Only the groups where this count exceeds 10 are passed on for further processing. In the end, the aggregate functions are applied to each group to yield a single tuple per group.

Observe that the above queries use **AS** to give names to columns produced by aggregate functions. Furthermore, **\*** is used with the **COUNT** function that appears in the **HAVING** clause. The **\*** is often convenient in conjunction with aggregate functions, and it can be used in both the **SELECT** list and the **HAVING** clause. However, it should be noted that, in the above example, there are several alternatives. We can use **P.Name** and even **P.DeptId** instead of **\*** because SQL will not eliminate duplicates without an explicit request (**DISTINCT**).

If we want to consider candidates for the dean's list on the basis of their grades for the 2003–2004 academic year, we might use

---

```
SELECT      T.StudId, AVG(T.Grade) AS CrsAvg
  FROM      TRANSCRIPT T
 WHERE     T.Semester IN ('F2003', 'S2004')
 GROUP BY   T.StudId
 HAVING     AVG(T.Grade) > 3.5
```

---

In general, the **HAVING** clause is just a syntactic convenience—the same result can always be achieved with the help of nested queries in the **FROM** clause. For example, the previous query can be replaced with the following query, which does not use **HAVING**:

---

```
SELECT      Stats.StudId, Stats.CrsAvg
  FROM      (SELECT  T.StudId,
                  AVG(T.Grade) AS CrsAvg
                FROM    TRANSCRIPT T
                WHERE   T.Semester IN ('F2002', 'S2004')
                GROUP BY T.StudId) AS Stats
 WHERE     Stats.CrsAvg > 3.5
```

---

However, the use of nested queries in the **FROM** clause should be avoided, as such queries are harder to understand and optimize.

**The ORDER BY clause.** Finally, the order of rows in the query result is generally not specified. If a particular ordering is desired, the **ORDER BY** clause can be used. For example, if we include the clause

---

**ORDER BY CrsAvg**

---

in the **SELECT** statement that produces the dean's list, rows of the query result will be in ascending order of the student's average grade. In general, the clause takes as an argument a list of column names of the query result. Rows are output in sorted order on the basis of the first column named in the list. In the case in which multiple rows have the same value in that column, the second column named in the list is used to decide the ordering, and so forth. For example, if we want to output the candidates for the dean's list ordered primarily by average grade and secondarily by student Id, we might use the following **SELECT** statement:

---

```
SELECT      T.StudId, AVG(T.Grade) AS CrsAvg
FROM        TRANSCRIPT T
WHERE       T.Semester IN ('F1997', 'S1998')
GROUP BY    T.StudId
HAVING     AVG(T.Grade) > 3.5
ORDER BY    CrsAvg, StudId
```

---

The attributes named in the **ORDER BY** clause must be the names of columns in the query result. Thus, we refer to the second element as **StudId** (not **T.StudId**) in this example since, by default, that is the column name in the query result. Similarly, we cannot order rows primarily by average grade without introducing the column alias **CrsAvg** in the **SELECT** clause because without the alias the column has no name.

Ascending order is used by default, but descending order can also be specified. If in the above example we had replaced the **ORDER BY** clause with

---

**ORDER BY DESC CrsAvg, ASC StudId**

---

the rows of the query result would have been presented in descending order of average grade and, for students with the same average grade, in ascending order of their Ids.

### 5.2.6 Join Expressions in the **FROM** Clause

In SQL terms, the objects that play the role of tables in the **FROM** clause are called **table expressions**. In most of the examples that we have seen so far, table

expressions were simply table names. However, we have also seen that an entire SELECT query can be a table expression. But SQL does not stop there—it allows algebraic expressions to be table expressions as well. These expressions take the form of a join: natural, theta-join, and the three outer joins discussed on page 142. The syntax is as follows (we present only some of the options):

---

*table1 [NATURAL] [INNER|FULL|LEFT|RIGHT] JOIN table2 [ON condition]*

---

The term **INNER** refers to the normal join, and **FULL**, **LEFT**, and **RIGHT** refer to the three types of the outer join. The options **NATURAL** and **ON** are alternatives. If **NATURAL** is specified, then the join is performed on the common attributes of both tables; otherwise, the **ON condition** option must be given, where *condition* can be any legal condition in the WHERE clause. More commonly, however, these conditions take the form

---

*table1.col1\_1 op1 table2.col2\_1 AND table1.col1\_2 op2 table2.col2\_2 AND...*

---

where *op1*, *op2*, etc., are the usual comparisons =, >, and so on.

We illustrate this feature with an example of a left outer join, where the query computes the average grade for *every* student in the database. We assume that the grade attribute in the TRANSCRIPT relation is numeric.

---

```
SELECT      S.Name, AVG(S.Grade)
FROM        (STUDENT LEFT JOIN TRANSCRIPT
           ON STUDENT.Id = TRANSCRIPT.StudId) S
GROUP BY    S.Id
```

5.36


---

Here students who never took a course will have their tuples padded with NULLs over the Grade attribute. The AVG function ignores NULL values, so the average grade for such students will be 0. Note that a regular join query such as

---

```
SELECT      S.Name, AVG(T.Grade)
FROM        STUDENT S, TRANSCRIPT T
WHERE       S.Id = T.StudId
GROUP BY    S.Id
```

---

would be incorrect. In our database, the STUDENT relation has tuples that do not match any record in the TRANSCRIPT relation and, therefore, the above regular join query will miss students who never took any course.

### 5.2.7 A Simple Query Evaluation Algorithm

The overall query evaluation process in the presence of aggregates and grouping is illustrated in Figure 5.10.

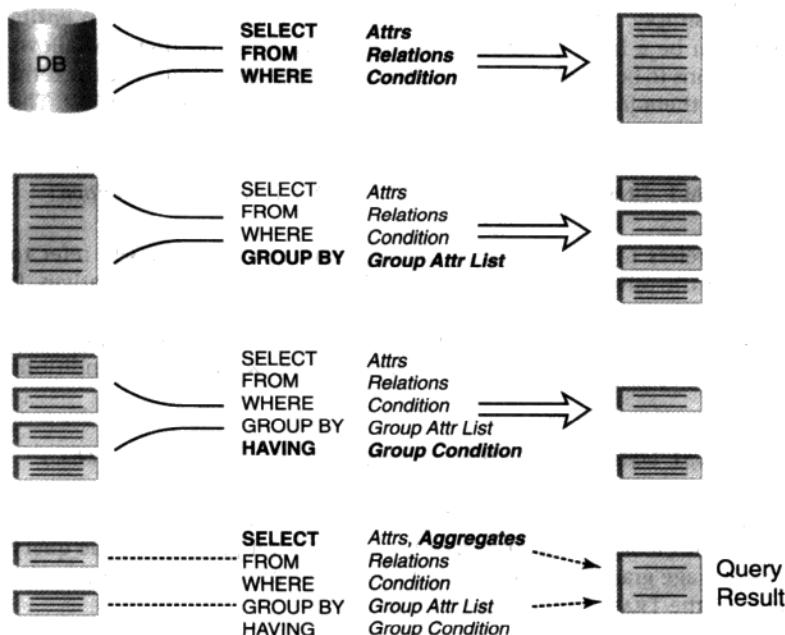


FIGURE 5.10 Query evaluation with aggregate functions.

**Step 1.** The **FROM** clause is evaluated. It produces a table that is the Cartesian product of the tables listed as its arguments.

**Step 2.** The **WHERE** clause is evaluated. It takes the table produced in step 1 and processes each row individually. Attribute values from the row are substituted for the attribute names in the condition, and the condition is evaluated. The table produced by the **WHERE** clause contains exactly those rows for which the condition evaluates to true. Steps 1 and 2 are shown in the top segment of Figure 5.10.

**Step 3.** The **GROUP BY** clause is evaluated. It takes the table produced in step 2 and splits it into groups of tuples, where each group consists precisely of those tuples that agree on all attributes of the group attribute list. This step is shown in the second segment of Figure 5.10.

**Step 4.** The **HAVING** clause is evaluated. It takes the groups produced in step 3 and eliminates those that fail the group condition. This step is shown in the third segment of Figure 5.10.

**Step 5.** The **SELECT** clause is evaluated. It takes the groups produced in step 4, evaluates the aggregate functions in the target list for each group, retains those

columns that are listed as arguments of the **SELECT** clause, and produces a single row for each group. This step is illustrated by the bottom segment of Figure 5.10.

**Step 6.** The **ORDER BY** clause is evaluated. It orders the rows produced in step 5 using the specified column list. The resulting table is output by the **SELECT** statement. This step is not shown in Figure 5.10.

**Restrictions on GROUP BY and HAVING.** Like your apartment lease, grouping comes with several strings attached. The difference is, some of these restrictions actually make sense! Consider a general form of SQL queries with aggregates:

---

<b>SELECT</b>	<i>attributeList, aggregates</i>
<b>FROM</b>	<i>relationList</i>
<b>WHERE</b>	<i>whereCondition</i>
<b>GROUP BY</b>	<i>groupList</i>
<b>HAVING</b>	<i>groupCondition</i>

---

**5.37**

The purpose of the **GROUP BY** clause is to partition the result of the query into groups, with all tuples in the same group agreeing on each attribute in *groupList*. The aggregate functions in the **SELECT** clause are then applied to each group to produce a *single* tuple. Since a single value is output for each attribute in *attributeList* in the **SELECT** clause (and that value is not an aggregate), all tuples in any given group must agree on each attribute in the list. This property is achieved in SQL by requiring that *attributeList* be a subset of *groupList*, which is a sufficient condition to ensure that each group yields a single tuple. (The condition is not necessary but is good enough for most purposes.)

The second restriction concerns the **HAVING** condition. Intuitively, we need to ensure that *groupCondition* is either true or false for each group of tuples specified in **GROUP BY**. Generally, this condition consists of a number of comparisons of the form *expr<sub>1</sub> op expr<sub>2</sub>*, which are tied together by the logical connectives **AND**, **OR**, and **NOT**. For an atomic comparison *expr<sub>1</sub> op expr<sub>2</sub>* to make sense, both *expr<sub>1</sub>* and *expr<sub>2</sub>* must evaluate to a single value for each group. In practice, this means that for every attribute mentioned in *groupCondition* either of the following must hold:

1. It is in *groupList* (and thus it has a single value per group).
2. It appears in *groupCondition* as an argument to an aggregate function (thus, the expression sees a single value when the aggregation is computed).

Most DBMSs enforce the above syntactic restrictions.

We should also note that the order of the clauses in (5.37) is important—for instance, the **HAVING** clause cannot precede the **GROUP BY** clause. However, the standard allows SQL statements that have the **HAVING** clause without the **GROUP BY** clause. In this case, the result of the **SELECT-FROM-WHERE** part of the query is treated as a single group and *groupCondition* in **HAVING** is applied to that group.

### 5.2.8 More on Views in SQL

The relations we have discussed up to this point are more technically referred to as **base relations**. They are the "normal" database relations. The contents of a base relation are physically stored on disk and are independent of the contents of other relations in the database.

As we already discussed in Chapter 3, a view is a relation whose contents are usually *not physically stored* in the database. Instead, it is defined as the query result of a **SELECT** statement. Each time the view is used, its contents are computed using the associated query. Hence, the *definition* of the view, that is, the query that determines the view's contents, is stored (in the system catalog) rather than the contents. Because each view is the result of executing a query, its contents depend on the contents of the base relations at the time the view is referenced.

The role of views in query languages is similar to that of *subroutines* in conventional programming languages. A view usually represents some meaningful query, which is used within several other, frequently asked queries, or it might have an independent interest. In either case, it makes sense to abstract the query and "pretend" that the database contains a relation whose contents precisely coincide with the query result.

Another important use of the view mechanism is to control user access to the data. Access control was discussed in Section 3.3.12; in the context of the views it is discussed later in this section.

**Using views in queries.** Once a view is defined, it can be used in SQL queries in the same way as any other table. Whenever it is used in a query, its definition is automatically substituted in the **FROM** clause, as illustrated in Figure 5.11.

Suppose that the university needs to find the department(s) where the average age of professors is the lowest. The application designer might determine that, in addition to the above query, a number of other queries compute the average age. In query processing, as in programming languages, this is a good enough reason to build a view for computing the average age.

**FIGURE 5.11** Process of query modification by views.

```
SELECT ... ...
FROM   VIEW1 V
WHERE  ... AND V.Attr = 'abc' AND ...
```

*Query that uses  
a view, VIEW1*

becomes

```
SELECT ...
FROM   (definition of VIEW1) AS V
WHERE  ... AND V.Attr = 'abc' AND...
```

*Query modified by  
the view definition*

---

```
CREATE VIEW AVGDEPTAGE(Dept,AvgAge) AS
    SELECT      P.DeptId, AVG(P.Age)
    FROM        PROFESSOR P
    GROUP BY    P.DeptId
```

---

Like a subroutine, this view allows us to solve part of a larger problem separately. For example, we can now find the departments with the minimum average age as follows:

---

<pre>SELECT      A.Dept FROM        AVGDEPTAGE A WHERE       A.AvgAge = (SELECT MIN(A.AvgAge)                       FROM AvgDEPTAGE A )</pre>	<b>5.38</b>
---	-------------

---

Views can make a complex SQL query easier to understand (and debug!). Consider query (5.28), page 161, which finds all students who have taken a course from each professor in the Department of Computer Science. The query uses two nested subqueries, one of which is correlated with the outer query. Because the issues of nesting and correlation are subtly intertwined, it might be hard to construct the right query the first time.

We can simplify the task with the help of views. The view

---

```
CREATE VIEW ALLCSPROFIDs(ProfId) AS
    SELECT      P.Id
    FROM        PROFESSOR P
    WHERE       P.DeptId = 'CS'
```

---

constructs the set of Ids of all computer science professors. Next, we define a view to represent the second correlated subquery of (5.28). This subquery has a target list with only one attribute, ProfId, but it also uses a global variable, S, which parameterizes the sets of answers returned by the query. Each query execution returns the set of all professors who have taught a particular student. Since SQL does not allow the creation of views parameterized by a global variable, we improvise by including the appropriate attributes in the target list of the view. More precisely, we include those attributes that are actually used in the subquery in conjunction with the global variable. In our case, the global variable is S and the additional attribute is StudId. The result turns out to be the already familiar view, PROFSTUD, defined in (3.5), page 59.

Unfortunately, we cannot subtract PROFSTUD from ALLCSPROFIDs yet, as they are not UNION-compatible. Furthermore, SQL allows the EXCEPT operator to be applied only to the results of subqueries—we cannot simply subtract one table from another. Therefore, we still must use nested subqueries. However, the subqueries are now much more manageable than in (5.28) since the views enable us to decompose a complex problem into smaller tasks.

---

```

SELECT  S.Id
FROM    STUDENT S
WHERE   NOT EXISTS (
            (SELECT P.Id FROM ALLCSPROFIDs P)
            EXCEPT
            (SELECT P.Id FROM PROFSTUD P
             WHERE P.StudId = S.Id))

```

---

Although the CREATE VIEW statement is quite different from the CREATE TABLE statement used for base relations, the deletion of views and tables from the system catalog uses similar statements: DROP VIEW for views and DROP TABLE for base tables.

What if we want to drop a view or a base table but the database has other views that were defined through this view or this table? The problem here is that dropping such a view or table means that all of the views defined through them will become "abandoned" and there will be no way to use them. In such a case, SQL leaves the decision to the designer of the DROP statement. The general format of the statement is

---

```
DROP {TABLE | VIEW } table-or-view {RESTRICT | CASCADE}
```

---

If the RESTRICT option is used, the drop operation fails if some view is dependent on the table or view being dropped. With the CASCADE option, all dependent views are also dropped.

**Access control and customization through views.** Database views are used not only as a subroutine mechanism but also as a flexible device for controlling access to the data. Thus, we might allow certain users to access a view but not some of the tables that underlie it. For instance, students might not be allowed to query the PROFESSOR relation because of the Social Security information stored there. However, there is nothing wrong with giving students access to the AVGDEPTAGE view defined earlier. Thus, while the access to PROFESSOR might be restricted to administrators, we might let students query the view AVGDEPTAGE:

---

```
GRANT SELECT ON AVGDEPTAGE TO ALL
```

---

Note that by using views we can repair a deficiency in the GRANT statement. In granting UPDATE (or INSERT) permission, we are allowed to (optionally) specify a list of columns that can be updated (or into which values can be inserted), but in granting SELECT permission SQL provides no way to specify such a list. We can get the same effect, however, by simply creating a view of accessible columns and granting access to that view instead of to the base table.

The creator (and thus the owner) of a view need not also be the owner of the underlying base relations. All that is required is that the view creator have SELECT

privileges on all of the underlying relations. For instance, if the PROFESSOR relation is owned by *Administrator*, who in turn grants the SELECT privilege on PROFESSOR to *Personnel*, *Personnel* can create the view AVGDEPTAGE and later issue the above GRANT statement.

What happens if *Administrator* decides to revoke the SELECT privilege from *Personnel*? Notice that if *Administrator* revokes the privilege, the view becomes “abandoned” and nobody can query it. The actual result depends on how the REVOKE statement is issued. If the administrator uses the RESTRICT option in the REVOKE statement, revocation fails. If the CASCADE option is used, the revocation proceeds and the view itself is dropped from the system catalog.

Yet another use of views is customization, which goes hand in hand with access control. A real production database might contain hundreds of relations, each with dozens of attributes. However, most users (both “naive” users and application developers) need to deal with only a small portion of the database schema, the part that is relevant to the particular task performed by the user or the application. There is no benefit in subjecting all users to the tortuous process of learning large parts of the database schema. A better strategy is to create views customized to the various user categories so that, for instance, AVGDEPTAGE can be one of the views customized for the statisticians. The advantages of this approach are threefold:

1. *Ease of use and learning.* This speeds up application development and might prevent bugs that occur as a result of misunderstanding parts of the database schema.
2. *Security.* Various users and applications can be granted access to specific views, thereby reducing the possible damage from human errors and malicious behavior.
3. *Logical data independence* (as discussed in Chapter 3). This benefit can result in huge savings in maintenance costs if later there is a need to change the database schema. Provided that schema reorganization does not lead to loss of information, none of the applications written against the views have to be changed—the only required change is in the view definitions themselves.

### 5.2.9 Materialized Views

If a view becomes popular with many queries, its contents might be stored in a cache. Cached views are often referred to as **materialized views**. View caching can dramatically improve the response time of queries defined in terms of such views, but update transactions must pay the price. If a view depends on a base relation and a transaction updates the base relation, the view cache might need to be updated as well.

Consider the view PROFSTUD in (3.5), page 59, and suppose that a transaction adds tuple (023456789, CS315, S1997, B) to TRANSCRIPT. This tuple joins with tuple (101202303, CS315, S1997) in TEACHING to produce a view tuple, (101202303, 023456789). But this latter tuple is already in the view (see Figure 3.9, page 60), so this update does not change the view. Had we added (023456789, MGT123, F1997, A)

to TRANSCRIPT, the view would have acquired new tuples, (783432188, 023456789) and (009406321, 023456789).

Consider now what might happen when tuples are deleted from the base relations underlying a materialized view. If a transaction deletes tuple (123454321, CS305, S1996, A) from TRANSCRIPT, one might think that tuple (101202303, 123454321) should also be deleted from the view cache. This is not the case, however, because (101202303, 123454321) can still be derived through a join between (123454321, CS315, S1997, A) and (101202303, CS315, S1997). Observe that Figure 3.9 indicates two reasons for tuple (101202303, 123454321) to be in the view, and the deletion of (123454321, CS305, S1996, A) removes only one of the reasons! However, if the transaction deleted (123454321, MAT123, S1996, C) from TRANSCRIPT, the tuple (900120450, 123454321) should be removed from the view cache as well.

View cache maintenance is an algorithmically nontrivial task. Of course, a view can be simply recomputed anew each time a base table is changed, but this can be unacceptably expensive, especially if base tables change frequently. A number of algorithms have been proposed, which make it possible to recompute views *incrementally*, that is, by recomputing only those parts of the view that are directly related to the changes in the base tables (refer back to the earlier discussion of the view PROFSTUD). These advanced methods include [Gupta et al. 1993; Mohania et al. 1997; Gupta et al. 1995; Blakeley and Martin 1990; Chaudhuri et al. 1995; Staudt and Jarke 1996; Gupta and Mumick 1995], and more research is still being conducted on the topic. The main difference between the various approaches is how to determine which parts of the view might need to be recomputed (which affects the amount of work involved in a view update) and what type of views a particular method can handle.

Materialized views are especially important in *data warehousing*. A **data warehouse** is an (infrequently updated) database that typically consist of complex materialized views of the data stored in a *separate* production database. Data warehouses are commonly used for online analytical processing (OLAP), which was briefly discussed in Chapter 1. In contrast to most production databases, data warehouses are optimized for querying, not transaction processing, and they are the primary beneficiaries of the advanced query capabilities of SQL discussed in this chapter. (In many transaction processing applications, rapid response time and high throughput requirements preclude the use of the complex queries.)

As materialized views are becoming more and more important, commercial DBMSs are beginning to provide support for such views. Unfortunately, the SQL standard has not yet caught up with the idea, so we will illustrate the SQL extensions for supporting these views using Oracle as an example. Other implementations differ in detail but not substance.

Maintenance of materialized views involves the following main considerations:

- **Build method.** This concerns the time when the view is actually materialized, that is, when its contents are first computed based on the base tables. The build method can be IMMEDIATE or DEFERRED. When the build method is immediate,

the view is populated right after the view is created. With the deferred build method, the view is populated using a utility that Oracle specifically provides for this purpose. The user must execute this utility manually.

- **Refresh mode.** This concerns the timing when the view is recomputed. Ideally, each materialized view should be refreshed each time a change is made to the underlying base tables. This refresh mode is known as **ON COMMIT**. However, if the changes happen frequently while the view access is infrequent, refreshing the view in this mode might consume computational resources for no apparent benefit. The **ON DEMAND** option allows the user to control when the refresh happens. In this case, the user must explicitly call a special routine to bring the view up to date. Yet another option is to refresh the view periodically (but automatically, without the user intervention). In this case, the user can tell when to perform the first refresh and how often to do it. This mode is especially useful in data warehousing applications where temporal discrepancy between views and the underlying base tables is not critical.
- **Refresh method.** This option tells the system how to refresh the view. One obvious way is to recompute the view from scratch, a **COMPLETE** refresh. Another possibility is a **FAST** refresh. This means that the system will try to use one of the advanced methods mentioned earlier, which try to update the view incrementally without recomputing it from scratch. Still, some complex views cannot be refreshed incrementally. If the user has difficulties determining whether a particular view can be refreshed using the **FAST** method, she can always specify the **FORCE** method. In this case, the system will try to determine if **FAST** is possible, and if not, it will use **COMPLETE**.
- **Query rewriting.** With a regular view, explicit references to the view cause the query to be rewritten to include the definition of the view, as explained in Figure 5.11. When a view is materialized, explicit references use the view cache instead. Using a cache instead of the view definition can have dramatic effects on performance. However, some queries might have been written without the mention of a materialized view even though the query can be equivalently rewritten into one that uses the view. This situation might occur for a number of reasons: the user might not have been aware of the view, the query could have been written before the view was added to the database, or the user might have failed to notice that the view can be used in a particular query. To illustrate the issue, consider the following query, which finds all students (just their Ids) who took a course from John Smyth.

---

```
SELECT R.StudId
  FROM TRANSCRIPT R, TEACHING T, PROFESSOR P
 WHERE T.ProfId = P.Id AND P.Name = 'John Smyth' AND
       R.CrsCode = T.CrsCode AND R.Semester = T.Semester
```

---

Notice that part of the **WHERE** clause, the condition **R.CrsCode = T.CrsCode** and **R.Semester = T.Semester**, is equivalent to the one used to define the view

PROFSTUD in (3.5), page 59. Therefore, the above query can be rewritten in the following equivalent form:

---

```
SELECT S.Stud
FROM PROFSTUD S, PROFESSOR P
WHERE S.Prof = P.Id AND P.Name = 'John Smyth'
```

---

If the view PROFSTUD is materialized, such a rewriting might result in a significant performance gain. The option **ENABLE QUERY REWRITE** tells the query optimizer that it should try to rewrite queries using materialized views in a way similar to the above example.

Now we are ready to see examples of materialized views defined using Oracle's extensions of SQL.

---

```
CREATE MATERIALIZED VIEW PROFSTUD(Prof , Stud)
BUILD IMMEDIATE
REFRESH FAST ON COMMIT
ENABLE QUERY REWRITE
AS
SELECT T.ProfId, R.StudId
FROM TRANSCRIPT R, TEACHING T
WHERE R.CrsCode = T.CrsCode AND R.Semester = T.Semester
```

---

This statement tells the system that the view should be filled in with data immediately and that it should be refreshed using an advanced algorithm for incremental view update. These refreshes should take place each time an update transaction commits changes to the underlying base tables. Finally, the query optimizer is told to attempt query rewriting using this view whenever possible. For the view AVGDEPTAGE, which was discussed in Section 5.2.8, we could choose different options:

---

```
CREATE MATERIALIZED VIEW AVGDEPTAGE(Dept , AvgAge)
BUILD DEFERRED
REFRESH COMPLETE
START CURRENT_DATE + 1 NEXT CURRENT_DATE + 3
AS
:
```

---

Here the view is said to be populated later on, when the user executes an appropriate system utility. View refreshing will be always done from scratch, and the first refresh should take place tomorrow. Subsequent refreshes should be done every other day regardless of the rate of changes to the base tables. No query rewriting should be attempted with this view.

### 5.2.10 The Null Value Quandary

In Chapter 3, we briefly discussed the concept of a *null value*. For instance, if we take the tuples in the TRANSCRIPT table to stand for courses taken in the past or those being taken in the current semester, some tuples might not have a valid value in the Grade attribute. NULL is a placeholder that SQL uses in such a case.

Null values are an unfortunately unavoidable headache in query processing. Indeed, what is the truth value of the condition `T.Grade = 'A'` if the value of T is a tuple that has NULL in the Grade attribute?

To account for this phenomenon, SQL uses so-called *3-valued logic*, where the truth values are *true*, *false*, and *unknown*, and where  $val_1 \text{ op } val_2$  (op being  $<$ ,  $>$ ,  $\neq$ , etc.) is considered to be *unknown* whenever at least one of the values,  $val_1$  or  $val_2$ , is NULL.

Nulls affect not only comparisons in the WHERE and CHECK clauses but also arithmetic expressions and aggregate functions. An arithmetic expression that encounters a NULL is itself evaluated to NULL. COUNT considers NULL to be a regular value (e.g., statement (5.32) on page 166 counts NULL as well as normal values in producing the number of professors in the Management Department). All other aggregates just throw NULLs away (e.g., statement (5.36) on page 171 ignores NULLs when computing the average). The following caveat holds, however: if such an aggregate function is applied to a column that has only NULLs, the result is a NULL.

If all this multitude of exceptions does not seem to be too bad—hang on: some vendors do not follow the standard and, for example, ignore NULL while counting.

Sadly, we have not reached the end of this confusing story. We also must decide what to do when a WHERE or a CHECK clause has the form  $cond_1 \text{ AND } cond_2$ ,  $cond_1 \text{ OR } cond_2$ , or  $\text{NOT } cond$ , and one of the subconditions evaluates to *unknown* because of a pesky NULL hidden inside the subcondition. This issue is resolved by the truth tables in Figure 5.12, which shows the value of various Boolean functions depending on the values of subconditions.

$cond_1$	$cond_2$	$cond_1 \text{ AND } cond_2$	$cond_1 \text{ OR } cond_2$	$cond$	$\text{NOT } cond$
<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>
<i>true</i>	<i>unknown</i>	<i>unknown</i>	<i>true</i>	<i>unknown</i>	<i>unknown</i>
<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>unknown</i>	<i>unknown</i>
<i>false</i>	<i>unknown</i>	<i>false</i>	<i>unknown</i>	<i>unknown</i>	<i>unknown</i>
<i>unknown</i>	<i>true</i>	<i>unknown</i>	<i>true</i>	<i>unknown</i>	<i>unknown</i>
<i>unknown</i>	<i>false</i>	<i>false</i>	<i>unknown</i>	<i>unknown</i>	<i>unknown</i>
<i>unknown</i>	<i>unknown</i>	<i>unknown</i>	<i>unknown</i>	<i>unknown</i>	<i>unknown</i>

FIGURE 5.12 SQL's truth tables used to deal with the unknown.

If these tables seem bewildering, they should not be. The idea is very simple. Suppose that we need to compute the value of *true AND unknown*. Because *unknown* might turn out to be either *true* or *false*, the value of the entire expression can also be either *true* or *false* (that is, *unknown*). On the other hand, the expressions *false AND unknown* and *true OR unknown* evaluate to *false* and *true*, respectively, regardless of whether *unknown* turns out to be *true* or *false*. The table for NOT can be explained away similarly.

SQL introduces one additional predicate, IS NULL, which is specifically designed to test if some value is NULL. For instance, *T.Grade IS NULL* is true whenever the tuple assigned to *T* has a null value in the *Grade* attribute; it evaluates to false otherwise. Interestingly, this is the only true 2-valued predicate in SQL!

So what happens when the entire condition evaluates to *unknown*? The answer depends on whether this is a WHERE or a CHECK clause. If a WHERE clause evaluates to *unknown*, it is treated as *false* and the corresponding tuple is not added to the query answer. If a CHECK clause evaluates to *unknown*, the integrity constraint is considered to be observed (i.e., the result is treated as *true*). The difference in the way the unknown values are treated in queries and constraints can be given a rational explanation. It is assumed that the user expects the queries to return only the answers that are definitely *true*. On the other hand, a constraint of the form CHECK (*condition*) is viewed as a statement that the condition should *not* evaluate to *false*. Thus, the *unknown* truth value is considered acceptable.

We have presented only the general framework behind null values in SQL. Some details have been left out but can be found in most standard SQL references, such as [Date and Darwen 1997]. For instance, what is the impact of nulls on the LIKE condition, the IN condition, set comparisons (such as > ALL), the EXISTS feature, duplicate elimination (i.e., queries that use DISTINCT), and the like? Think of what might be reasonable in these cases and then compare your conclusions with those of [Date and Darwen 1997].

## 5.3 Modifying Relation Instances in SQL

So far we have been discussing the query sublanguage—by far the hardest part of SQL. However, databases exist not only for querying but also for entering and modifying the appropriate data. This section deals with the part of SQL that is used for data insertion, deletion, and modification.

### 5.3.1 Inserting Data

The INSERT statement has several forms, in the simplest of which the programmer specifies just the tuple to be inserted. The second version of this statement can insert multiple tuples and uses a query to tell which tuples to insert. To insert a single tuple, the programmer simply writes

---

```
INSERT INTO PROFESSOR(DeptId,Id,Name)
VALUES ('MATH','100100100','Bob Parker')
```

---

The order of the attributes listed in the INTO clause need not correspond to the *default* order (i.e., the order in which they were listed in the CREATE TABLE statement). However, if you know the default attribute order, you can omit the attribute list in the INTO clause. Of course, the order of items in the VALUE clause must then correspond to the default attribute order. Despite the potential time-saving, omitting the attribute list in the INTO clause is error prone and is thus viewed as poor programming style (e.g., consider what might happen if the schema is later changed).

The second form of the INSERT statement enables bulk insertion of tuples into relations. The tuples to be inserted are the result of a query in the INSERT statement. Note that, even though a query is used to define the tuples to be inserted into a relation, this mechanism is fundamentally different from defining views via queries.

As an example, let us insert some tuples into a relation called HARDCLASS. A hard class<sup>7</sup> is one that is failed by more than 10% of the students. The attributes in HARDCLASS are course code, semester, and failure rate.

This query is actually quite complicated because expressing the failure rate in SQL requires some thought. We will tackle this query in steps and first define two views.

--Number of failures per class

```
CREATE VIEW CLASSFAILURES(CrsCode, Semester, Failed) AS
    SELECT      T.CrsCode, T.Semester, COUNT(*)
    FROM        TRANSCRIPT T
    WHERE       T.Grade = 'F'
    GROUP BY    T.CrsCode, T.Semester
```

Similarly, we can define the view CLASSENROLLMENT, which counts the number of students enrolled in each course.

--Number of enrolled students per class

```
CREATE VIEW CLASSENROLLMENT(CrsCode, Semester, Enrolled) AS
    SELECT      T.CrsCode, T.Semester, COUNT(*)
    FROM        TRANSCRIPT T
    GROUP BY    T.CrsCode, T.Semester
```

Now, HARDCLASS can be populated with tuples as follows:

```
INSERT INTO HARDCLASS(CrsCode, Semester, FailRate)           5.39
    SELECT      F.CrsCode, F.Semester, F.Failed/E.Enrolled
    FROM        CLASSFAILURES F, CLASSENROLLMENT E
    WHERE       F.CrsCode = E.CrsCode AND F.Semester = E.Semester
                AND (F.Failed/E.Enrolled) > 0.1
```

<sup>7</sup> By "class" we mean a particular course offering in a given semester.

The final query looks simple, but imagine how complex it would have been if not for the views!

There are a few more subtleties we must mention. First, if an **INSERT** statement inserts a tuple that violates some integrity constraint, the entire operation is aborted and no tuple is inserted (assuming that constraint checking has not been **DEFERRED**, as will be described in Section 8.3).

Second, it is possible to omit a value in the list of values in the **VALUE** clause (and the corresponding attribute in the attribute list) and an attribute in the **SELECT** clause if the **CREATE TABLE** statement does not specify **NOT NULL**. If the **CREATE TABLE** statement specifies a default for the missing attribute, that default value is used. Instead of omitting values, a better style is to use the more informative keywords **DEFAULT** or **NULL** (whichever is appropriate) in place of the missing values (e.g., **(100100100, NULL, DEFAULT)**).

### 5.3.2 Deleting Data

Deletion of tuples is analogous to the second form of **INSERT**, except that now the keyword **DELETE** is used. For example, to delete the hard classes taught in the fall and spring of 2003 we use

---

```
DELETE FROM HARDCLASS
WHERE Semester IN ('S2003', 'F2003')
```

---

Note that the **DELETE** statement does not allow the use of tuple variables in the **FROM** clause.

Suppose that in order to improve teaching standards, the university decides to fire all professors with an excessively high failure rate in one of the courses. This turns out to be difficult (and not because of tenure!). The **DELETE** statement has a very limited form of the **FROM** clause. The programmer can specify just one relation, the one whose tuples are to be deleted. (What would it mean to delete rows from the Cartesian product of two relations?) However, in order to find out who to fire we need to look inside the **HARDCLASS** relation, but there is no room for this relation in the **FROM** clause. So what are we to do? Use nested subqueries!

---

```
DELETE FROM PROFESSOR
WHERE IdIN
  (SELECT T.ProfId
   FROM TEACHING T, HARDCLASS H
   WHERE T.CrsCode = H.CrsCode
     AND T.Semester = H.Semester
     AND H.FailRate > 0.5)
```

---

**5.40**

### 5.3.3 Updating Existing Data

Sometimes it is necessary to change the values of some attributes of existing tuples in a relation. For instance, the following statement changes the grade of student 666666666 for course EE101 from B to A:

---

```
UPDATE TRANSCRIPT
SET Grade = 'A'
WHERE StudId = '666666666' AND CrsCode = 'EE101'
```

---

Observe that, like DELETE, the UPDATE statement does not allow tuple variables and it uses only a limited form of the FROM clause (more precisely, UPDATE itself is a kind of FROM clause). As a result, some of the more complex updates require the use of nested subqueries. For instance, if instead of firing them we decide to transfer all poorly performing professors to administration, we use a subquery similar to the one in (5.40).

---

```
UPDATE PROFESSOR
SET DeptId = 'Adm'
WHERE Id IN
      (SELECT T.ProfId
       FROM TEACHING T, HARDCLASS H
       WHERE T.CrsCode = H.CrsCode
             AND T.Semester = H.Semester
             AND H.FailRate > 0.5)
```

5.41


---

And, yes, here is our favorite again: *Raise the salary of all administrators by 10%:*

---

```
UPDATE EMPLOYEE
SET Salary = Salary * 1.1
WHERE Department = 'Adm'
```

---

### 5.3.4 Updates on Views

Because views are often used as a customization device that shields users and programmers from the complexities of the conceptual database schema, it is only natural to let programmers update their views. Unfortunately, this is easier said than done because of the following three problems:

1. Suppose that we have a simple view over TRANSCRIPT—a projection on the attributes CrsCode, StudId, and Semester. If the programmer wants to insert a new tuple in such a view, the value for the Grade attribute will be missing. This problem is not serious. We can pad the missing attributes with null values if the CREATE TABLE statement for the underlying base relation permits this; if it does not, the insertion command can be rejected.

2. Consider a view, CSPROF, over the PROFESSOR relation, which is obtained by a simple selection on DeptId = 'CS'. Suppose that the programmer inserts (121232343, 'Paul Schmidt', 'EE'). If we propagate this insertion to the underlying base relation (i.e., PROFESSOR), we can observe the anomaly that querying the view *after* the insertion does not show any traces of the tuple we just inserted! Indeed, Paul Schmidt is not a CS professor, so he does not appear in the view defined through the selection DeptId = 'CS'!

By default, SQL does not forbid such anomalies, but a careful database designer might include the clause WITH CHECK OPTION to ensure that the newly inserted or updated tuples in a view do, indeed, satisfy the view definition. In our example, we can write

---

```
CREATE VIEW CSPROF(Id,Name,DeptId) AS
    SELECT P.Id, P.Name, P.DeptId
    FROM   PROFESSOR P
    WHERE  P.DeptId = 'CS'
    WITH CHECK OPTION
```

---

3. The following problem is much more involved. It turns out that some view updates might have several possible translations into the updates of the underlying base relations. This is potentially a very serious problem because the possibilities arising from a single view update might have drastically different consequences with respect to the underlying stored data.

To illustrate this problem, consider the view PROFSTUD, discussed earlier.

---

```
CREATE VIEW PROFSTUD(ProfId,StudId) AS
    SELECT T.ProfId, R.StudId
    FROM   TEACHING T, TRANSCRIPT R
    WHERE  T.CrsCode = R.CrsCode
           AND T.Semester = R.Semester
```

---

The contents of this view were depicted in Figure 3.9 on page 60. Suppose that we now decide to delete the tuple (101202303, 123454321) from that view. How should this update be propagated back to the base relations? There are four possibilities:

1. Delete (101202303, CS315, S1997) and (101202303, CS305, S1996) from the relation TEACHING.
2. Delete (123454321, CS315, S1997, A) and (123454321, CS305, S1996, A) from the relation TRANSCRIPT.
3. Delete (101202303, CS315, S1997) from TEACHING and (123454321, CS305, S1996, A) from TRANSCRIPT.
4. Delete (101202303, CS305, S1996) from TEACHING and (123454321, CS315, S1997, A) from TRANSCRIPT.

For each of these possibilities, the join implied by the view does not contain the tuple  $\langle 101202303, 123454321 \rangle$ . The only problem is, Which possibility should be used for the view update?

This example shows that, in the absence of additional information, it might not be possible to translate view updates into the updates of the underlying base relations uniquely. Much work has been done on defining heuristics aimed at disambiguating view updates, but none has emerged as an acceptable solution. SQL takes a simpleminded approach by defining only a very restricted class of views as updatable. The essence of these restrictions on the view definition is summarized here:

1. Exactly one table can be mentioned in the **FROM** clause (and only once). The **FROM** clause cannot have nested subqueries.
2. Aggregates, **GROUP BY**, or **HAVING** clauses and set operations, such as **UNION** and **EXCEPT**, are not allowed.
3. Nested subqueries in the **WHERE** clause of the view cannot refer to the (unique) table used in the **FROM** clause of the view definition. Moreover, a nested subquery cannot refer to this table, either explicitly, in the **FROM** clause, or implicitly, through a tuple variable defined in the outer query.
4. No expressions and no **DISTINCT** keyword in the **SELECT** clause are allowed.

Views satisfying these conditions (and some other rather obscure restrictions) are called **updatable** (in the SQL sense). Here is an example of an updatable view.

---

```
CREATE VIEW CANTEACH(Professor, Course)
  SELECT T.ProfId, T.CrsCode
  FROM   TEACHING  T
```

---

Referring to the database of Figure 3.5, page 39, suppose that we delete  $\langle 09406321, \text{MGT123} \rangle$  from the view **CANTEACH**. There are two tuples in the underlying base relation (**TEACHING**) that give rise to the view tuple in question:  $\langle 09406321, \text{MGT123}, \text{F1994} \rangle$  and  $\langle 09406321, \text{MGT123}, \text{F1997} \rangle$ . The translation of the view update into an update of **TEACHING** must therefore delete both of these tuples.

## BIBLIOGRAPHIC NOTES

Relational algebra was introduced in Codd's seminal papers [Codd 1972, 1970]. SQL was developed by IBM's System R research group [Astrahan et al. 1981]. [Melton and Simon 1992; Date and Darwen 1997] are references to SQL-92, and [Gulutzan and Pelzer 1999] describe the extensions provided in SQL:1999.

The view update problem received considerable attention in the past. The following is a partial list of works that propose various solutions: [Bancilhon and Spyros 1981; Masunaga 1984; Cosmadakis and Papadimitriou 1983; Gottlob et al. 1988; Keller 1985; Langerak 1990; Chen et al. 1995]. The maintenance problem

for materialized views has also been an active research area [Gupta et al. 1993; Mohania et al. 1997; Gupta et al. 1995; Blakeley and Martin 1990; Chaudhuri et al. 1995; Staudt and Jarke 1996; Gupta and Mumick 1995]. More research is being conducted because of the importance of materialized views in data warehousing.

## EXERCISES

- 5.1 Assume that  $R$  and  $S$  are relations containing  $n_R$  and  $n_S$  tuples, respectively. What is the maximum and minimum number of tuples that can possibly be in the result of each of the following expressions (assuming appropriate union compatibilities)?
- $R \cup S$
  - $R \cap S$
  - $R - S$
  - $R \times S$
  - $R \bowtie S$
  - $R / S$
  - $\sigma_{S=4}(R) \times \pi_{S,t}(S)$
- 5.2 Assume that  $R$  and  $S$  are tables representing the relations of the previous exercise. Design SQL queries that will return the results of each of the expressions of that exercise.
- 5.3 Verify that the Cartesian product is an associative operator—that is,
- $$r \times (s \times t) = (r \times s) \times t$$
- for all relations  $r$ ,  $s$ , and  $t$ .
- 5.4 Verify that selections commute—that is, for any relation  $r$  and any pair of selection conditions  $cond_1$  and  $cond_2$ ,  $\sigma_{cond_1}(\sigma_{cond_2}(r)) = \sigma_{cond_2}(\sigma_{cond_1}(r))$ .
- 5.5 Verify that, for any pair of relations  $r$  and  $s$ ,  $\sigma_{cond}(r \times s) = r \times \sigma_{cond}(s)$  if the selection condition  $cond$  involves *only* the attributes mentioned in the schema of relation  $s$ .
- 5.6 Prove that, if  $r$  and  $s$  are union-compatible, then  $r \cap s = r \bowtie s$ .
- 5.7 Using division, write a relational algebra expression that produces all students who have taken all courses offered in every semester (this implies that they might have taken the same course twice).
- 5.8 Using division, write a relational algebra expression that produces all students who have taken all courses that have been offered. (If a course has been offered more than once, they have to have taken it at least once.)
- 5.9 Construct a relational algebra query that produces the same result as the outer join  $r \bowtie_{cond}^{\text{outer}} s$  using only these operators: union, difference, Cartesian product, projection, general join (not outer join). You can also use constant relations, that is, relations with fixed content (e.g., one that has tuples filled with nulls or other predefined constants).
- 5.10 Express each of the following queries in (i) relational algebra and (ii) SQL using the Student Registration System schema of Figure 3.4.
- List all courses that are taught by professors who belong to the EE or MGT departments.

- b. List the names of all students who took courses *both* in spring 1997 and fall 1998.
  - c. List the names of all students who took courses from at least two professors in different departments.
  - d. List all courses that are offered by the MGT Department and that have been taken by all students.
  - \*e. Find every department that has a professor who taught all courses ever offered by that department.
- 5.11** Use the relational algebra to find the list of all “problematic” classes (i.e., course-semester pairs) where the failure rate is higher than 20%. (Assume, for simplicity, that grades are numbers between 1 and 4, and that a failing grade is anything less than 2.)

Because the relational algebra does not have aggregate operators, we must add them to be able to solve the above problem. The additional operator you should use is  $\text{count}_{A/B}(\mathbf{r})$ .

The meaning of this operator is as follows:  $A$  and  $B$  must be lists of attributes in  $\mathbf{r}$ . The *schema* of  $\text{count}_{A/B}(\mathbf{r})$  consists of all attributes in  $B$  plus one additional attribute, which represents the counted value. The *contents* of  $\text{count}_{A/B}(\mathbf{r})$  are defined as follows: for each tuple,  $t \in \pi_B(\mathbf{r})$  (the projection on  $B$ ), take  $\pi_A(\sigma_{B=t}(\mathbf{r}))$  and count the number of tuples in the resulting relation (where  $\sigma_{B=t}(\mathbf{r})$  stands for the set of all tuples in  $\mathbf{r}$  whose value on the attributes in  $B$  is  $t$ ). Let us denote this number by  $c(t)$ . Then the relation  $\text{count}_{A/B}(\mathbf{r})$  is defined as  $\{< t, c(t) > \mid t \in \pi_B(\mathbf{r})\}$ .

You should be able to recognize the above construction as a straightforward adaptation of GROUP BY of SQL to the relational algebra.

- 5.12** State the English meaning of the following algebraic expressions (some of these queries are likely to yield empty results in a typical university, but this is beside the point):

- a.  $\pi_{\text{CrsCode}, \text{Semester}}(\text{TRANSCRIPT}) / \pi_{\text{CrsCode}}(\text{TRANSCRIPT})$
- b.  $\pi_{\text{CrsCode}, \text{Semester}}(\text{TRANSCRIPT}) / \pi_{\text{Semester}}(\text{TRANSCRIPT})$
- c.  $\pi_{\text{CrsCode}, \text{StudId}}(\text{TRANSCRIPT}) / (\pi_{\text{Id}}(\text{STUDENT}))[\text{StudId}]$
- d.  $\pi_{\text{CrsCode}, \text{Semester}, \text{StudId}}(\text{TRANSCRIPT}) / (\pi_{\text{Id}}(\text{STUDENT}))[\text{StudId}]$

- 5.13** Consider the following query:

---

```

SELECT S.Name
FROM STUDENT S, TRANSCRIPT T
WHERE S.Id = T.StudId
AND T.CrsCode IN ('CS305', 'CS315')

```

---

What does this query mean (express the meaning in one short English sentence)? Write an equivalent SQL query without using the IN operator and the set construct.

- 5.14** Explain the conceptual difference between views and bulk insertion of tuples into base relations using the INSERT statement with an attached query.
- 5.15** Explain why a view is like a subroutine.
- 5.16** Write query (5.38) on page 175 without the use of the views.
- 5.17** Express the following queries using SQL. Assume that the STUDENT table is augmented with an additional attribute, Age, and that the PROFESSOR table has additional attributes, Age and Salary.

- a. Find the average age of students who received an A for *some* course.
- b. Find the minimum age among straight A students *per course*.
- c. Find the minimum age among straight A students per course among the students who have taken CS305 or MAT123. (*Hint:* a HAVING clause might help.)
- d. Raise by 10% the salary of every professor who is now younger than 40 and who taught MAT123 in the spring 1997 or fall 1997 semester. (*Hint:* Try a nested subquery in the WHERE clause of the UPDATE statement.)
- e. Find the professors whose salaries are at least 10% higher than the average salary of all professors. (*Hint:* Use views, as in the HARDCLASS example (5.39), page 183.)
- f. Find all professors whose salaries are at least 10% higher than the average salary of all professors *in their departments*. (*Hint:* Use views, as in (5.39).)

**5.18** Express the following queries in relational algebra.

- a. (5.16), page 154
- b. (5.20), page 156
- c. (5.23), page 158

**5.19** Write an equivalent expression in relational algebra for the following SQL query:

---

```
SELECT P.Name, C.Name
FROM PROFESSOR P, COURSE C, TAUGHT T
WHERE P.Id = T.ProfId AND T.Semester = 'S2002'
      AND T.CrsCode = C.CrsCode
```

---

**5.20** Consider the following schema:

---

```
TRANSCRIPT(StudId, CrsCode, Semester, Grade)
TEACHING(ProfId, CrsCode, Semester)
PROFESSOR(Id, ProfName, Dept)
```

---

Write the following query in relational algebra and in SQL: *Find all student Ids who have taken a course from each professor in the MUS Department.*

**5.21** Define the above query as an SQL view and then use this view to answer the following query: *For each student who has taken a course from every professor in the MUS Department, show the number of courses taken, provided that this number is more than 10.*

**5.22** Consider the following schema:

---

```
BROKER(Id, Name) ACCOUNT(Acct#, BrokerId, Gain)
```

---

Write the following query in relational algebra and in SQL: *Find the names of all brokers who have made money in all accounts assigned to them (i.e., Gain > 0).*

**5.23** Write an SQL statement (for the database schema given in Exercise 5.22) to fire all brokers who lost money in at least 40% of their accounts. Assume that every broker has at least one account. (*Hint:* Define intermediate views to facilitate formulation of the query.)

- 5.24 Consider the following schema that represents houses for sale and customers who are looking to buy:

---

```
CUSTOMER(Id, Name, Address)
PREFERENCE(CustId, Feature)
AGENT(Id, AgentName)
HOUSE(Address, OwnerId, AgentId)
AMENITY(Address, Feature)
```

---

PREFERENCE is a relation that lists all features requested by the customers (one tuple per customer/feature; e.g., (123, '5BR'), (123, '2BATH'), (432, 'pool')), and AMENITY is a relation that lists all features of each house (one tuple per house/feature).

A customer is *interested* in buying a house if the set of all features specified by the customer is a subset of the amenities the house has. A tuple in the HOUSE relation states who is the owner and who is the real estate agent listing the house. Write the following queries in SQL:

- a. Find all customers who are interested in every house listed with the agent with Id 007.
  - b. Using the previous query as a view, retrieve a set of tuples of the form *(feature, number\_of\_customers)*, where each tuple in the result shows a feature and the number of customers who want this feature such that
    - Only the customers who are interested in every house listed with Agent 007 are considered.
    - The number of customers interested in *feature* is greater than three. (If this number is not greater than three, the corresponding tuple *(feature, number\_of\_customers)* is not added to the result.)
- 5.25 Consider the schema PERSON(Id, Name, Age). Write an SQL query that finds the 100th oldest person in the relation. A 100th oldest person is one such that there are 99 people who are strictly older. (There can be several such people who might have the same age, or there can be none.)
- 5.26 The last section of this chapter presented four main rules that characterize updatable views in SQL. The third rule states that if the WHERE clause contains a nested subquery, none of the tables mentioned in that subquery (explicitly or implicitly) can be the table used in the FROM clause of the view definition.  
Construct a view that violates condition 3 but satisfies conditions 1, 2, and 4 for upgradability, such that there is an update to this view that has two different translations into the updates on the underlying base relation.
- 5.27 Using the relations TEACHING and PROFESSOR, create a view of TRANSCRIPT containing only rows corresponding to classes taught by John Smyth. Can this view be used in a GRANT statement whose purpose is to allow Smyth to update student grades, but only those he has assigned? Explain.



# 6

## Database Design with the Relational Normalization Theory

Conceptual modeling using the E-R or UML approach is a good way to start dealing with the complexity of modeling a real-world enterprise. However, conceptual modeling is only a set of guidelines that requires considerable expertise and intuition to use successfully, and it can lead to several alternative designs for the same enterprise. Unfortunately, E-R and UML do not provide the criteria or tools to help evaluate alternative designs and suggest improvements. In this chapter, we present the **relational normalization theory**, which includes a set of concepts and algorithms that can help with the evaluation and refinement of the designs obtained through conceptual modeling.

The main tool used in normalization theory is the notion of *functional dependency* (and, to a lesser degree, *join dependency*). Functional dependency is a generalization of the key dependencies in the E-R and UML approaches whereas join dependency does not have a counterpart. Both types of dependency are used by designers to spot situations in which conceptual modeling unnaturally places attributes of two distinct entity types into the same relation schema. These situations are characterized in terms of *normal forms*, from which comes the term “normalization theory.” Normalization theory forces relations into an appropriate normal form using *decompositions*, which break up schemas involving unhappy unions of attributes of unrelated entity types. Because of the central role that decompositions play in relational design, the techniques that we are about to discuss are sometimes also called **relational decomposition theory**.

### 6.1 The Problem of Redundancy

The best way to understand the potential problems with relational designs based on the E-R or UML approach is through an example. Consider the `CREATE TABLE PERSON` statement (4.1) on page 87. Recall that this relation schema was obtained by direct translation from the E-R diagram in Figure 4.1. The first indication of something wrong with this translation was the realization that `SSN` is not a key of the resulting `PERSON` relation. Instead, the key is a combination (`SSN, Hobby`). In other words, the attribute `SSN` does not uniquely identify the tuples in the `PERSON` relation even though it does uniquely identify the entities in the `PERSON` entity set.

Not only is this counterintuitive, but it also has a number of undesirable effects on the instances of the PERSON relation schema.

To see this, we take a closer look at the relation instance shown in Figure 4.13, page 86. Notice that John Doe and Mary Doe are both represented by multiple tuples and that their addresses, names, and IDs occur multiple times as well. Redundant storage of the same information is apparent here. However, wasted space is the least of the problems. The real issue is that when database updates occur, we must keep all the redundant copies of the same data consistent with each other, and we must do it efficiently. Specifically, we can identify the following problems:

- **Update anomaly.** If John Doe moves to 1 Hill Top Drive, updating the relation in Figure 4.13 requires changing the address in both tuples that describe the John Doe entity.
- **Insertion anomaly.** Suppose that we decide to add Homer Simpson to the PERSON relation, but Homer's information sheet does not specify any hobbies. One way around this problem might be to add the tuple (023456789, Homer Simpson, Fox 5 TV, NULL)—that is, to fill in the missing field with NULL. However, Hobby is part of the primary key, and SQL does not allow null values in primary keys. Why? For one thing, DBMSs generally maintain an index on the primary key, and it is not clear how the index should refer to the null value. Assuming that this problem can be solved, suppose that a request is made to insert (023456789, Homer Simpson, Fox 5 TV, acting). Should this new tuple just be added, or should it replace the existing tuple (023456789, Homer Simpson, Fox 5 TV, NULL)? A human will most likely choose to replace it, because humans do not normally think of hobbies as a defining characteristic of a person. However, how does a computer know that the tuples with primary key (111111111, NULL) and (111111111, acting) refer to the same entity? (Recall that the information about which tuple came from which entity is lost in the translation!) Redundancy is at the root of this ambiguity. If Homer were described by at most one tuple, only one course of action would be possible.
- **Deletion anomaly.** Suppose that Homer Simpson is no longer interested in acting. How are we to delete this hobby? We can, of course, delete the tuple that talks about Homer's acting hobby. However, since there is only one tuple that refers to Homer (see Figure 4.13), this throws out perfectly good information about Homer's ID and address. To avoid this loss of information, we can try to replace acting with NULL. Unfortunately, this again raises the issue of nulls in primary key attributes. Once again, redundancy is the culprit. If only one tuple could possibly describe Homer, the attribute Hobby would not be part of the key.

For convenience, we sometimes use the term “update anomalies” to refer to all of the above anomaly types.

## 6.2 Decompositions

The problems caused by redundancy—wasted storage and anomalies—can be fixed using the following simple technique. Instead of having one relation describe all that is known about persons, we can use two separate relation schemas.

---

PERSON1(SSN, Name, Address)  
HOBBY(SSN, Hobby)

---

**6.1**

Projecting the relation in Figure 4.13 on each of these schemas yields the result shown in Figure 6.1. The new design has the following important properties:

1. Assuming for the moment that everyone has hobbies, the original relation of Figure 4.13 is exactly the natural join of the two relations in Figure 6.1. In fact, one can prove that this property is not an artifact of our particular choice of relation instances—if SSN uniquely determines the name and address of a person, then *every* relation,  $r$ , over the schema of Figure 4.13 equals the natural join of the projections of  $r$  on PERSON1 and HOBBY. This property, called *losslessness*, will be discussed in Section 6.6.1. This means that our decomposition preserves the original information represented by the PERSON relation.
2. We no longer have to accept the unnatural result that the Hobby attribute is part of a key. SSN is now the key of both relations. Whereas we could not describe people with no hobbies in PERSON, it is now not even necessary to use nulls for this purpose: we simply do not include rows for such people in the relation HOBBY. Thus, the removal of Bart Simpson's hobbies from the database does not delete the information about his address. The natural join of PERSON1 and HOBBY is no longer PERSON, but then we should not expect this to be the case since we are now describing a more diverse group of people.

SSN	Name	Address
1111111111	John Doe	123 Main St.
5556667777	Mary Doe	7 Lake Dr.
987654321	Bart Simpson	Fox 5 TV

(a) PERSON1

SSN	Hobby
1111111111	stamps
1111111111	hiking
1111111111	coins
5556667777	hiking
5556667777	skating
987654321	acting

(b) HOBBY

**FIGURE 6.1** Decomposition of the PERSON relation shown in Figure 4.13.

SSN	Name	Address	Hobby
111111111	John Doe	123 Main St.	stamps
555666777	Mary Doe	7 Lake Dr.	hiking
987654321	Bart Simpson	Fox 5 TV	coins skating acting

FIGURE 6.2 The “ultimate” decomposition.

- The redundancy present in the original relation of Figure 4.13 is gone and so are the update anomalies. The only items that are stored more than once are SSNs, which are identifiers of entities of type PERSON. Thus, changes to addresses, names, or hobbies now affect only a single tuple. The insertion anomaly is also gone because we can now add people and hobbies independently.

Observe that the new design still has a certain amount of redundancy and that we might still need to use null values in certain cases. First, since we use SSNs as tuple identifiers, each SSN can occur multiple times and all of these occurrences must be kept consistent across the database. So consistency maintenance has not been eliminated completely. However, if the identifiers are not dynamic (for instance, SSNs do not change frequently), consistency maintenance is considerably simplified. Second, imagine a situation in which we add a person to our PERSON1 relation and the address is not known. Clearly, even with the new design, we have to insert NULL in the Address field for the corresponding tuple. However, Address is not part of a primary key, so the use of NULL here is not that bad (we will still have difficulties joining PERSON1 on the Address attribute, though).

It is important to realize that not all decompositions are created equal. In fact, most of them do not make any sense even though they might be doing a good job at eliminating redundancy. The decomposition

---

SSN(SSN)  
NAME(Name)  
ADDRESS(Address)  
HOBBY(Hobby)

---

6.2

is the ultimate “redundancy eliminator.” Projecting of the relation of Figure 4.13 on these schemas yields a database where each value appears exactly once, as shown in Figure 6.2. Unfortunately, this new database is completely devoid of any useful information; for instance, it is no longer possible to tell where John Doe lives or who collects stamps as a hobby. This situation is in sharp contrast with the decomposition of Figure 6.1, where we were able to completely restore the information represented by the original relation using a natural join.

**The need for schema refinement.** Translation of the PERSON entity type into the relational model indicates that one cannot rely solely on conceptual modeling for designing database schemas. Furthermore, the problems exhibited by the PERSON example are by no means rare or unique. Consider the relationship HASACCOUNT of Figure 4.30. A typical translation of the relationship HASACCOUNT of Figure 4.30, page 108, into the relational model might be

---

```
CREATE TABLE HASACCOUNT (
    AccountNumber INTEGER NOT NULL,
    ClientId      CHAR(20),
    OfficeId      INTEGER,
    PRIMARY KEY (ClientId, OfficeId),
    FOREIGN KEY (OfficeId) REFERENCES OFFICE
    ... ... ... )
```

---

6.3

Recall that a client can have at most one account in an office, and hence (ClientId, OfficeId) is a key. Also, an account must be assigned to exactly one office. Careful analysis shows that this requirement leads to some of the same problems that we saw in the PERSON example. For example, a tuple that records the fact that a particular account is managed by a particular office cannot be added without also recording client information (since ClientId is part of the primary key), which is an insertion anomaly. This (and the dual deletion anomaly) is perhaps not a serious problem, because of the specifics of this particular application, but the update anomaly could present maintenance issues. Moving an account from one office to another involves changing OfficeId in every tuple corresponding to that account. If the account has multiple clients, this might be a problem.

We return to this example later in this chapter because HASACCOUNT exhibits certain interesting properties not found in the PERSON example. For instance, even though a decomposition of HASACCOUNT might still be desirable, it incurs additional maintenance overhead that the decomposition of PERSON does not.

The above discussion brings out two key points: (1) Decomposition of relation schemas can serve as a useful tool that complements the E-R approach by eliminating redundancy problems; (2) The criteria for choosing the right decomposition are not immediately obvious, especially when we have to deal with schemas that contain many attributes. For these reasons, the purpose of Sections 6.3 through 6.6 is to develop techniques and criteria for identifying relation schemas that are in need of decomposition as well as to understand what it means for a decomposition not to lose information.

The central tool in developing much of decomposition theory is **functional dependency**, which is a generalization of the idea of key constraints. Functional dependencies are used to define **normal forms**—a set of requirements on relational schemas that are desirable in update-intensive transaction systems. This is why the theory of decompositions is often also called **normalization theory**. Sections 6.7 through 6.9 develop algorithms for carrying out the normalization process.

## 6.3 Functional Dependencies

For the remainder of this chapter we use a special notation for representing attributes, which is common in relational normalization theory. Capital letters from the beginning of the alphabet (e.g.,  $A, B, C, D$ ) represent individual attributes; capital letters from the middle to the end of the alphabet with bars over them (e.g.,  $\bar{P}, \bar{V}, \bar{W}, \bar{X}, \bar{Y}, \bar{Z}$ ) represent sets of attributes. Also, strings of letters, such as  $ABCD$ , denote sets of the respective attributes ( $\{A, B, C, D\}$  in our case); strings of letters with bars over them, (e.g.,  $\bar{X} \bar{Y} \bar{Z}$ ), stand for unions of these sets (i.e.,  $\bar{X} \cup \bar{Y} \cup \bar{Z}$ ). Although this notation requires some getting used to, it is very convenient and provides a succinct language, which we use in examples and definitions.

A **functional dependency** (FD) on a relation schema,  $R$ , is a constraint of the form  $\bar{X} \rightarrow \bar{Y}$ , where  $\bar{X}$  and  $\bar{Y}$  are sets of attributes used in  $R$ . If  $r$  is a relation instance of  $R$ , it is said to **satisfy** this functional dependency if

For every pair of tuples,  $t$  and  $s$ , in  $r$ , if  $t$  and  $s$  agree on all attributes in  $\bar{X}$ , then  $t$  and  $s$  agree on all attributes in  $\bar{Y}$ .

Put another way, there must not be a pair of tuples in  $r$  such that they have the same values for every attribute in  $\bar{X}$  but different values for some attribute in  $\bar{Y}$ .

**Example 6.3.1 (Functional Dependencies).** Consider the relations PERSON1 and HOBBY in Figure 6.1. The FD SSN  $\rightarrow$  Name Address is satisfied by the relation PERSON1. On the other hand, the FD SSN  $\rightarrow$  Hobby is *not* satisfied by the relation HOBBY. Indeed, there are tuples that have the same value 11111111 in the attribute SSN but different values in the attribute Hobby. Similarly, Hobby  $\rightarrow$  SSN is violated by the relation HOBBY: the two tuples that have the value hiking in the Hobby attribute differ in their SSN attribute. ■

Note that the key constraint, introduced in Section 3.2.2, is a special kind of FD. Suppose that  $\text{key}(\bar{K})$  is a key constraint on the relational schema  $R$  and that  $r$  is a relational instance over  $R$ . By definition,  $r$  satisfies  $\text{key}(\bar{K})$  if and only if there is no pair of distinct tuples,  $t, s \in r$ , such that  $t$  and  $s$  agree on every attribute in  $\text{key}(\bar{K})$ . Therefore, this key constraint is equivalent to the FD  $\bar{K} \rightarrow \bar{R}$ , where  $\bar{K}$  is the set of attributes in the key constraint and  $\bar{R}$  denotes the set of all attributes in the schema  $R$ .

Keep in mind that functional dependencies are associated with relation schemas, but when we consider whether or not a functional dependency is satisfied we must consider relation instances over those schemas. This is because FDs are *integrity constraints* on the schema (much like key constraints), which restrict the set of allowable relation instances to those that satisfy the given FDs. It is quite common for some instances of the schema to satisfy the FDs that are not part of that schema. Such satisfaction is considered *accidental* because it is not sanctioned in the schema and thus is not guaranteed to hold in the future.

**Example 6.3.2 (Schema Constraints vs. Accidental FDs).** Consider again the relation PERSON1 in Figure 6.1. In designing a real-life schema for a relation that is

supposed to hold basic information about people, such as PERSON1, we will likely make the FD  $\text{SSN} \rightarrow \text{Name Address}$  part of the schema but will leave out the FDs  $\text{Name} \rightarrow \text{Address}$  and  $\text{Address} \rightarrow \text{SSN}$ . Yet the relation in 6.1(a) *happens* to satisfy both of these FDs.

However, since we did not include these FDs into the schema for PERSON1, nothing precludes us from later adding another John Doe with a different SSN and a different address. Likewise, if we later find out that Mary Doe's child lives at 7 Lake Drive, we will be free to add this new person to the relation PERSON1 without the fear of violating a constraint. ■

To summarize, given a schema,  $R = (\bar{R}; \text{Constraints})$ , where  $\bar{R}$  is a set of attributes and *Constraints* is a set of FDs, a **legal instance** of  $R$  is a relation with the attributes  $\bar{R}$  that satisfies every FD in *Constraints*. We are interested in legal instances because only such relations can exist in a correct database state.

*Brain Teaser:* What does the FD  $X \rightarrow Y$  mean, if  $X$  is an empty set of attributes?

**Functional dependencies and update anomalies.** Certain functional dependencies that exist in a relational schema can lead to redundancy in the corresponding relation instances. Consider the two examples discussed in Sections 6.1 and 6.2: the schemas PERSON and HASACCOUNT. Each has a primary key, as illustrated by the corresponding CREATE TABLE commands (4.1), page 87, and (6.3), page 197, respectively. Correspondingly, there are the following functional dependencies:

---

PERSON:	$\text{SSN}, \text{Hobby} \rightarrow \text{SSN}, \text{Name}, \text{Address}, \text{Hobby}$	<b>6.4</b>
HASACCOUNT:	$\text{ClientId}, \text{OfficeId} \rightarrow \text{AccountNumber},$ $\text{ClientId}, \text{OfficeId}$	

---

These are not the only FDs implied by the original specifications, however. For instance, both Name and Address are defined as single-valued attributes in the E-R diagram of Figure 4.1. This clearly implies that one PERSON entity (identified by its attribute SSN) can have at most one name and one address. Similarly, the business rules of PSSC (the brokerage firm discussed in Section 4.7) require that every account be assigned to exactly one office, which means that the following FDs must also hold for the corresponding relation schemas:

---

PERSON:	$\text{SSN} \rightarrow \text{Name}, \text{Address}$	<b>6.5</b>
HASACCOUNT:	$\text{AccountNumber} \rightarrow \text{OfficeId}$	

---

It is easy to see that the syntactic structure of the dependencies in (6.4) closely corresponds to the update anomalies that we identified for the corresponding relations. For instance, the problem with PERSON is that for any given SSN we cannot change the values for the attributes Name and Address independently of whether the corresponding person has hobbies: if the person has multiple hobbies, the change has to

occur in multiple rows. Likewise with HASACCOUNT we cannot change the value of OfficeId (i.e., transfer an account to a different office) without having to look for all clients associated with this account. Since a number of clients might share the same account, there can be multiple rows in HASACCOUNT that refer to the same account, hence, multiple rows in which OfficeId must be changed.

Note that in both cases, the attributes involved in the update anomalies appear on the left-hand sides of an FD. We can see that update anomalies are associated with certain kinds of functional dependencies. Which dependencies are the bad guys? At the risk of giving away the store, we draw your attention to one major difference between the dependencies in (6.4) and (6.5): the former specify key constraints for their corresponding relations whereas the latter do not.

However, simply knowing which dependencies cause the anomalies is not enough—we must do something about them. We cannot just abolish the offending FDs, because they are part of the semantics of the enterprise being modeled by the database. They are implicitly or explicitly part of the Requirements Document and cannot be changed without an agreement with the customer. On the other hand, we saw that schema decomposition can be a useful tool. Even though a decomposition cannot abolish a functional dependency, it can make it behave. For instance, the decomposition shown in (6.1) on page 195 yields schemas in which the offending FD, SSN → Name, Address, becomes a well-behaved key constraint.

## 6.4 Properties of Functional Dependencies

Before going any further, we need to learn some mathematical properties of functional dependencies and develop algorithms to test them. Since these properties and algorithms rely heavily on the notational conventions introduced at the beginning of Section 6.3, it might be a good idea to revisit these conventions.

The properties of FDs that we are going to study are based on *entailment*. Consider a set of attributes  $\bar{R}$ , a set,  $\mathcal{F}$ , of FDs over  $\bar{R}$ , and another FD,  $f$ , on  $\bar{R}$ . We say that  $\mathcal{F}$  entails  $f$  if every relation  $r$  over the set of attributes  $\bar{R}$  has the following property:

*If  $r$  satisfies every FD in  $\mathcal{F}$ , then  $r$  satisfies the FD  $f$ .*

Given a set of FDs,  $\mathcal{F}$ , the **closure** of  $\mathcal{F}$ , denoted  $\mathcal{F}^+$ , is the set of all FDs entailed by  $\mathcal{F}$ . Clearly,  $\mathcal{F}^+$  contains  $\mathcal{F}$  as a subset.<sup>1</sup>

If  $\mathcal{F}$  and  $\mathcal{G}$  are sets of FDs, we say that  $\mathcal{F}$  entails  $\mathcal{G}$  if  $\mathcal{F}$  entails every individual FD in  $\mathcal{G}$ .  $\mathcal{F}$  and  $\mathcal{G}$  are said to be **equivalent** if  $\mathcal{F}$  entails  $\mathcal{G}$  and  $\mathcal{G}$  entails  $\mathcal{F}$ .

**Algorithm for computing a candidate key.** Why should you care about entailment? For one, you will see that almost all design algorithms discussed in this section

<sup>1</sup> If  $f \in \mathcal{F}$ , then every relation that satisfies every FD in  $\mathcal{F}$  obviously satisfies  $f$ . Therefore, by the definition of entailment,  $f$  is entailed by  $\mathcal{F}$ .

involve entailment in one way or another. Here is an immediate payoff, however—an algorithm for finding a candidate key in a relation schema. Let  $R = (\bar{R}, \mathcal{F})$  be a database schema with the set of attributes  $\bar{R}$  and a set of FDs  $\mathcal{F}$ . Suppose that we know how to effectively check whether  $\mathcal{F}$  entails an arbitrary FD (we will develop such an algorithm later). Then we can find a candidate key of  $R$  as follows: Pick up an arbitrary attribute,  $A \in \bar{R}$ , and check if  $\mathcal{F}$  entails  $(\bar{R} - A) \rightarrow \bar{R}$ . If it does, we know that some candidate key hides inside  $(\bar{R} - A)$ . Remove another arbitrary attribute from what was left (let us denote the remaining set by  $X$ ) and test that  $X \rightarrow \bar{R}$  is still entailed by  $\mathcal{F}$ . If it does not, choose another attribute and test the same. Continue in this way trying to remove more and more attributes. The algorithm terminates when you cannot remove any attribute and still have  $X \rightarrow \bar{R}$  entailed by  $\mathcal{F}$ . The remaining set of attributes must be a key.

The above algorithm works from the top down by eliminating attributes. There is also a bottom-up counterpart: start with an empty set of attributes and keep adding attributes until you find an  $X$  such that  $X \rightarrow \bar{R}$  is entailed by  $\mathcal{F}$ . Can we find all keys in this way? The answer is yes, but not so quickly. There can be an exponential number of such keys, and insisting on finding all of them can cost you a lunch, dinner, and the next breakfast. One way, which is just slightly better than brute force, is to order all subsets of  $\bar{R}$  by the amount of attributes in them and systematically apply the top-down algorithm or the bottom-up one until you cannot proceed any further. Each instance of the algorithm will discover some key. All the instances together will find all keys.

We now present several simple but important properties of entailment and later develop an algorithm for testing entailment.

**Reflexivity.** Some FDs are satisfied by every relation no matter what. These dependencies all have the form  $\bar{X} \rightarrow \bar{Y}$ , where  $\bar{Y} \subseteq \bar{X}$ , and are called **trivial FDs**.

- The reflexivity property states that, if  $\bar{Y} \subseteq \bar{X}$ , then  $\bar{X} \rightarrow \bar{Y}$ .

To see why trivial FDs are always satisfied, consider a relation,  $r$ , whose set of attributes includes all of the attributes mentioned in  $\bar{X}$ . Suppose that  $t, s \in r$  are tuples that agree on  $\bar{X}$ . But, since  $\bar{Y} \subseteq \bar{X}$ , this means that  $t$  and  $s$  agree on  $\bar{Y}$  as well. Thus,  $r$  satisfies  $\bar{X} \rightarrow \bar{Y}$ .

We can now relate trivial FDs and entailment. Because a trivial FD is satisfied by every relation, it is entailed by every set of FDs (even the empty set)! In particular,  $\mathcal{F}^+$  contains every trivial FD.

**Augmentation.** Consider an FD,  $\bar{X} \rightarrow \bar{Y}$ , and another set of attributes,  $\bar{Z}$ . Let  $\bar{R}$  contain  $\bar{X} \cup \bar{Y} \cup \bar{Z}$ . Then  $\bar{X} \rightarrow \bar{Y}$  entails  $\bar{X}\bar{Z} \rightarrow \bar{Y}\bar{Z}$ . In other words, every relation  $r$  over  $\bar{R}$  that satisfies  $\bar{X} \rightarrow \bar{Y}$  must also satisfy the FD  $\bar{X}\bar{Z} \rightarrow \bar{Y}\bar{Z}$ .

- The augmentation property states that, if  $\bar{X} \rightarrow \bar{Y}$ , then  $\bar{X}\bar{Z} \rightarrow \bar{Y}\bar{Z}$ .

To see why this is true, observe that if tuples  $t, s \in r$  agree on every attribute of  $\bar{X}\bar{Z}$  then in particular they agree on  $\bar{X}$ . Since  $r$  satisfies  $\bar{X} \rightarrow \bar{Y}$ ,  $t$  and  $s$  must also

agree on  $\bar{Y}$ . They also agree on  $\bar{Z}$ , since we have assumed that they agree on a bigger set of attributes,  $\bar{X}\bar{Z}$ . Thus, if  $s, t$  agree on every attribute in  $\bar{X}\bar{Z}$ , they must agree on  $\bar{Y}\bar{Z}$ . As this is an arbitrarily chosen pair of tuples in  $r$ , it follows that  $r$  satisfies  $\bar{X}\bar{Z} \rightarrow \bar{Y}\bar{Z}$ .

**Transitivity.** The set of FDs  $\{\bar{X} \rightarrow \bar{Y}, \bar{Y} \rightarrow \bar{Z}\}$  entails the FD  $\bar{X} \rightarrow \bar{Z}$ .

- The transitivity property states that, if  $\bar{X} \rightarrow \bar{Y}$  and  $\bar{Y} \rightarrow \bar{Z}$ , then  $\bar{X} \rightarrow \bar{Z}$ .

This property can be established similarly to the previous two (see the exercises at the end of the chapter).

These three properties of FDs are known as **Armstrong's axioms**; they are typically used as *inference rules* in the proofs of correctness of various database design algorithms. However, they are also a powerful tool used by (real, breathing, human) database designers because they can help spot problematic FDs in relational schemas. We now show how Armstrong's axioms are used to derive new FDs.

**Union of FDs.** Any relation,  $r$ , that satisfies  $\bar{X} \rightarrow \bar{Y}$  and  $\bar{X} \rightarrow \bar{Z}$  must also satisfy  $\bar{X} \rightarrow \bar{Y}\bar{Z}$ . To show this, we can derive  $\bar{X} \rightarrow \bar{Y}\bar{Z}$  from  $\bar{X} \rightarrow \bar{Y}$  and  $\bar{X} \rightarrow \bar{Z}$  using simple syntactic manipulations defined by Armstrong's axioms. Such manipulations can be easily programmed on a computer, unlike the tuple-based considerations we used to establish the axioms themselves. Here is how it is done:

- (a)  $\bar{X} \rightarrow \bar{Y}$  Given
- (b)  $\bar{X} \rightarrow \bar{Z}$  Given
- (c)  $\bar{X} \rightarrow \bar{Y}\bar{X}$  Adding  $\bar{X}$  to both sides of (a): Armstrong's augmentation rule
- (d)  $\bar{Y}\bar{X} \rightarrow \bar{Y}\bar{Z}$  Adding  $\bar{Y}$  to both sides of (b): Armstrong's augmentation rule
- (e)  $\bar{X} \rightarrow \bar{Y}\bar{Z}$  By Armstrong's transitivity rule, applied to (c) and (d)

**Decomposition of FDs.** In a similar way, we can prove the following rule: every relation that satisfies  $\bar{X} \rightarrow \bar{Y}\bar{Z}$  must also satisfy the FDs  $\bar{X} \rightarrow \bar{Y}$  and  $\bar{X} \rightarrow \bar{Z}$ . This is accomplished by the following simple steps:

- (a)  $\bar{X} \rightarrow \bar{Y}\bar{Z}$  Given
- (b)  $\bar{Y}\bar{Z} \rightarrow \bar{Y}$  By Armstrong's reflexivity rule, since  $\bar{Y} \subseteq \bar{Y}\bar{Z}$
- (c)  $\bar{X} \rightarrow \bar{Y}$  By transitivity from (a) and (b)

Derivation of  $\bar{X} \rightarrow \bar{Z}$  is similar.

**Brain Teaser:** What kind of FD is  $X \rightarrow Y$ , if  $Y$  is an empty set of attributes?

Armstrong's axioms are obviously *sound*. By **sound** we mean that any expression of the form  $\bar{X} \rightarrow \bar{Y}$  derived using the axioms from a set of FDs  $\mathcal{F}$  is actually a functional dependency that holds in any relation that satisfies every FD in  $\mathcal{F}$ . Soundness follows from the fact that we have proved that these inference rules

are valid for every relation. It is much less obvious, however, that they are also **complete**—that is, if a set of FDs,  $\mathcal{F}$ , entails another FD,  $f$ , then  $f$  can be derived from  $\mathcal{F}$  by a sequence of steps, similar to the ones above, that rely solely on Armstrong's axioms! For the curious, we provide a proof of this fact at the end of this section.

Note that the definition of entailment of FDs on page 200 does not even hint at an algorithm for checking entailment (i.e., for testing whether  $f \in \mathcal{F}^+$ ). The definition is completely *semantic* in nature and, according to the definition, testing  $f \in \mathcal{F}^+$  involves perusing an infinite number of relations. In contrast, Armstrong's axioms provide *syntactic* manipulations that can be carried out by a computer.

Of course, not all syntactic manipulations make sense. For instance, deleting attribute  $A$  from every FD does not. However, if the manipulations are sound and complete, then using them is *correct* and is *equivalent* to using the definition of entailment. Thus, soundness and completeness of Armstrong's axioms is not just a theoretical curiosity—this result has considerable practical value because it guarantees that entailment of FDs can be verified by a computer program. We are now going to develop one such algorithm (and a better one later).

**Naive algorithm for checking entailment.** An obvious way to verify entailment of an FD,  $f$ , by a set of FDs,  $\mathcal{F}$ , is to instruct the computer to apply Armstrong's axioms to  $\mathcal{F}$  in all possible ways. Since the number of attributes mentioned in  $\mathcal{F}$  and  $f$  is finite, this derivation process cannot go on forever. When we are satisfied that all possible derivations have been made, we can simply check whether  $f$  is among the FDs derived by this process. Completeness of Armstrong's axioms guarantees that  $f \in \mathcal{F}^+$  if and only if  $f$  is one of the FDs thus derived.

**Example 6.4.1 (Entailment Checking with Armstrong's Axioms).** To see how this process works, consider the following sets of FDs:  $\mathcal{F} = \{AC \rightarrow B, A \rightarrow C, D \rightarrow A\}$  and  $\mathcal{G} = \{A \rightarrow B, A \rightarrow C, D \rightarrow A, D \rightarrow B\}$ . We can use Armstrong's axioms to prove that these two sets are equivalent, that is, that every FD in  $\mathcal{G}$  is entailed by  $\mathcal{F}$ , and vice versa. For instance, to prove that  $A \rightarrow B$  is implied by  $\mathcal{F}$ , we can apply Armstrong's axioms in all possible ways. Most of these attempts will not lead anywhere, but a few will. For instance, the following derivation establishes the desired entailment:

- (a)  $A \rightarrow C$  An FD in  $\mathcal{F}$
- (b)  $A \rightarrow AC$  From (a) and Armstrong's augmentation axiom
- (c)  $A \rightarrow B$  From (b),  $AC \rightarrow B \in \mathcal{F}$ , and Armstrong's transitivity axiom

The FDs  $A \rightarrow C$  and  $D \rightarrow A$  belong to both  $\mathcal{F}$  and  $\mathcal{G}$ , so the derivation is trivial. For  $D \rightarrow B$  in  $\mathcal{G}$ , the computer can try to apply Armstrong's axioms until this FD is derived. After awhile, it will stumble upon this valid derivation:

- (a)  $D \rightarrow A$  an FD in  $\mathcal{F}$
- (b)  $A \rightarrow B$  derived previously
- (c)  $D \rightarrow B$  from (a), (b), and Armstrong's transitivity axiom

This shows that every FD in  $\mathcal{F}$  entailed by  $\mathcal{G}$  is done similarly. ■

Although the simplicity of checking entailment by blindly applying Armstrong's axioms is attractive, it is not very efficient. In fact, the size of  $\mathcal{F}^+$  can be exponential in the size of  $\mathcal{F}$ , so for large database schemas it can take a very long time before the designer ever sees the result. We are therefore going to develop a more efficient algorithm, which is also based on Armstrong's axioms but which applies them much more judiciously.

**Checking entailment of FDs using attribute closure.** The idea of the new algorithm for verifying entailment is based on the concept of *attribute closure*.

Given a set of FDs,  $\mathcal{F}$ , and a set of attributes,  $\bar{X}$ , we define the **attribute closure** of  $\bar{X}$  with respect to  $\mathcal{F}$ , denoted  $\bar{X}_{\mathcal{F}}^+$ , as follows:

$$\bar{X}_{\mathcal{F}}^+ = \{A \mid \bar{X} \rightarrow A \in \mathcal{F}^+\}$$

In other words,  $\bar{X}_{\mathcal{F}}^+$  is a set of all those attributes,  $A$ , such that  $\bar{X} \rightarrow A$  is entailed by  $\mathcal{F}$ . Note that  $\bar{X} \subseteq \bar{X}_{\mathcal{F}}^+$  because, if  $A \in \bar{X}$ , then, by Armstrong's reflexivity axiom,  $\bar{X} \rightarrow A$  is a trivial FD that is entailed by every set of FDs, including  $\mathcal{F}$ .

It is important to keep in mind that the closure of  $\mathcal{F}$  (i.e.,  $\mathcal{F}^+$ ) and the closure of  $\bar{X}$  (i.e.,  $\bar{X}_{\mathcal{F}}^+$ ) are related but *very different* notions:  $\mathcal{F}^+$  is a set of functional dependencies, whereas  $\bar{X}_{\mathcal{F}}^+$  is a set of attributes.

Always true:  $\bar{X} \subseteq \bar{X}_{\mathcal{F}}^+$  and  $\mathcal{F} \subseteq \mathcal{F}^+$

**Example 6.4.2 (Attribute Closure).** Let  $\mathcal{F} = \{B \rightarrow E, C \rightarrow F, BD \rightarrow G\}$ . Then the attribute closure of  $ABC$  with respect to  $\mathcal{F}$  is  $ABCEF$ .

To see this, you can verify by direct inspection that  $\mathcal{F}$  entails  $ABCEF \rightarrow x$ , where  $x$  is  $A$ ,  $B$ ,  $C$ ,  $E$ , or  $F$ . You can also verify that  $\mathcal{F}$  does *not* entail  $ABCEF \rightarrow y$ , where  $y$  is any other attribute. A standard method to prove such a negative result is to construct the following relation as a counterexample:

$A$	$B$	$C$	$E$	$D$	$G$	$\dots$	$y$	$\dots$	
0	0	0	0	0	0	...	0	...	0
0	0	0	0	1	1	...	1	...	1

It obviously does not satisfy  $ABCEF \rightarrow y$ , but it does satisfy every FD in  $\mathcal{F}$ . Therefore  $\mathcal{F}$  does not entail  $ABCEF \rightarrow y$ . ■

If we knew how to compute attribute closure efficiently, we could check FD entailment using the following algorithm: Given a set of FDs,  $\mathcal{F}$ , and an FD,  $\bar{X} \rightarrow \bar{Y}$ , check whether  $\bar{Y} \subseteq \bar{X}_{\mathcal{F}}^+$ . If this is so, then  $\mathcal{F}$  entails  $\bar{X} \rightarrow \bar{Y}$ . Otherwise, if  $\bar{Y} \not\subseteq \bar{X}_{\mathcal{F}}^+$ , then  $\mathcal{F}$  does not entail  $\bar{X} \rightarrow \bar{Y}$ .

The correctness of this algorithm follows from Armstrong's axioms. If  $\bar{Y} \subseteq \bar{X}_{\mathcal{F}}^+$ , then  $\bar{X} \rightarrow A \in \mathcal{F}^+$  for every  $A \in \bar{Y}$  (by the definition of  $\bar{X}_{\mathcal{F}}^+$ ). By the union rule for FDs,

**FIGURE 6.3** Computation of attribute closure  $\overline{X}_{\mathcal{F}}^+$ .

```

closure :=  $\overline{X}$ 
repeat
  old := closure
  if there is an FD  $\overline{Z} \rightarrow \overline{V} \in \mathcal{F}$  such that  $\overline{Z} \subseteq closure$  and  $\overline{V} \not\subseteq closure$  then
    closure := closure  $\cup \overline{V}$ 
  until old = closure
  return closure

```

---

it follows that  $\mathcal{F}$  entails  $\overline{X} \rightarrow \overline{Y}$ . Conversely, if  $\overline{Y} \not\subseteq \overline{X}_{\mathcal{F}}^+$ , then there is  $B \in \overline{Y}$  such that  $B \notin \overline{X}_{\mathcal{F}}^+$ . Hence,  $\overline{X} \rightarrow B$  is not entailed by  $\mathcal{F}$ . But then  $\mathcal{F}$  cannot entail  $\overline{X} \rightarrow \overline{Y}$ . If it did, it would have to entail  $\overline{X} \rightarrow B$  as well, by the decomposition rule for FDs.

The heart of the above algorithm is a check of whether a set of attributes belongs to  $\overline{X}_{\mathcal{F}}^+$ . Therefore, we are not done yet. We need an algorithm for computing the closure of  $\overline{X}$ , which we present in Figure 6.3. The idea behind the algorithm is to enlarge the set of attributes known to belong to  $\overline{X}_{\mathcal{F}}^+$  by applying the FDs in  $\mathcal{F}$ . The closure is initialized to  $\overline{X}$ , since we know that  $\overline{X}$  is always a subset of  $\overline{X}_{\mathcal{F}}^+$ .

The soundness of the algorithm can be proved by induction. Initially,  $closure$  is  $\overline{X}$ , so  $\overline{X} \rightarrow closure$  is in  $\mathcal{F}^+$ . Then, assuming that  $\overline{X} \rightarrow closure \in \mathcal{F}^+$  at some intermediate step in the **repeat** loop of Figure 6.3, and given an FD  $\overline{Z} \rightarrow \overline{V} \in \mathcal{F}$  such that  $\overline{Z} \subset closure$ , we can use the *generalized transitivity rule* (see Exercise 6.10) to infer that  $\mathcal{F}$  entails  $\overline{X} \rightarrow closure \cup \overline{V}$ . Thus, if  $A \in closure$  at the end of the computation, then  $A \in \overline{X}_{\mathcal{F}}^+$ . The converse is also true: if  $A \in \overline{X}_{\mathcal{F}}^+$ , then at the end of the computation  $A \in closure$  (see Exercise 6.11).

Unlike the simple-minded algorithm that uses Armstrong's axioms indiscriminately, the run-time complexity of the algorithm in Figure 6.3 is quadratic in the size of  $\mathcal{F}$ . In fact, an algorithm for computing  $\overline{X}_{\mathcal{F}}^+$  that is *linear* in the size of  $\mathcal{F}$  is given in [Beeri and Bernstein 1979]. This algorithm is better suited for a computer program, but its inner workings are more complex.

**Example 6.4.3 (Checking Entailment).** Consider a relational schema,  $R = (\bar{R}; \mathcal{F})$ , where  $\bar{R} = ABCDEFGHIJ$ , and the set of FDs,  $\mathcal{F}$ , which contains the following FDs:  $AB \rightarrow C$ ,  $D \rightarrow E$ ,  $AE \rightarrow G$ ,  $GD \rightarrow H$ ,  $ID \rightarrow J$ . We wish to check whether  $\mathcal{F}$  entails  $ABD \rightarrow GH$  and  $ABD \rightarrow HJ$ .

First, let us compute  $ABD_{\mathcal{F}}^+$ . We begin with  $closure = ABD$ . Two FDs can be used in the first iteration of the loop in Figure 6.3. For definiteness, let us use  $AB \rightarrow C$ , which makes  $closure = ABDC$ . In the second iteration, we can use  $D \rightarrow E$ , which makes  $closure = ABDCE$ . Now it becomes possible to use the FD  $AE \rightarrow G$  in the third iteration, yielding  $closure = ABDCEG$ . This in turn allows  $GD \rightarrow H$  to be applied in the fourth iteration, which results in  $closure = ABDCEGH$ . In the fifth iteration, we cannot apply any new FDs, so  $closure$  does not change and the loop terminates. Thus,  $ABD_{\mathcal{F}}^+ = ABDCEGH$ .

**FIGURE 6.4** Testing equivalence of sets of FDs.

```

Input:  $\mathcal{F}, \mathcal{G}$  – FD sets
Output: true, if  $\mathcal{F}$  is equivalent to  $\mathcal{G}$ ; false otherwise
for each  $f \in \mathcal{F}$  do
    if  $\mathcal{G}$  does not entail  $f$  then return false
for each  $g \in \mathcal{G}$  do
    if  $\mathcal{F}$  does not entail  $g$  then return false
return true

```

---

Since  $GH \subseteq ABDCEGH$ , we conclude that  $\mathcal{F}$  entails  $ABD \rightarrow GH$ . On the other hand,  $HJ \not\subseteq ABDCEGH$ , so we conclude that  $ABD \rightarrow HJ$  is not entailed by  $\mathcal{F}$ . Note, however, that  $\mathcal{F}$  does entail  $ABD \rightarrow H$ . ■

The above algorithm for testing entailment leads to a simple test for equivalence between a pair of sets of FDs. Let  $\mathcal{F}$  and  $\mathcal{G}$  be such sets. To check that they are equivalent, we must check that every FD in  $\mathcal{G}$  is entailed by  $\mathcal{F}$ , and vice versa. The algorithm is depicted in Figure 6.4.

**Proof of completeness of Armstrong's axioms.\*** Suppose that an FD  $f : \bar{X} \rightarrow A$  is entailed by a set of FDs  $\mathcal{F}$ , but there is no derivation of  $f$  from  $\mathcal{F}$  based on Armstrong's axioms. We will show that then there must be a relation,  $r$ , that satisfies every FD in  $\mathcal{F}$ , but not  $f$ . This would be a contradiction since we assumed that  $f$  is entailed by  $\mathcal{F}$ .

Let the relation  $r$  consist of just two tuples,  $t_0$  and  $t_1$ , where  $t_0$  has 0 in every position and  $t_1$  has 0 for every attribute in the set

$$\hat{X} = \{B \mid \bar{X} \rightarrow B \text{ can be derived from } \mathcal{F} \text{ using Armstrong's axioms}\}$$

and 1 everywhere else:

	$\hat{X}$	other attributes
$t_0$ :	0 0 0 0 0	0 0 0 0 0 0 ...
$t_1$ :	0 0 0 0 0	1 1 1 1 1 1 ...

Note that the set  $\hat{X}$  is similar to the attribute closure  $\bar{X}_{\mathcal{F}}^+$  except that instead of using every  $\bar{X} \rightarrow B$  that is *entailed* by  $\mathcal{F}$  we consider only those that are *derivable* from  $\mathcal{F}$ . Because of the soundness of Armstrong's axioms, every derivable FD is entailed by  $\mathcal{F}$ , so  $\hat{X} \subseteq \bar{X}_{\mathcal{F}}^+$ . In fact, the two sets are equal, but we do not know this yet because we have not proved the completeness of the axioms.

Let us note a few simple properties of  $\hat{X}$ . First,  $\bar{X} \subseteq \hat{X}$ , because  $\bar{X} \rightarrow B$  is Armstrong's reflexivity axiom for every  $B \in \bar{X}$ . Second, since we assumed that  $f : X \rightarrow A$  is not derivable from  $\mathcal{F}$ , the attribute  $A$  is not in  $\hat{X}$ . Therefore,  $t_0$  and  $t_1$  have different

values over  $A$  and so  $f$  is violated in  $r$ . Thus, if we show that every FD in  $\mathcal{F}$  holds in  $r$ , then we will arrive at a contradiction with the assumption that  $\mathcal{F}$  entails  $f$ .

So, let us prove that every  $g : \bar{Y} \rightarrow C \in \mathcal{F}$  holds in  $r$ .

- If  $\bar{Y} \subseteq \hat{X}$  then  $C$  must be in  $\hat{X}$ . Indeed,  $\bar{Y} \subseteq \hat{X}$  implies that the FD  $\bar{X} \rightarrow \bar{Y}$  is derivable from  $\mathcal{F}$  using Armstrong's axioms. (Proving this is left as a simple exercise.) Therefore,  $\bar{X} \rightarrow C$  can be derived from  $\bar{X} \rightarrow \bar{Y}$  and  $g$  by Armstrong's transitivity rule. By the definition of  $\hat{X}$ ,  $C$  must be in  $\hat{X}$ ; and since  $t_0$  and  $t_1$  have the same value 0 over all attributes of  $\hat{X}$ , it follows that the FD  $g$  is satisfied in  $r$ .
- If  $\bar{Y} \not\subseteq \hat{X}$  then the tuples  $t_0$  and  $t_1$  have different values over some attributes in  $\bar{Y}$ , so they do not have to agree over  $C$ . In this case,  $g$  is satisfied in  $r$  in a trivial manner.

Since  $g$  was chosen arbitrarily, the above establishes that every FD in  $\mathcal{F}$  is satisfied by  $r$ , which completes the proof.

## 6.5 Normal Forms

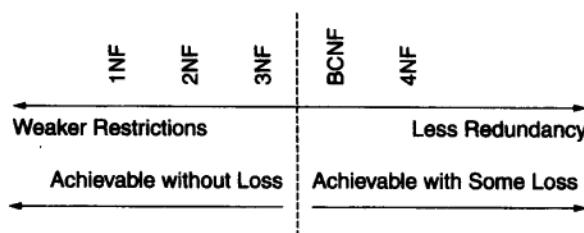
To eliminate redundancy and potential update anomalies, database theory identifies several *normal forms* for relational schemas such that, if a schema is in one of the normal forms, it has certain predictable properties. Each normal form is characterized by a set of restrictions. Thus, for a schema to be in a normal form it must satisfy the restrictions associated with that form. Originally, [Codd 1970] proposed three normal forms, each successively imposing more restrictions and eliminating more and more anomalies and redundancies than the previous one.

The **first normal form (1NF)**, as introduced by Codd, is equivalent to the definition of the relational data model. In particular, the value of an attribute must be atomic. It cannot be anything that has structure, such as a record (with multiple fields) or a set. The **second normal form (2NF)** says that a schema must not have an FD,  $X \rightarrow Y$ , where  $X$  is a strict subset of that schema's key. This normal form is of no practical use, and we do not discuss it any further.

The **third normal form (3NF)** was initially thought to be the “ultimate” normal form. However, Boyce and Codd soon realized that 3NF can still harbor undesirable combinations of functional dependencies, so they introduced the **Boyce-Codd normal form (BCNF)**. Unfortunately, there is rarely a free lunch in computational sciences. Even though BCNF is more desirable, it is not always achievable without paying a price elsewhere. In this section, we define both BCNF and 3NF. Subsequent sections in this chapter develop algorithms for automatically converting relational schemas that possess various bad properties into sets of schemas in 3NF and BCNF. We also study the trade-offs associated with such conversions.

In Section 6.9, we show that certain types of redundancy are caused by dependencies other than the FDs. To deal with this problem, we introduce the **fourth normal form (4NF)**, which further extends BCNF.

**FIGURE 6.5** Relationship among normal forms.



The relationship between the different normal forms is depicted in Figure 6.5. The higher normal forms (to the right) impose more restrictions on relational schemas, and this ensures that the corresponding relations have less redundant information. Note that as we go to the left, restrictions become weaker, and thus any relation schema that satisfies the conditions of a higher normal form also satisfies the conditions of the lower. The figure also shows that the lower normal forms can always be achieved, while the higher ones can be achieved only at a certain price. You will learn what all this means in the remainder of this chapter.

### 6.5.1 The Boyce-Codd Normal Form

A relational schema,  $R = (\bar{R}; \mathcal{F})$ , where  $\bar{R}$  is the set of attributes of  $R$  and  $\mathcal{F}$  is the set of functional dependencies associated with  $R$ , is in Boyce-Codd normal form if, for every FD  $\bar{X} \rightarrow \bar{Y} \in \mathcal{F}$ , either of the following is true:

- $\bar{Y} \subseteq \bar{X}$  (i.e., this is a trivial FD).
- $\bar{X}$  is a superkey of  $R$ .

In other words, the only nontrivial FDs are those in which a key functionally determines one or more attributes.

**Examples.** It is easy to see that PERSON1 and HOBBY, the relational schemas given in (6.1), are in BCNF, because the only nontrivial FD is  $SSN \rightarrow Name, Address$ . It applies to PERSON1, which has SSN as a key.

On the other hand, consider the schema PERSON defined by the CREATE TABLE statement (4.1), page 87, and the schema HASACCOUNT defined by the SQL statement (6.3), page 197. As discussed earlier, these statements fail to capture some important relationships, which are represented by the FDs in (6.5), page 199. Each of these FDs is in violation of the requirement to be in BCNF. They are not trivial, and their left-hand sides— $SSN$  and  $AccountNumber$ —are not keys of their respective schemas.

**Properties of BCNF.** Note that a BCNF schema can have more than one key. For instance,  $R = (ABCD; \mathcal{F})$ , where  $\mathcal{F} = \{AB \rightarrow CD, AC \rightarrow BD\}$  has two keys,  $AB$  and  $AC$ . And yet it is in BCNF because the left-hand side of each of the two FDs in  $\mathcal{F}$  is a key.

Observe that we defined BCNF by looking only at the set of FDs in  $\mathcal{F}$ , not  $\mathcal{F}^+$ . One might wonder, therefore, whether  $\mathcal{F}^+ - \mathcal{F}$  might have any FDs that violate BCNF, in which case this normal form will not make much sense. Fortunately, the above situation cannot occur. To see this, consider an arbitrary FD  $\bar{X} \rightarrow A \in (\mathcal{F}^+ - \mathcal{F})$  such that  $A \notin \bar{X}$ . As we know, it must be the case that  $A \in \bar{X}_g^+$  and  $\bar{X}_g^+$  should be computable by the attribute closure algorithm in Figure 6.3. Recall that the main step in that algorithm hinges on being able to find an FD  $\bar{Z} \rightarrow \bar{V} \in \mathcal{F}$  such that  $\bar{Z} \subseteq \text{closure}$ . Since initially  $\text{closure} = \bar{X}$ , there must be an FD in  $\mathcal{F}$  whose left-hand side,  $\bar{Z}$ , is a subset of  $\bar{X}$ . But since the schema is in BCNF,  $\bar{Z}$  must be a superkey. Hence, so must be  $\bar{X}$ , that is, the FD  $\bar{X} \rightarrow A$  does not violate the BCNF conditions.

**Nonredundancy of BCNF.** An important property of BCNF schemas is that their instances do not contain redundant information that arises due to FDs. Since we have been illustrating redundancy problems only through concrete examples, the above statement might seem vague. Exactly what is redundant information? For instance, does the abstract relation

A	B	C	D
1	1	3	4
2	1	3	4

over the above-mentioned BCNF schema R store redundant information?

Superficially it might seem so because the two tuples agree on all but one attribute. However, having identical values in some attributes of different tuples does not necessarily imply that the tuples are storing redundant information. Redundancy arises when the values of some set of attributes,  $\bar{X}$ , necessarily implies the value that must exist in another attribute,  $A$ —a functional dependency. If two distinct tuples have the same values in  $\bar{X}$ , they must have the same value of  $A$ . This means that an association between  $A$  and the attributes in  $\bar{X}$  is stored multiple times. Redundancy is eliminated if we store such an association only once (in a separate relation) instead of repeating it in all tuples of an instance of the schema R that agree on  $\bar{X}$ . Since R does not have FDs over the attributes  $BCD$ , no redundant information is stored. The fact that the tuples in the relation coincide over  $BCD$  is coincidental. For instance, the value of attribute  $D$  in the first tuple can be changed from 4 to 5 without regard for the second tuple.

A DBMS automatically eliminates one type of redundancy. Two tuples with the same values in the key fields are prohibited in any instance of a schema. This is a special case. The key identifies an entity and so determines the values of all attributes describing that entity. As the definition of BCNF precludes associations that do not contain keys, the relations over BCNF schemas do not store redundant information. As a result, deletion and update anomalies do not arise in BCNF relations.

Relations with more than one key still can have insertion anomalies. To see this, suppose that associations over  $ABD$  and over  $ACD$  are added to our relation as shown:

A	B	C	D
1	1	3	4
2	1	3	4
3	4	NULL	5
3	NULL	2	5

Because the value over the attribute  $C$  in the first association and over  $B$  in the second is unknown, we fill in the missing information with NULL. However, now we cannot tell if the two newly added tuples are the same—it all depends on the real values for the nulls. A practical solution to this problem, as adopted by the SQL standard, is to designate one key as *primary* and to prohibit null values in its attributes. Under this restriction, every tuple is defined over the attributes of the primary key and, in particular, the above situation with newly added tuples is impossible.

### 6.5.2 The Third Normal Form

A relational schema,  $R = (\bar{R}; \mathcal{F})$ , where  $\bar{R}$  is the set of attributes of  $R$  and  $\mathcal{F}$  is the set of functional dependencies associated with  $R$ , is in **third normal form** if, for every FD  $\bar{X} \rightarrow \bar{Y} \in \mathcal{F}$ , any of the following conditions are true:

- $\bar{Y} \subseteq \bar{X}$  (i.e., this is a trivial FD).
- $\bar{X}$  is a superkey of  $R$ .
- Each attribute in  $A \in \bar{Y} - \bar{X}$  belongs to some candidate key,  $\bar{K}$ , of  $R$ .

Observe that the first two conditions in the definition of 3NF are identical to the conditions that define BCNF. Thus, 3NF is a relaxation of BCNF's requirements. Every schema that is in BCNF must also be in 3NF, but the converse is not true in general. For instance, the relation HASACCOUNT (6.3) on page 197 is in 3NF because the only FD that is not based on a key constraint is  $\text{AccountNumber} \rightarrow \text{OfficeId}$ , and  $\text{OfficeId}$  is part of the key. However, this relation is not in BCNF, as shown previously.<sup>2</sup>

If you are wondering about the intrinsic merit of the third condition in the definition of 3NF, the answer is that there is none. In a way, 3NF was discovered by accident—in the search for what we now call BCNF! The reason for the remarkable survival of 3NF is that it was later found to have some very desirable algorithmic properties, which BCNF does not possess. We discuss these issues in subsequent sections.

<sup>2</sup> In fact, HASACCOUNT is the *smallest* possible example of a 3NF relation that is not in BCNF (see Exercise 6.5).

**Brain Teaser:** Find a 2NF relation that is not in 3NF.

**Redundancy in 3NF.** Recall from Section 6.2 that relation instances over HAS-ACCOUNT might store redundant information. Now we can see that this redundancy arises because of the functional dependency that relates AccountNumber and OfficeId and that is not implied by key constraints.

For another example, consider the schema PERSON discussed earlier. This schema violates the 3NF requirements because, for example, the FD SSN → Name is not based on a key constraint (SSN is not a superkey) and Name does not belong to a key of PERSON. However, the decomposition of this schema into PERSON1 and HOBBY in (6.1), page 195, yields a pair of schemas that are in both 3NF and BCNF.

We can ask the same question as in the case of BCNF: is it possible that some FD in  $F^+ - F$  violates the 3NF conditions, thereby making this normal form ill-defined? It is easy to guess that the answer is “no,” for, otherwise, 3NF would not be worth including in a textbook. However, proving this is a notch harder than in the case of BCNF, so we leave it to Exercise 6.25.

## 6.6 Properties of Decompositions

Since there is no redundancy in BCNF schemas and redundancy in 3NF is limited, we are interested in decomposing a given schema into a collection of schemas, each of which is in one of these normal forms.

The main thrust of the discussion in the previous section was that 3NF does not completely solve the redundancy problem. Therefore, at first glance, there appears to be no justification to consider 3NF as a goal for database design. It turns out, however, that the maintenance problems associated with redundancy do not show the whole picture. As we will see, maintenance is also associated with integrity constraints,<sup>3</sup> and 3NF decompositions sometimes have better properties in this regard than do BCNF decompositions. Our first step is to define these properties.

Recall from Section 6.2 that not all decompositions are created equal. For instance, the decomposition of PERSON shown in (6.1) is considered good while the one in (6.2) makes no sense. Is there an objective way to tell which decompositions make sense and which do not, and can this objective way be explained to a computer? The answer to both questions is “yes.” The decompositions that make sense are called *lossless*. Before tackling this notion, we need to be more precise about what we mean by a decomposition in the first place.

A **decomposition of a schema**,  $R = (\bar{R}; F)$ , where  $\bar{R}$  is a set of attributes of the schema and  $F$  is its set of functional dependencies, is a collection of schemas

$$R_1 = (\bar{R}_1; F_1), R_2 = (\bar{R}_2; F_2), \dots, R_n = (\bar{R}_n; F_n)$$

<sup>3</sup> For example, a particular integrity constraint in the original table might be checkable in the decomposed tables only by taking the join of these tables—which results in significant overhead at run time.

such that the following conditions hold:

1.  $R_i \neq R_j$ , if  $i \neq j$
2.  $\bar{R} = \cup_{i=1}^n \bar{R}_i$
3.  $\mathcal{F}$  entails  $\mathcal{F}_i$  for every  $i = 1, \dots, n$ .

The first part of the definition is clear: a decomposition should not introduce new attributes, and it should not drop attributes found in the original schema. The second part of the definition says that a decomposition should not introduce new functional dependencies (but may drop some). We discuss the second requirement in more detail later.

The decomposition of a schema naturally leads to decomposition of relations over it. A **decomposition of a relation**,  $r$ , defined over schema  $R$ , relative to a schema decomposition  $\bar{R} = (\bar{R}_1; \mathcal{F}_1), \dots, \bar{R}_n = (\bar{R}_n; \mathcal{F}_n)$  is a set of relations

$$r_1 = \pi_{\bar{R}_1}(r), r_2 = \pi_{\bar{R}_2}(r), \dots, r_n = \pi_{\bar{R}_n}(r)$$

where  $\pi$  is the projection operator. It can be shown (see Exercise 6.12) that, if  $r$  is a valid instance of  $R$ , then each  $r_i$  satisfies all FDs in  $\mathcal{F}_i$  and thus each  $r_i$  is a valid relation instance over the schema  $R_i$ . The purpose of a decomposition is to replace the original relation,  $r$ , with a set of relations  $r_1, \dots, r_n$  over the schemas that constitute the decomposed schema.

In view of the above definitions, it is important to realize that *schema decomposition* and *relation instance decomposition* are two different (but related) notions—the former is a set of relation schemas, while the latter is a set of relation instances.

**Example 6.6.1 (Decomposition).** Applying the above definitions to our running example, we see that splitting PERSON into PERSON1 and HOBBY (see (6.1) and Figure 6.1 on page 195 yields a decomposition. Splitting PERSON as shown in (6.2) and Figure 6.2 is also a decomposition in the above sense. It clearly satisfies the first requirement for being a decomposition. It also satisfies the second since only trivial FDs hold in (6.2), and these are entailed by every set of dependencies. ■

This last example shows that the above definition of a decomposition does not capture all of the desirable properties of a decomposition because, as you may recall from Section 6.2, the decomposition in (6.2) makes no sense. In the following section, we introduce additional desirable properties of decompositions.

### 6.6.1 Lossless and Lossy Decompositions

Consider a relation,  $r$ , and its decomposition,  $r_1, \dots, r_n$ , as defined above. Since after the decomposition the database no longer stores the relation  $r$  and instead maintains its projections  $r_1, \dots, r_n$ , the database must be able to reconstruct the original relation  $r$  from these projections. Not being able to reconstruct  $r$  means that the decomposition does not represent the same information as does the original

database (imagine a bank losing the information about who owns which account or, worse, associating those accounts with the wrong owners!).

In principle, one can use any computational method that guarantees reconstruction of  $r$  from its projections. However, the natural and, in most cases, practical method is the natural join. We thus assume that  $r$  is reconstructible if and only if

$$r = r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

Reconstructibility must be a property of schema decomposition and not of a particular instance over this schema. At the database design stage, the designer manipulates schemas, not relations, and any transformation performed on a schema must guarantee that reconstructibility holds for all of its valid relation instances.

This discussion leads to the following notion. A decomposition of schema  $R = (\bar{R}; F)$  into a collection of schemas

$$R_1 = (\bar{R}_1; F_1), R_2 = (\bar{R}_2; F_2), \dots, R_n = (\bar{R}_n; F_n)$$

is **lossless** if, for every valid instance  $r$  of schema  $R$ ,

$$r = r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

where

$$r_1 = \pi_{\bar{R}_1}(r), r_2 = \pi_{\bar{R}_2}(r), \dots, r_n = \pi_{\bar{R}_n}(r)$$

A decomposition is **lossy** otherwise.

In plain terms, a lossless schema decomposition is one that guarantees that any valid instance of the original schema can be reconstructed from its projections on the individual schemas of the decomposition. Note that

$$r \subseteq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

holds for *any* decomposition whatsoever (Exercise 6.13), so losslessness really just asserts the opposite inclusion:

$$r \supseteq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

The fact that

$$r \subseteq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

holds, no matter what, may seem confusing at first. If we can get more tuples by joining the projections of  $r$ , why is such a decomposition called lossy? After all, we gained more tuples—not less! To clarify this issue, observe that what we might lose here are not tuples but rather information about *which tuples are the right ones*. Consider, for instance, the decomposition (6.2) of schema PERSON on page 196. Figure 6.2 presents the corresponding decomposition of a valid relation instance over PERSON shown in Figure 4.13. However, if we now compute a natural join of the relations in the decomposition (which becomes a Cartesian product since these

relations do not share attributes), we will not be able to tell who lives where and who has what hobbies. The relationship among names and SSNs is also lost. In other words, when reconstructing the original relation, getting more tuples is as bad as getting fewer—we must get *exactly* the set of tuples in the original relation.

Now that we are convinced of the importance of losslessness, we need an algorithm that a computer can use to verify this property since the definition of lossless joins does not provide an effective test but only tells us to try every possible relation. This is neither feasible nor efficient.

A general test of whether a decomposition into  $n$  schemas is lossless exists but is somewhat complex. It can be found in [Beeri et al. 1981]. However, there is a much simpler test that works for binary decompositions, that is, decompositions into a pair of schemas. This test can establish losslessness of a decomposition into more than two schemas provided that this decomposition was obtained by a series of binary decompositions. Since most decompositions are obtained in this way, the simple binary test introduced below is sufficient for most practical purposes.

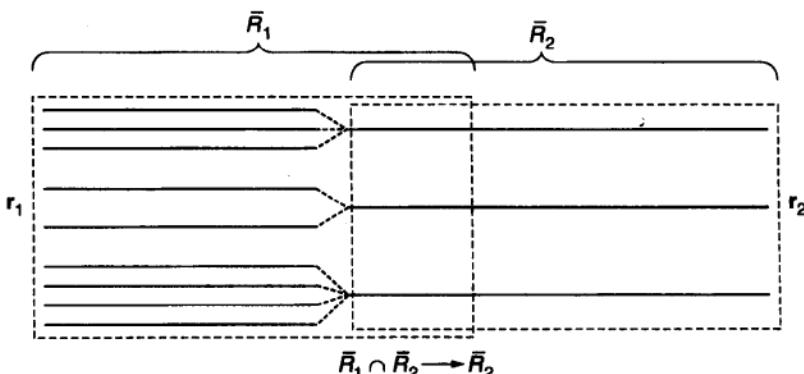
**Testing the losslessness of a binary decomposition.** Let  $R = (\bar{R}; \mathcal{F})$  be a schema and  $R_1 = (\bar{R}_1; \mathcal{F}_1)$ ,  $R_2 = (\bar{R}_2; \mathcal{F}_2)$  be a binary decomposition of  $R$ . This decomposition is lossless if and only if either of the following is true:

- $(\bar{R}_1 \cap \bar{R}_2) \rightarrow \bar{R}_1 \in \mathcal{F}^+$ .
- $(\bar{R}_1 \cap \bar{R}_2) \rightarrow \bar{R}_2 \in \mathcal{F}^+$ .

To see why this is so, suppose that  $(\bar{R}_1 \cap \bar{R}_2) \rightarrow \bar{R}_2 \in \mathcal{F}^+$ . Then  $\bar{R}_1$  is a superkey of  $R$  since, by augmentation with  $\bar{R}_1$ , we can derive  $\bar{R}_1 \rightarrow \bar{R}_1 \cup \bar{R}_2$ , and  $\bar{R} = \bar{R}_1 \cup \bar{R}_2$ . Let  $r$  be a valid relation instance for  $R$ . Since  $\bar{R}_1$  is a superkey of  $\bar{R}$  every tuple in  $r_1 = \pi_{\bar{R}_1}(r)$  extends to exactly one tuple in  $r$ . Thus, as depicted in Figure 6.6, the cardinality (the number of tuples) in  $r_1$  equals the cardinality of  $r$ , and every tuple in  $r_1$  joins with exactly one tuple in  $r_2 = \pi_{\bar{R}_2}(r)$  (if more than one tuple in  $r_2$  joined with a tuple in  $r_1$ , the FD  $(\bar{R}_1 \cap \bar{R}_2) \rightarrow \bar{R}_2$  would not be satisfied in  $r_2$ ). Therefore, the cardinality of  $r_1 \bowtie r_2$  equals the cardinality of  $r_1$ , which in turn equals the cardinality of  $r$ . Since  $r$  must be a subset of  $r_1 \bowtie r_2$ , it follows that  $r = r_1 \bowtie r_2$ . Conversely, if neither of the above FDs holds, it is easy to construct a relation  $r$  such that  $r \subset r_1 \bowtie r_2$ . Details of this construction are left to Exercise 6.14.

The above test can now be used to substantiate our intuition that the decomposition (6.1) of PERSON into PERSON1 and HOBBY is a good one. The intersection of the attributes of HOBBY and PERSON1 is {SSN}, and SSN is a key of PERSON1. Thus, this decomposition is lossless.

Note that this test can be used to verify losslessness of certain decompositions into three or more schemas. Indeed, it is easy to verify that if we take a lossless decomposition and losslessly decompose one of its member schemas into a pair of subschemas, then the result is lossless as well (Exercise 6.15). Therefore, any decomposition that can be derived by a sequence of binary lossless decompositions is itself lossless.



**FIGURE 6.6** Tuple structure in a lossless binary decomposition: a row of  $r_1$  combines with exactly one row of  $r_2$ .

A general algorithm for testing losslessness of an arbitrary  $n$ -ary decomposition exists but is outside of the scope of this book. A curious reader is referred to [Maier 1983].

## 6.6.2 Dependency-Preserving Decompositions

Consider the schema HASACCOUNT once again. Recall that it has attributes AccountNumber, ClientId, OfficeId, and its FDs are

---


$$\text{ClientId, OfficeId} \rightarrow \text{AccountNumber}$$

**6.6**

$$\text{AccountNumber} \rightarrow \text{OfficeId}$$

**6.7**

According to the losslessness test, the following decomposition is lossless:

---


$$\text{ACCTOFFICE} = (\text{AccountNumber}, \text{OfficeId}; \{\text{AccountNumber} \rightarrow \text{OfficeId}\})$$

**6.8**

$$\text{ACCTCLIENT} = (\text{AccountNumber}, \text{ClientId}; \{ \})$$


---

because AccountNumber (the intersection of the two attribute sets) is a key of the first schema, ACCTOFFICE.

Even though decomposition (6.8) is lossless, something seems to have fallen through the cracks. ACCTOFFICE hosts the FD (6.7), but ACCTCLIENT's associated set of FDs is empty. This leaves the FD (6.6), which exists in the original schema, homeless. Neither of the two schemas in the decomposition has all of the attributes needed to house this FD; furthermore, the FD cannot be derived from the FDs that belong to the schemas ACCTOFFICE and ACCTCLIENT.

In practical terms, this means that even though decomposing relations over the schema HASACCOUNT into relations over ACCTOFFICE and ACCTCLIENT does not lead to information loss, it might incur a cost for maintaining the integrity constraint that corresponds to the lost FD. Unlike the FD (6.7), which can be checked locally (in the relation ACCTOFFICE), verification of the FD (6.6) requires computing the join of ACCTOFFICE and ACCTCLIENT before checking of the FD can begin. In such cases, we say that the decomposition does not preserve the dependencies in the original schema. We now define what this means precisely.

Consider a schema,  $R = (\bar{R}; \mathcal{F})$ , and suppose that

$$R_1 = (\bar{R}_1; \mathcal{F}_1), R_2 = (\bar{R}_2; \mathcal{F}_2), \dots, R_n = (\bar{R}_n; \mathcal{F}_n)$$

is a decomposition.  $\mathcal{F}$  entails each  $\mathcal{F}_i$  by definition, so  $\mathcal{F}$  entails  $\bigcup_{i=1}^n \mathcal{F}_i$ . However, this definition does not require that the two sets of dependencies be equivalent—that is, that  $\bigcup_{i=1}^n \mathcal{F}_i$  must also entail  $\mathcal{F}$ . This reverse entailment is what is missing in the above example. The FD set of HASACCOUNT, which consists of dependencies (6.6) and (6.7), is not entailed by the union of the dependencies in decomposition (6.8), which consists of only a single dependency  $\text{AccountNumber} \rightarrow \text{OfficeId}$ . Because of this, (6.8) is not a dependency-preserving decomposition.

Formally,

$$R_1 = (\bar{R}_1; \mathcal{F}_1), R_2 = (\bar{R}_2; \mathcal{F}_2), \dots, R_n = (\bar{R}_n; \mathcal{F}_n)$$

is said to be a **dependency-preserving decomposition** of  $R = (\bar{R}; \mathcal{F})$  if and only if it is a decomposition of  $R$  and the sets of FDs  $\mathcal{F}$  and  $\bigcup_{i=1}^n \mathcal{F}_i$  are equivalent.

**Example 6.6.2 (Dependency-Preserving Decomposition).** Consider a schema with the attributes SSN, Emp1Id, and DeptId, and the FDs  $\mathcal{F} = \{f_1 : \text{SSN} \rightarrow \text{Emp1Id}, f_2 : \text{Emp1Id} \rightarrow \text{SSN}, f_3 : \text{SSN} \rightarrow \text{DeptId}\}$ . Its decomposition into  $R_1 = (\text{SSN Emp1Id}; \mathcal{F}_1 = \{f_1, f_2\})$  and  $R_2 = (\text{Emp1Id DeptId}; \mathcal{F}_2 = \{f_4 : \text{Emp1Id} \rightarrow \text{DeptId}\})$  is dependency preserving.

Note that the FD  $f_3 \in \mathcal{F}$  is not in  $\mathcal{F}_1 \cup \mathcal{F}_2 = \{f_1, f_2, f_4\}$ , but nonetheless  $\mathcal{F}^+ = (\mathcal{F}_1 \cup \mathcal{F}_2)^+$  because  $f_3$  can be derived from  $f_1$  and  $f_4$ ; and  $f_4$  from  $f_2$  and  $f_3$ . ■

The above example shows that even if some FD,  $f \in \mathcal{F}$ , is not found in any of the  $\mathcal{F}_i$ s this does *not* mean that the decomposition is not dependency preserving, since  $f$  might be entailed by  $\bigcup_{i=1}^n \mathcal{F}_i$ . In this case, maintaining  $f$  as a functional dependency requires no extra effort. If the FDs in  $\bigcup_{i=1}^n \mathcal{F}_i$  are maintained,  $f$  will be also. It is only when  $f$  is not entailed by  $\bigcup_{i=1}^n \mathcal{F}_i$  that the decomposition is not dependency preserving and so maintenance of  $f$  requires a join.

**Example 6.6.3 (Nonpreserving Decomposition).** The decomposition (6.8) of HASACCOUNT is not dependency preserving, and this is precisely what is wrong. The dependencies that exist in the original schema but are lost in the decomposition become interrelational constraints that cannot be maintained locally. Each time a relation in the decomposition is changed, satisfaction of the interrelational con-

AccountNumber	ClientId	OfficeId
B123	11111111	SB01
A908	123456789	MN08

HASACCOUNT

AccountNumber	OfficeId
B123	SB01
A908	MN08

ACCTOFFICE

AccountNumber	ClientId
B123	11111111
A908	123456789

ACCTCLIENT

**FIGURE 6.7** Decomposition of the HASACCOUNT relation.

straints can be checked only after the reconstruction of the original relation. To illustrate, consider the decomposition of HASACCOUNT in Figure 6.7.

If we now add the tuple  $(B567, SB01)$  to the relation ACCTOFFICE and the tuple  $(B567, 11111111)$  to ACCTCLIENT, the two relations will still satisfy their local FDs (in fact, we see from (6.8) that only ACCTOFFICE has a dependency to satisfy). In contrast, the interrelational FD (6.6) is not satisfied after these updates, but this is not immediately apparent. To verify this, we must join the two relations, as depicted in Figure 6.8. We now see that constraint (6.6) is violated by the first two tuples in the updated HASACCOUNT relation. ■

AccountNumber	ClientId	OfficeId
B123	11111111	SB01
B567	11111111	SB01
A908	123456789	MN08

HASACCOUNT

AccountNumber	OfficeId
B123	SB01
B567	SB01
A908	MN08

ACCTOFFICE

AccountNumber	ClientId
B123	11111111
B567	11111111
A908	123456789

ACCTCLIENT

**FIGURE 6.8** HASACCOUNT and its decomposition after the insertion of several rows.

The next question to ask is how hard it is to check whether a decomposition is dependency preserving. If we already have a decomposition complete with sets of functional dependencies attached to the local schemas, dependency preservation can be checked in polynomial time. We simply need to check that each FD in the original set is entailed by the union of FDs in the local schemas. For each such test, we can use the quadratic attribute closure algorithm discussed in Section 6.4.

In practice, the situation is more involved. Typically, we (and computer algorithms) must first decide how to split the attribute set in order to form the decomposition, and only then attach FDs to those attribute sets. We can attach the FD  $\bar{X} \rightarrow \bar{Y}$  to an attribute set  $\bar{S}$  if  $\bar{X} \cup \bar{Y} \subseteq \bar{S}$ . We state this more formally as follows.

Consider a schema,  $R = (\bar{R}; \mathcal{F})$ , a relation,  $r$ , over  $R$ , and a set of attributes,  $\bar{S}$ , such that  $\bar{S} \subseteq \bar{R}$ . If  $\bar{S}$  is one of the schemas in a decomposition of  $R$ , the only FDs that are guaranteed to hold over  $\pi_{\bar{S}}(r)$  are  $\bar{X} \rightarrow \bar{Y} \in \mathcal{F}^+$  such that  $\bar{X} \bar{Y} \subseteq \bar{S}$ . This leads us to the following notion:

$$\pi_{\bar{S}}(\mathcal{F}) = \{\bar{X} \rightarrow \bar{Y} \mid \bar{X} \rightarrow \bar{Y} \in \mathcal{F}^+ \text{ and } \bar{X} \cup \bar{Y} \subseteq \bar{S}\}$$

which is called the **projection of the set  $\mathcal{F}$  of FDs onto the set of attributes  $\bar{S}$** .

The notion of projection for FDs opens a way to construct decompositions knowing only how to split the attribute set of the original schema. If  $R = (\bar{R}; \mathcal{F})$  is a schema and  $\bar{R}_1, \dots, \bar{R}_n$  are subsets of attributes such that  $\bar{R} = \bigcup_{i=1}^n \bar{R}_i$ , then the collection of schemas  $(\bar{R}_1; \pi_{\bar{R}_1}(\mathcal{F})), \dots, (\bar{R}_n; \pi_{\bar{R}_n}(\mathcal{F}))$  is a decomposition.

From now on we will consider only decompositions of this kind, that is, those where  $\mathcal{F}_i$  is equivalent to  $\pi_{\bar{R}_i}(\mathcal{F})$ . Since, given an attribute set in a decomposition, it is now possible to uniquely determine the corresponding set of FDs by taking a projection, it is customary to omit the FDs when specifying schema decompositions.

Constructing decompositions thus requires computing projections of FDs. This computation involves calculating the closure of  $\mathcal{F}$ ,<sup>4</sup> which, in the worst case, can take time exponential in the size of  $\mathcal{F}$ . If the cost of this computation is factored into the cost of checking for dependency preservation, this checking is exponential as well. In this regard, it is interesting that, in order to test a decomposition for losslessness, one does not need to compute the FDs that belong to the local schemas. The test presented earlier used the original set  $\mathcal{F}$  of FDs and is polynomial in the size of  $\mathcal{F}$ .

To summarize, we considered two important properties of schema decomposition: losslessness and dependency preservation. We also saw an example of a relation (HASACCOUNT) that has a lossless but not dependency-preserving decomposition into BCNF (and which, as we shall see, does not have a BCNF decomposition that has both properties). Which of these two properties is more important? The answer is that losslessness is mandatory while dependency preservation, though very desirable, is optional. The reason is that lossy decompositions lose information contained in the original database, and this is not acceptable. In contrast, decompositions that

<sup>4</sup> There is a way to avoid computing the entire closure  $\mathcal{F}^+$ , but the worst-case complexity is the same.

do not preserve FDs only lead to computational overhead when the database is changed and interrelational constraints need to be checked.

## 6.7 An Algorithm for BCNF Decomposition

We are now ready to present our first decomposition algorithm. Let  $R = (\bar{R}; \mathcal{F})$  be a relational schema that is not in BCNF. The algorithm in Figure 6.9 constructs a new decomposition by repeatedly splitting  $R$  into smaller subschemas so that at each step the new database schema has strictly fewer FDs that violate BCNF than does the schema in the previous iteration (see Exercise 6.16). Thus, the algorithm always terminates and all schemas in the result are in BCNF.

To see how the BCNF decomposition algorithm works, consider the HASACCOUNT example once again. This schema is not in BCNF, because of the FD  $\text{AccountNumber} \rightarrow \text{OfficeId}$  whose left-hand side is not a superkey. Therefore, we can use this FD to split HASACCOUNT in the while loop of Figure 6.9. The result is, not surprisingly, the decomposition we saw in (6.8).

The next example is more involved and also much more abstract.

**Example 6.7.1 (BCNF Decomposition).** Consider a relation schema,  $R = (\bar{R}; \mathcal{F})$ , where  $\bar{R} = ABCDEFGH$  (recall that  $A, B$ , etc., denote attribute names), and let the set  $\mathcal{F}$  of FDs be

$$\begin{aligned} ABH &\rightarrow C \\ A &\rightarrow DE \\ BGH &\rightarrow F \\ F &\rightarrow ADH \\ BH &\rightarrow GE \end{aligned}$$

To apply the BCNF decomposition algorithm, we first need to identify the FDs that violate BCNF—those whose left-hand side is not a superkey. We can see that the first FD is *not* one of these because the attribute closure  $(ABH)^+$  (computed with the algorithm in Figure 6.3) contains all schema attributes and so  $ABH$  is a superkey.

**FIGURE 6.9** Lossless decomposition into BCNF.

**Input:**  $R = (\bar{R}; \mathcal{F})$

**Output:** A lossless decomposition of  $R$  where each local schema is in BCNF.

*Decomposition := {R}*

**while** there is a schema  $S = (\bar{S}; \mathcal{F}')$  in *Decomposition* that is not in BCNF **do**

/\* Let  $\bar{X} \rightarrow \bar{Y}$  be an FD in  $\mathcal{F}'^+$  such that  $\bar{X}\bar{Y} \subseteq \bar{S}$  and

it violates BCNF in  $S$ . Decompose using this FD \*/

Replace  $S$  in *Decomposition* with schemas  $S_1 = (\bar{X}\bar{Y}; \mathcal{F}'_1)$  and

$S_2 = ((\bar{S} - \bar{Y}) \cup \bar{X}; \mathcal{F}'_2)$ , where  $\mathcal{F}'_1 = \pi_{\bar{X}\bar{Y}}(\mathcal{F}')$  and  $\mathcal{F}'_2 = \pi_{(\bar{S}-\bar{Y})\cup\bar{X}}(\mathcal{F}')$

**end**

**return** *Decomposition*

However, the second FD,  $A \rightarrow DE$ , does violate BCNF. The attribute closure of  $A$  is  $ADE$ , and so  $A$  is not a superkey. We can thus split  $R$  using this FD:

$$R_1 = (ADE; \{A \rightarrow DE\})$$

$$R_2 = (ABCFGH; \{ABH \rightarrow C, BGH \rightarrow F, F \rightarrow AH, BH \rightarrow G\})$$

Notice that we separated  $F \rightarrow ADH$  into  $\{F \rightarrow AH, F \rightarrow D\}$  and  $BH \rightarrow GE$  into  $\{BH \rightarrow G, BH \rightarrow E\}$  and that some FDs fell by the wayside:  $F \rightarrow D$  and  $BH \rightarrow E$  no longer have a home since none of the new schemas contains all the attributes used by these FDs. However, things are still looking bright since the FD  $F \rightarrow D$  can be derived from other FDs embedded in the new schemas  $R_1$  and  $R_2$ :  $F \rightarrow AH$  and  $A \rightarrow DE$ . Similarly,  $BH \rightarrow E$  can still be derived because the attribute closure of  $BH$  with respect to the FDs embedded in  $R_1$  and  $R_2$ , contains  $E$ . (Verify this claim using the algorithm in Figure 6.3!) The above decomposition is therefore dependency preserving.

It is easy to see that  $R_1$  is in BCNF. Although  $A \rightarrow DE$  violates BCNF in  $R$ , it does not violate BCNF in  $R_1$  since, by construction,  $A$  is a key of  $R_1$ . Note that it will always be true that the offending FD,  $f$ , in  $R$  that is used as the basis for the decomposition is converted to a nonoffending FD in  $R_1$ . In general, however, you cannot assume that  $R_1$  will always be in BCNF since there could be other FDs in  $R_1$  that violate BCNF.

What about  $R_2$ ? The FDs  $ABH \rightarrow C$  and  $BGH \rightarrow F$  did not violate BCNF in  $R$  since both  $ABH$  and  $BGH$  are superkeys. As a result, they do not violate BCNF in  $R_2$  (which has only a subset of the attributes of  $R$ ). The FD that clearly violates BCNF here is  $F \rightarrow AH$ , so the algorithm might pick it up and split  $R_2$  accordingly.

$$R_{21} = (FAH; \{F \rightarrow AH\})$$

$$R_{22} = (FBCG; \{FB \rightarrow CG\})$$

(Note that the FD  $FB \rightarrow CG$  is not in  $\mathcal{F}$  but is derivable from it.)

Now both schemas,  $R_{21}$  and  $R_{22}$ , are in BCNF. However, the price is that the FDs  $ABH \rightarrow C$ ,  $BGH \rightarrow F$ , and  $BH \rightarrow G$  that were present in  $R_2$  are now homeless. Furthermore, none of these FDs can be derived using the FDs that are still embedded in  $R_1$ ,  $R_{21}$ , and  $R_{22}$ . For instance, computing  $(ABH)^+$  with respect to this set of FDs yields  $ABHDE$ , which does not contain  $C$ , so  $ABH \rightarrow C$  is not derivable. Thus, we obtain a nondependency-preserving decomposition of  $R$  into three BCNF schemas:  $R_1$ ,  $R_{21}$ , and  $R_{22}$ .

This decomposition is by no means unique. For instance, if our algorithm had picked up  $F \rightarrow ADH$  at the very first iteration, the first decomposition would have been

$$R'_1 = (FADH; \{F \rightarrow ADH, A \rightarrow D\})$$

$$R'_2 = (FBCEG; \{F \rightarrow E, FB \rightarrow CG\})$$

Observe that none of these schemas is in BCNF ( $A \rightarrow D$  violates the BCNF requirements in the first schema and  $F \rightarrow E$  in the second) and need to be decomposed further. This is not the end of the differences, however: some FDs that were present in the decomposition into  $R_1$ ,  $R_{21}$ ,  $R_{22}$  are no longer embedded in the new decomposition (e.g.,  $A \rightarrow E$ ). ■

**Properties of the BCNF decomposition algorithm.** First and foremost, the BCNF decomposition algorithm in Figure 6.9 always yields a lossless decomposition. To see this, consider the two schemas involving attribute sets  $\bar{X}\bar{Y}$  and  $(\bar{S} - \bar{Y}) \cup \bar{X}$  that replace the schema  $S = (\bar{S}, \mathcal{F}')$  in the algorithm. Notice that  $\bar{X}\bar{Y} \cap ((\bar{S} - \bar{Y}) \cup \bar{X}) = \bar{X}$  and thus  $\bar{X}\bar{Y} \cap ((\bar{S} - \bar{Y}) \cup \bar{X}) \rightarrow \bar{X}\bar{Y}$  since  $\bar{X} \rightarrow \bar{Y} \in \mathcal{F}'$ . Therefore, according to the losslessness test for binary decompositions on page 214,  $\{\bar{X}\bar{Y}, (\bar{S} - \bar{Y}) \cup \bar{X}\}$  is a lossless decomposition of  $S$ . This means that at every step in our algorithm we replace one schema by its lossless decomposition. Thus, by Exercise 6.15, the final decomposition produced by this algorithm is also lossless.

Are the decompositions produced by the BCNF algorithm always dependency preserving? We have seen that this is not the case. Decomposition (6.8) of HASACCOUNT is not dependency preserving. Moreover, it is easy to see that no BCNF decomposition of HASACCOUNT (not only those produced by this particular algorithm) is both lossless and dependency preserving. Indeed, there are just three decompositions to try, and we can simply check them all.

The BCNF decomposition algorithm is nondeterministic.

Finally, Example 6.7.1 shows that the BCNF decomposition algorithm is nondeterministic. The final result depends on the order in which FDs are selected in the **while** loop. The decomposition chosen by the database designer can be a matter of taste, or it can be based on objective criteria. For instance, some decompositions might be dependency preserving, others not; some might lead to fewer FDs left out as interrelational constraints (e.g., the decomposition  $R_1, R_{21}, R_{22}$  in Example 6.7.1 is better in this sense than the decomposition  $R'_1, R'_2$ ). Some attribute sets might be more likely to be queried together so they better not be separated in the decomposition. The next section describes one common approach that can help in choosing one BCNF decomposition over another.

## 6.8 Synthesis of 3NF Schemas

We have seen that some schemas (such as HASACCOUNT, in Figure 6.8) cannot be decomposed into BCNF so that the result is also dependency preserving. However, if we agree to settle for 3NF instead of BCNF, dependency-preserving decompositions are always possible (but recall that 3NF schemas might contain redundancies—see page 211).

Before we present a 3NF decomposition algorithm, we need to introduce the concept of *minimal cover*, which is really very simple. We know that sets of FDs might look completely different but nonetheless be logically equivalent. Figure 6.4 presented one fairly straightforward way of testing equivalence. Since there might be many sets of FDs equivalent to any given set, we question whether there is a set of FDs that can be viewed as canonical. It turns out that defining a unique canonical set is not an easy task, but the notion of minimal cover comes close.

### 6.8.1 Minimal Cover

Let  $\mathcal{F}$  be a set of FDs. A **minimal cover** of  $\mathcal{F}$  is a set of FDs,  $\mathcal{G}$ , that has the following properties:

1.  $\mathcal{G}$  is equivalent to  $\mathcal{F}$  (but, possibly, different from  $\mathcal{F}$ ).
2. All FDs in  $\mathcal{G}$  have the form  $\bar{X} \rightarrow A$ , where  $A$  is a single attribute.
3. It is not possible to make  $\mathcal{G}$  “smaller” (and still satisfy the first two properties) by either of the following:
  - (a) Deleting an FD
  - (b) Deleting an attribute from an FD

An FD,  $f$ , that can be deleted from a set,  $\mathcal{F}$ , while preserving the equivalence (i.e., when  $\mathcal{F} - f$  is equivalent to  $\mathcal{F}$ ) is said to be a **redundant FD**. An attribute,  $A$ , in  $f$  that can be deleted while preserving the equivalence (i.e., if  $\mathcal{F}$  and  $\mathcal{F} - \{f\} \cup \{f'\}$  are equivalent, where  $f'$  is  $f$  with  $A$  deleted) is said to be a **redundant attribute**. Thus, a minimal cover has neither redundant FDs nor redundant attributes.

Clearly, because of Armstrong's rule of decomposition for functional dependencies, it is easy to convert  $\mathcal{F}$  into an equivalent set of FDs where the right-hand sides are singleton attributes. However, property 3 is more subtle. Before presenting an algorithm for computing minimal covers, we illustrate it with a concrete example.

**Example 6.8.1 (Minimal Cover).** Consider the attribute set  $ABCDEFGH$  and the following set,  $\mathcal{F}$ , of FDs:

$$\begin{array}{ll} ABH \rightarrow C & F \rightarrow AD \\ A \rightarrow D & E \rightarrow F \\ C \rightarrow E & BH \rightarrow E \\ BGH \rightarrow F & \end{array}$$

Since not all right-hand sides are single attributes, we can use the decomposition rule to obtain an FD set that satisfies the first two properties of minimal covers.

$$\begin{array}{ll} ABH \rightarrow C & F \rightarrow A \\ A \rightarrow D & F \rightarrow D \\ C \rightarrow E & E \rightarrow F \\ BGH \rightarrow F & BH \rightarrow E \end{array} \quad \text{6.9}$$

We can see that  $BGH \rightarrow F$  is entailed by  $BH \rightarrow E$  and  $E \rightarrow F$ , and that  $F \rightarrow D$  is entailed by  $F \rightarrow A$  and  $A \rightarrow D$ . Thus, we are left with

$$\begin{array}{ll} ABH \rightarrow C & F \rightarrow A \\ A \rightarrow D & E \rightarrow F \\ C \rightarrow E & BH \rightarrow E \end{array} \quad \text{6.10}$$

It is easy to check by computing attribute closures that none of these FDs is redundant; that is, one cannot simply throw out an FD from this set without sacrificing equivalence to the original set  $\mathcal{F}$ . However, is the resulting set a minimal cover of  $\mathcal{F}$ ? The answer turns out to be *no* because it is possible to delete the attribute  $A$  from the first FD, since  $BH \rightarrow C$  is entailed by the set (6.10) (verify this by computing the attribute closure of  $BH$ ) and  $ABH \rightarrow C$  is obviously entailed by  $BH \rightarrow C$ . Thus, we get

$$\begin{array}{ll} BH \rightarrow C & F \rightarrow A \\ A \rightarrow D & E \rightarrow F \\ C \rightarrow E & BH \rightarrow E \end{array} \quad \text{6.11}$$

Interestingly, the latter set of FDs is still not minimal because the FD  $BH \rightarrow E$  is redundant. Removing this FD yields a minimal cover at last. ■

**Nonuniqueness of minimal covers.** Observe that the outcome of steps 2 and 3 in the algorithm in Figure 6.10 may depend on the particular order in which we test the candidates for removal (both attributes and FDs). This suggests that a set of FDs can have several minimal covers. For instance,  $\{A \rightarrow B, B \rightarrow C, C \rightarrow A, A \rightarrow C, C \rightarrow B, B \rightarrow A\}$  has two minimal covers:  $\{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$  and  $\{A \rightarrow C, C \rightarrow B, B \rightarrow A\}$ .

**Noninterchangeability of steps 2 and 3.** The algorithm for computing minimal covers is presented in Figure 6.10. Step 1 is performed by a simple splitting of the FDs according to their right-hand sides. For instance,  $\bar{X} \rightarrow AB$  turns into  $\bar{X} \rightarrow A$  and  $\bar{X} \rightarrow B$ .

Step 2 is performed by checking every left-hand attribute in  $\mathcal{G}$  for redundancy. That is, for every FD  $\bar{X} \rightarrow A \in \mathcal{G}$  and every attribute  $B \in \bar{X}$ , we have to check if  $(\bar{X} - B) \rightarrow A$  is entailed by  $\mathcal{G}$ —very tedious work if done manually. In the above example, we performed this step when we checked that  $BH \rightarrow C$  is entailed by the FD set (6.10), which allowed us to get rid of the redundant attribute  $A$  in  $ABH \rightarrow C$ . Step 3 is accomplished by another tedious algorithm: for every  $g \in \mathcal{G}$ , check that the FD  $g$  is entailed by  $\mathcal{G} - \{g\}$ .

FIGURE 6.10 Computation of a minimal cover.

**Input:** a set of FDs  $\mathcal{F}$

**Output:**  $\mathcal{G}$ , a minimal cover of  $\mathcal{F}$

**Step 1:**  $\mathcal{G} := \mathcal{F}$ , where all FDs are converted to use singleton attributes on the right-hand side.

**Step 2:** Remove all redundant attributes from the left-hand sides of FDs in  $\mathcal{G}$ .

**Step 3:** Remove all redundant FDs from  $\mathcal{G}$ .

---

**return**  $\mathcal{G}$

An important observation about the algorithm in Figure 6.10 is that steps 2 and 3 *cannot* be done in a different order. Performing step 3 before 2 will not always return a minimal cover. In fact, we have already seen this phenomenon in Example 6.8.1. We obtained set (6.10) by removing redundant FDs from (6.9); then we obtained set (6.11) by deleting redundant attributes. Nevertheless, the result still had a redundant FD,  $BH \rightarrow E$ . On the other hand, if we first remove the redundant attributes from (6.9), we get

$$\begin{array}{ll} BH \rightarrow C & F \rightarrow A \\ A \rightarrow D & F \rightarrow D \\ C \rightarrow E & E \rightarrow F \\ BH \rightarrow F & BH \rightarrow E \end{array}$$

Then removing the redundant FDs  $BH \rightarrow F$ ,  $F \rightarrow D$ , and  $BH \rightarrow E$  yields the following minimal cover:

$$\begin{array}{ll} BH \rightarrow C & F \rightarrow A \\ A \rightarrow D & E \rightarrow F \\ C \rightarrow E & \end{array} \quad \text{6.12}$$

## 6.8.2 3NF Decomposition through Schema Synthesis

The algorithm for constructing dependency-preserving 3NF decompositions works very differently from its BCNF counterpart. Instead of starting with one big schema and successively splitting it, the 3NF algorithm starts with individual attributes and groups them into schemas. For this reason, it is called **3NF synthesis**. Given a schema,  $R = (\bar{R}; \mathcal{F})$ , where  $\bar{R}$  is a superset of attributes and  $\mathcal{F}$  is a set of FDs, the algorithm carries out four steps:

1. Find a minimal cover,  $\mathcal{G}$ , for  $\mathcal{F}$ .
2. Partition  $\mathcal{G}$  into FD sets  $\mathcal{G}_1, \dots, \mathcal{G}_n$ , such that each  $\mathcal{G}_i$  consists of all FDs in  $\mathcal{G}$  that share the same left-hand side. (It is not necessary to assume that different  $\mathcal{G}_i$ s have different left-hand sides, but it is usually a good idea to merge sets whose left-hand sides are the same.)
3. For each  $\mathcal{G}_i$ , form a relation schema,  $R_i = (\bar{R}_i; \mathcal{G}_i)$ , where  $\bar{R}_i$  is the set of all attributes mentioned in  $\mathcal{G}_i$ .
4. If one of the  $\bar{R}_i$ s, is a superkey of  $R$  (i.e.,  $(\bar{R}_i)^+ = \bar{R}$ ), we are done— $R_1, \dots, R_n$  is the desired decomposition. If no  $\bar{R}_i$  is a superkey of  $R$ , let  $\bar{R}_0$  be some key of  $R$ , and let  $R_0 = (\bar{R}_0; \{\})$  be a new schema. Then  $R_0, R_1, \dots, R_n$  is the desired decomposition.

Note that the collection of schemas obtained after step 3 might not even be a decomposition because some attributes of  $R$  might be missing (see Example 6.8.2).

below). However, any such missing attributes will be recaptured in step 4, because these attributes must be part of the key of  $R$  (Exercise 6.28).

**Is it dependency preserving?** This is easy to see: By construction, every FD in  $\mathcal{G}$  has a home, so  $\mathcal{G} = \cup \mathcal{G}_i$ . By definition of minimal covers,  $\mathcal{G}^+ = \mathcal{F}^+$ , so  $\mathcal{F}$  is preserved.

**Checking for BCNF—pitfalls.** It might seem that each  $R_i$  is in BCNF, because every FD in  $\mathcal{G}_i$  is superkey-based. However, do not fall for this argument so easily. The FDs in  $\mathcal{G}_i$  might not be the only ones that hold in  $R_i$ , because  $\mathcal{G}^+$  might have other FDs whose attribute set is entirely contained within  $\bar{R}_i$ . To see this, consider a slight modification of our tried-and-true schema HASACCOUNT. Here  $\bar{R}$  consists of the attributes AccountNumber, ClientId, OfficeId, DateOpened, and  $\mathcal{F}$  consists of the FDs  $\text{ClientId}, \text{OfficeId} \rightarrow \text{AccountNumber}$  and  $\text{AccountNumber} \rightarrow \text{OfficeId}, \text{DateOpened}$ . The above algorithm then produces two schemas:

---


$$\begin{aligned} R_1 &= (\{\text{ClientId}, \text{OfficeId}, \text{AccountNumber}\}, \\ &\quad \{\text{ClientId}, \text{OfficeId} \rightarrow \text{AccountNumber}\}) \\ R_2 &= (\{\text{AccountNumber}, \text{OfficeId}, \text{DateOpened}\}, \\ &\quad \{\text{AccountNumber} \rightarrow \text{OfficeId}, \text{DateOpened}\}) \end{aligned}$$


---

A careful examination shows that, even though  $\text{AccountNumber} \rightarrow \text{OfficeId}$  is not explicitly specified for  $R_1$ , it must nonetheless hold over the attributes of that schema because this FD is implicitly imposed by  $R_2$ .

To understand why the FDs specified for  $R_2$  must be taken into account when considering  $R_1$ , observe that the pair of attributes  $\text{AccountNumber}, \text{OfficeId}$  represents the same real-world relationship in both  $R_1$  and  $R_2$ , so it is an inconsistency if the tuples in  $R_2$  obey a constraint over this pair of attributes while the tuples in  $R_1$  do not.

Thus, in general, to check which of the schemas  $R_i$  obtained by the 3NF synthesis are in BCNF, it is not enough to look for violations of BCNF among the FDs in  $\mathcal{G}_i$ . Instead, it is necessary to compute the projections  $\pi_{\bar{R}_i}(\mathcal{G})$  and look for the violations there. As explained earlier, this is rather tiresome to do manually (except for small examples) because computing projections of FDs is exponentially hard.

**Is it really 3NF?** It seems obvious that each  $R_i$  is a 3NF schema, because the only FDs associated with  $R_i$  are those in  $\mathcal{G}_i$  and they all share the same left-hand side (which is thus a superkey of  $R_i$ ). However, the above argument regarding BCNF shows that schemas produced by the synthesis algorithm might have FDs with different left-hand sides and thus conformance to 3NF is not at all obvious. Nevertheless, it can be *proved* that the above algorithm always yields 3NF decompositions (see Exercise 6.17).

**Is it lossless?** The final question is whether the synthesis algorithm yields lossless decompositions of the input schema. The answer is yes, but proving this is more

difficult than proving the 3NF property. Although it might not be obvious, achieving losslessness is in fact the only purpose of step 4 in that algorithm. This is illustrated in the following example.

**Brain Teaser:** Does 3NF synthesis always produce a unique result?

**Example 6.8.2 (3NF Synthesis Where Step 4 Is Essential).** Consider the schema with FDs depicted in (6.9) on page 222. A minimal cover for this set is shown in (6.12). Since no two FDs here share the same left-hand side, we end up with the following schemas:  $(BHC; BH \rightarrow C)$ ,  $(AD; A \rightarrow D)$ ,  $(CE; C \rightarrow E)$ ,  $(FA; F \rightarrow A)$ , and  $(EF; E \rightarrow F)$ . Notice that none of these schemas forms the superkey for the entire set of attributes. For instance, the attribute closure of  $BHC$  does not contain  $G$ . In fact, the attribute  $G$  is not even included in any of the schemas! So, according to our remark about the purpose of step 4, this decomposition is not lossless (in fact, it is not even a decomposition!) To make it lossless, we perform step 4 and add the schema  $(BGH; \{ \})$ . ■

### 6.8.3 BCNF Decomposition through 3NF Synthesis

So, how can 3NF synthesis help design BCNF database schemas? The answer is simple. To decompose a schema into BCNF relations, do *not* use the BCNF algorithm first. Instead, use 3NF synthesis, which is lossless and guaranteed to preserve dependencies. If the resulting schemas are already in BCNF (as in Example 6.8.2), no further action is necessary. If, however, some schema in the result is not in BCNF, use the BCNF algorithm to split it until no violation of BCNF remains. Repeat this step for each non-BCNF schema produced by the 3NF synthesis.

The advantage of this approach is that, if a lossless and dependency-preserving decomposition exists, 3NF synthesis is likely to find it. If some schemas are not in BCNF after the first stage, loss of some FDs is inevitable (Exercise 6.34). But at least we tried hard. Here is a complete example that illustrates the above approach.

**Example 6.8.3 (Combining Schema Synthesis and Decomposition).** Let the attribute set be  $St$  (student),  $C$  (course),  $Sem$  (semester),  $P$  (professor),  $T$  (time), and  $R$  (room) with the following FDs:

$$\begin{array}{l} St \text{ } C \text{ } Sem \rightarrow P \\ P \text{ } Sem \rightarrow C \\ C \text{ } Sem \text{ } T \rightarrow P \end{array}$$

$$\begin{array}{l} P \text{ } Sem \text{ } T \rightarrow C \text{ } R \\ P \text{ } Sem \text{ } C \text{ } T \rightarrow R \\ P \text{ } Sem \text{ } T \rightarrow C \end{array}$$

These functional dependencies apply at a university in which multiple sections of the same course might be taught in the same semester (in which case providing the name of a course and a semester does not uniquely identify a professor) and a professor teaches only one course a semester (in which case providing the name of a professor and a semester uniquely identifies a course) and all the sections of a course are taught at different times.

We begin by finding a minimal cover for the above set. The first step is to split the right-hand sides of the set of FDs into singleton attributes.

$$\text{St C Sem} \rightarrow \text{P}$$

$$\text{P Sem} \rightarrow \text{C}$$

$$\text{C Sem T} \rightarrow \text{P}$$

$$\text{P Sem T} \rightarrow \text{C}$$

$$\text{P Sem T} \rightarrow \text{R}$$

$$\text{P Sem C T} \rightarrow \text{R}$$

$$\text{P Sem T} \rightarrow \text{C}$$

Let  $\mathcal{F}$  denote this set of FDs. The last FD is a duplicate, so we delete it from the set. Next we reduce the left-hand sides by eliminating redundant attributes. For instance, to check the left-hand side  $\text{St C Sem}$ , we must compute several attribute closures— $(\text{St Sem})_{\mathcal{F}}^+ = \{\text{St}, \text{Sem}\}$ ;  $(\text{St C})_{\mathcal{F}}^+ = \{\text{St}, \text{C}\}$ ;  $(\text{C Sem})_{\mathcal{F}}^+ = \{\text{C}, \text{Sem}\}$ —which show that there are no redundant attributes in the first FD. Similarly,  $\text{P Sem}$ ,  $\text{C Sem T}$ , and  $\text{P Sem T}$  cannot be reduced. However, checking  $\text{P Sem C T}$  brings a reward:  $(\text{P Sem T})_{\mathcal{F}}^+ = \text{P Sem T C R}$ , so  $\text{C}$  can be deleted.

The outcome from this stage is the following set of FDs, which we number for convenient reference.

---

FD1.  $\text{St C Sem} \rightarrow \text{P}$

FD2.  $\text{P Sem} \rightarrow \text{C}$

FD3.  $\text{C Sem T} \rightarrow \text{P}$

FD4.  $\text{P Sem T} \rightarrow \text{C}$

FD5.  $\text{P Sem T} \rightarrow \text{R}$

---

The next step is to get rid of the redundant FDs, which are detected with the help of attribute closure, as usual. Since  $(\text{St C Sem})_{\mathcal{F}-\text{FD1}}^+ = \text{St C Sem}$ , FD1 cannot be eliminated. Nor can FDs 2, 3, and 5. However, FD4 is redundant (because of FD2), so it can be eliminated. Thus, the minimal cover is

---

$\text{St C Sem} \rightarrow \text{P}$

$\text{P Sem} \rightarrow \text{C}$

$\text{C Sem T} \rightarrow \text{P}$

$\text{P Sem T} \rightarrow \text{R}$

---

This leads to the following dependency-preserving 3NF decomposition:

---

$(\text{St C Sem P}; \text{St C Sem} \rightarrow \text{P})$

$(\text{P Sem C}; \text{P Sem} \rightarrow \text{C})$

$(\text{C Sem T P}; \text{C Sem T} \rightarrow \text{P})$

$(\text{P Sem T R}; \text{P Sem T} \rightarrow \text{R})$

---

It is easy to verify that none of the above schemas forms a superkey for the original schema; therefore, to make the decomposition lossless we also need to add

a schema whose attribute closure contains all the original attributes. The schema ( $St \text{ } T \text{ } Sem \text{ } P; \{ \}$ ) is one possibility here.

If you trust that the 3NF synthesis algorithm is correct and that we did not make mistakes applying it, no checking for 3NF is necessary. However, a quick look reveals that the first and the third schemas are not in BCNF because of the FD  $P \text{ } Sem \rightarrow C$  embedded in the second schema.

Further decomposition of the first schema with respect to  $P \text{ } Sem \rightarrow C$  yields  $(P \text{ } Sem \text{ } C; P \text{ } Sem \rightarrow C)$  and  $(P \text{ } Sem \text{ } St; \{ \})$ —a lossless decomposition but one in which the FD  $St \text{ } C \text{ } Sem \rightarrow P$  is not preserved.

Decomposition of the third schema with respect to  $P \text{ } Sem \rightarrow C$  yields  $(P \text{ } Sem \text{ } C; P \text{ } Sem \rightarrow C)$  and  $(P \text{ } Sem \text{ } T; \{ \})$ —another lossless decomposition, which, alas, does not preserve  $C \text{ } Sem \text{ } T \rightarrow P$ .

So, the final BCNF decomposition is

---

$(P \text{ } Sem \text{ } C; P \text{ } Sem \rightarrow C)$   
 $(P \text{ } Sem \text{ } St)$   
 $(P \text{ } Sem \text{ } T)$   
 $(P \text{ } Sem \text{ } T \text{ } R; P \text{ } Sem \text{ } T \rightarrow R)$   
 $(St \text{ } T \text{ } Sem \text{ } P)$

---

This decomposition is lossless because we first obtained a lossless 3NF decomposition and then applied the BCNF algorithm, which preserves losslessness. It is not dependency preserving, however, since  $St \text{ } C \text{ } Sem \rightarrow P$  and  $C \text{ } Sem \text{ } T \rightarrow P$  are not represented in the above schemas. ■

## 6.9 The Fourth Normal Form

Not all of the world's problems are due to bad FDs. Consider the following schema:

---

PERSON(SSN, PhoneN, ChildSSN)

---

**6.13**

where we assume that a person can have several phone numbers and several children. Here is one possible relation instance.

SSN	PhoneN	ChildSSN
111-22-3333	516-123-4567	222-33-4444
111-22-3333	516-345-6789	222-33-4444
111-22-3333	516-123-4567	333-44-5555
111-22-3333	516-345-6789	333-44-5555
222-33-4444	212-987-6543	444-55-6666
222-33-4444	212-987-1111	555-66-7777
222-33-4444	212-987-6543	555-66-7777
222-33-4444	212-987-1111	444-55-6666

**6.14**

As there are no nontrivial functional dependencies (we assume that most children in the database have two parents and so the FD  $\text{ChildSSN} \rightarrow \text{SSN}$  does not hold), this schema is in 3NF and even BCNF. Nonetheless, it is clearly not a good design as it exhibits a great deal of redundancy. There is no particular association between phone numbers and children, except through the SSN, so every child item related to a given SSN must occur in one tuple with every PhoneN related to the same SSN. Thus, whenever a phone number is added or deleted, several tuples might need to be added or deleted as well. If a person gives up all phone numbers, the information about her children will be lost (or NULL values will have to be used).

It might seem that a compression technique can help here. For instance, we might decide to store only some tuples as long as there is a way to reconstruct the original information.

SSN	PhoneN	ChildSSN
111-22-3333	516-123-4567	222-33-4444
111-22-3333	516-345-6789	333-44-5555
222-33-4444	212-987-6543	444-55-6666
222-33-4444	212-987-1111	555-66-7777

Still, although this is more efficient, it solves none of the aforesaid anomalies. Also, it imposes an additional burden on the applications, which now must be aware of the compression schema.

In our discussion of BCNF, we concluded that redundancy arises when a particular semantic relationship among attribute values is stored more than once. In Figure 4.13 on page 86, the fact that the person with SSN 111111111 lives at 123 Main Street is an example of that—it is stored two times. In that case, the problem was traced back to the functional dependency that relates SSN and Address and the fact that SSN is not a key (and hence there can be several rows with the same SSN value). The redundant storage of a semantic relationship, however, is not limited to this situation. In the relation  $r$  shown in (6.14), the relationships SSN-PhoneN and SSN-ChildSSN are stored multiple times and there are no FDs involved. The problem arises here because there are several attributes—in this case PhoneN and ChildSSN—that have the property that their sets of values are associated with a single value of another attribute—in this case SSN. A relationship between a particular SSN value and a particular PhoneN value is stored as many times as there are children of the person with that SSN. Note that the relation satisfies the following property:

$$r = \pi_{\text{SSN}, \text{PhoneN}}(r) \bowtie \pi_{\text{SSN}, \text{ChildSSN}}(r)$$

6.15

**Join dependencies.** When (6.15) is required of all legal instances of a schema, this property is known as *join dependency*. A join dependency can arise when characteristics of an enterprise are described by sets of values. With the E-R approach to database design, we saw that such characteristics are represented as set-valued attributes and

that translating them into attributes in the relational model is awkward. In particular, when an entity type or a relationship type has several set-valued attributes, a join dependency results.

Condition (6.15) should look familiar to you. It guarantees that a decomposition of  $r$  into the two tables  $\pi_{SSN, Phone}(r)$  and  $\pi_{SSN, ChildSSN}(r)$  will be lossless. That is certainly true in this case, but it is not our immediate concern. The condition also tells us something about  $r$ : a join dependency indicates that semantic relationships can be stored redundantly in an instance of  $r$ .

Formally, let  $\bar{R}$  be a set of attributes. A **join dependency (JD)** is a constraint of the form

$$\bar{R} = \bar{R}_1 \bowtie \cdots \bowtie \bar{R}_n$$

where  $\bar{R}_1, \dots, \bar{R}_n$  are attribute sets that represent a decomposition of  $\bar{R}$ . Note that here the  $\bar{R}_i$ 's are sets of attributes and  $\bowtie$  is just a symbol—we are not actually joining any relations. However, this expression is related to a join. Recall that earlier (on page 198) we defined the notion of satisfaction of FDs by relational instances. We now define the same notion for JDs: a relation instance,  $r$ , over  $\bar{R}$  satisfies the above join dependency if

$$r = \pi_{\bar{R}_1}(r) \bowtie \cdots \bowtie \pi_{\bar{R}_n}(r)$$

It is easy to see from the definition that the existence of a JD is really another way of saying that there is a lossless decomposition of the schema. Why are we defining the same thing twice? The answer is that previously we used FDs to state that such a decomposition exists, but now we find that a lossless decomposition cannot always be indicated by the presence of certain kinds of FDs. Therefore, we need a new kind of syntactic constraint—join dependencies—which we can attach to a database schema to indicate the presence of a lossless decomposition. In fact, as we will soon see, an FD always implies some sort of a JD, and this is precisely the reason why lossless decompositions are possible with respect to FDs. For instance, consider the schema PERSON2(SSN, Name, ChildSSN) with the FD  $SSN \rightarrow Name$ . It is easy to see from the conditions for losslessness on page 214 that this FD implies the JD

$$(SSN, Name, ChildSSN) = (SSN, Name) \bowtie (SSN, ChildSSN)$$

Moreover, due to the one-to-many relationship between SSN and ChildSSN, a typical relational instance over PERSON2 will have similar redundancy to that of relation (6.14) above.

Let  $R = (\bar{R}; Constraints)$  be a relational schema, where  $\bar{R}$  is a set of attributes and  $Constraints$  is a set of FDs and JDs. As in the case of FDs alone, a relation over the set of attributes  $\bar{R}$  is a **legal instance** of  $R$  if and only if it satisfies all constraints in  $Constraints$ .

**Multivalued dependencies and 4NF.** Of particular interest are binary join dependencies, also known as **multivalued dependencies (MVD)**. These are JDs of the form

$\bar{R} = \bar{R}_1 \bowtie \bar{R}_2$ . The redundancy exhibited by the relation schema PERSON (6.13) was caused by this particular type of join dependency. The *fourth normal form*, introduced in [Fagin 1977], is designed to prevent redundancies of this type.

MVDs constrain instances of a relation in the same way that FDs do, so a description of a relation schema must include both. As a result, we describe a relation schema,  $R$ , as  $(\bar{R}; \mathcal{D})$ , where  $\mathcal{D}$  is now a set of FDs and MVDs. **Entailment** of JDs is defined in the same way as entailment of FDs. Let  $S$  be a set of JDs (and possibly FDs) and  $d$  be a JD (or an FD). Then  $S$  **entails**  $d$  if every relation instance  $r$  that satisfies all dependencies in  $S$  also satisfies  $d$ . In Section 6.10.1, we show how an MVD might be entailed by a set of MVDs. With this in mind, a relation schema,  $R = (\bar{R}; \mathcal{D})$ , is said to be in **fourth normal form** (4NF) if, for every MVD  $\bar{R} = \bar{X} \bowtie \bar{Y}$  that is entailed by  $\mathcal{D}$ , either of the following is true:

- $\bar{X} \subseteq \bar{Y}$  or  $\bar{Y} \subseteq \bar{X}$  (i.e., the MVD is trivial).
- $\bar{X} \cap \bar{Y}$  is a superkey of  $\bar{R}$  (i.e.,  $(\bar{X} \cap \bar{Y}) \rightarrow \bar{R}$  is entailed by  $\mathcal{D}$ ).

It is easy to see that PERSON is *not* a 4NF schema, because the MVD  $\text{PERSON} = (\text{SSN}, \text{PhoneN}) \bowtie (\text{SSN}, \text{ChildSSN})$  holds whereas  $\text{SSN} = \{\text{SSN}, \text{PhoneN}\} \cap \{\text{SSN}, \text{ChildSSN}\}$  is not a superkey. What is the intuition here? If SSN were a superkey, then for each value of SSN there would be at most one value of PhoneN and one value of ChildSSN and hence no redundancy. On the other hand, splitting PERSON into a relation over the attributes (SSN, PhoneN) and another one over the attributes (SSN, ChildSSN) yields a lossless decomposition where every relation is in 4NF and no redundant information is stored.

**4NF and BCNF.** As it turns out, 4NF schemas are also BCNF schemas (i.e., 4NF closes the loopholes that BCNF leaves behind). To see this, suppose that  $R = (\bar{R}; \mathcal{D})$  is a 4NF schema and  $\bar{X} \rightarrow \bar{Y}$  is a nontrivial functional dependency that holds in  $R$ . To show that 4NF schemas are also BCNF schemas we must demonstrate that  $\bar{X}$  is a superkey of  $R$ . For simplicity, assume that  $\bar{X}$  and  $\bar{Y}$  are disjoint. Then  $\bar{R}_1 = \bar{X}\bar{Y}$ ,  $\bar{R}_2 = \bar{R} - \bar{Y}$  is a lossless decomposition of  $R$ . This follows directly from the test for losslessness of binary schema decompositions presented in Section 6.6.1 on page 214. Thus,  $\bar{R} = \bar{R}_1 \bowtie \bar{R}_2$  is a binary join dependency, that is, an MVD that holds in  $R$ . But by the definition of 4NF it follows that either  $\bar{X}\bar{Y} = \bar{R}_1 \subseteq \bar{R}_2 = \bar{R} - \bar{Y}$  (an impossibility) or  $\bar{R} - \bar{Y} = \bar{R}_2 \subseteq \bar{R}_1 = \bar{X}\bar{Y}$  (which implies that  $\bar{R} = \bar{X}\bar{Y}$  and  $\bar{X}$  is a superkey) or that  $\bar{R}_1 \cap \bar{R}_2 (= \bar{X})$  is a superkey. This means that every nontrivial FD in  $R$  satisfies the BCNF requirements.

It can also be shown (but it is harder to do) that if  $R = (\bar{R}; \mathcal{D})$  is such that  $\mathcal{D}$  consists only of FDs, then  $R$  is in 4NF if and only if it is in BCNF (see [Fagin 1977]). In other words, 4NF is an extension of the requirements for BCNF to design environments where MVDs, in addition to FDs, must be specified.

**Brain Teaser:** Can a 4NF schema not be in BCNF if FDs are the only dependencies?

**Designing 4NF schemas.** Because 4NF implies BCNF, we cannot hope to find a general algorithm for constructing a dependency-preserving and lossless decomposition of an arbitrary relation into relations in 4NF. However, as with BCNF, a lossless decomposition into 4NF can always be achieved. Such an algorithm is very similar to that for BCNF. It is an iterative process that starts with the original schema and at each stage yields decompositions that have fewer MVDs that violate 4NF: if  $R_i = (\bar{R}_i; \mathcal{D}_i)$  is such an intermediate schema and  $\mathcal{D}_i$  entails an MVD of the form  $\bar{R}_i = \bar{X} \bowtie \bar{Y}$ , which violates 4NF, then the algorithm replaces  $R_i$  with a pair of schemas  $(\bar{X}; \mathcal{D}_{i,1})$  and  $(\bar{Y}; \mathcal{D}_{i,2})$ . The new schemas do not have the offending MVD. Eventually, there will be no MVDs left that violate the requirements for 4NF.

Two important points regarding this algorithm need to be emphasized. First, if  $R_i = (\bar{R}_i; \mathcal{D}_i)$  is a schema and  $\bar{S} \rightarrow \bar{T} \in \mathcal{D}$  (for simplicity, assume that  $\bar{S}$  and  $\bar{T}$  are disjoint), then this FD implies the MVD  $R_i = \bar{S}\bar{T} \bowtie (\bar{R}_i - \bar{T})$ . Thus, the 4NF decomposition algorithm can treat FDs as MVDs. The other nonobvious issue in the 4NF decomposition algorithm has to do with determining the set of dependencies that hold in the decomposition. That is, if  $R_i = (\bar{R}_i; \mathcal{D}_i)$  is decomposed with respect to the MVD  $\bar{R}_i = \bar{X} \bowtie \bar{Y}$ , what is the set of dependencies that is expected to hold over the attributes  $\bar{X}$  and  $\bar{Y}$  in the resulting decomposition? The answer is  $\pi_{\bar{X}}(\mathcal{D}_i^+)$  and  $\pi_{\bar{Y}}(\mathcal{D}_i^+)$ —the projections of  $\mathcal{D}_i^+$  on  $\bar{X}$  and  $\bar{Y}$ . Here  $\mathcal{D}_i^+$  is the closure of  $\mathcal{D}_i$ , that is, the set of all FDs and MVDs entailed by  $\mathcal{D}_i$  (the optional Section 6.10 provides a set of inference rules for MVD entailment).

Projection of an FD on a set of attributes has been defined in Section 6.6.2. **Projection of an MVD**,  $\bar{R}_i = \bar{V} \bowtie \bar{W}$ , denoted  $\pi_{\bar{X}}(\bar{R}_i = \bar{V} \bowtie \bar{W})$ , is defined as  $\bar{X} = (\bar{X} \cap \bar{V}) \bowtie (\bar{X} \cap \bar{W})$ , if  $\bar{V} \cap \bar{W} \subseteq \bar{X}$ , and it is undefined otherwise. It follows directly from the definitions that the projection rule for MVDs is sound, that is, if an MVD,  $m$ , holds in a relation  $r$  then  $\pi_{\bar{X}}(m)$  holds in  $\pi_{\bar{X}}(r)$  (see Exercise 6.31).

**Example 6.9.1 (4NF Decomposition).** Consider a schema with attributes  $ABCD$  and the MVDs  $ABCD = AB \bowtie BCD$ ,  $ABCD = ACD \bowtie BD$ , and  $ABCD = ABC \bowtie BCD$ . Applying the first MVD, we obtain the following decomposition:  $AB$ ,  $BCD$ . Projection of the remaining MVDs on  $AB$  is undefined. Projection of the second MVD on  $BCD$  is  $BCD = CD \bowtie BD$ , and projection of the third MVD on  $BCD$  is  $BCD = BC \bowtie BCD$ , which is a trivial MVD. Thus, we can decompose  $BCD$  with respect to this last MVD, which yields the following final result:  $AB$ ,  $BD$ ,  $CD$ . Note that if we first decomposed  $ABCD$  with respect to the third MVD, the final result would be different:  $AB$ ,  $BC$ ,  $BD$ ,  $CD$ . ■

The design theory for 4NF is not as well developed as that for 3NF and BCNF, and very few algorithms are known. The basic recommendation is to start with a decomposition into 3NF and then proceed with the above algorithm and further decompose the offending (non-4NF) schemas. On a more sophisticated level, the work reported in [Beeri and Kifer 1986a, 1986b, 1987], among others, develops a design theory and the corresponding algorithms that can rectify design problems by synthesizing *new(!)* attributes. These advanced issues are briefly surveyed in Section 6.10.

**Example 6.9.2 (Combining 3NF Synthesis with 4NF Decomposition).** Consider the schema  $R$  over the attributes ABCDEFG with the following functional dependencies:

$$\begin{aligned} AB &\rightarrow C \\ C &\rightarrow B \\ BC &\rightarrow DE \\ E &\rightarrow FG \end{aligned}$$

and the following multivalued dependencies:

$$\begin{aligned} R &= BC \bowtie ABDEFG \\ R &= EF \bowtie FGABCD \end{aligned}$$

We begin with 3NF synthesis using the FDs only. This step is already familiar to us, so we present only the final result:

$$\begin{aligned} R_1 &= (ABC; \{AB \rightarrow C, C \rightarrow B\}) \\ R_2 &= (CBDE; \{C \rightarrow BDE\}) \\ R_3 &= (EFG; \{E \rightarrow FG\}) \end{aligned}$$

The first schema,  $R_1$ , is not in BCNF due to the FD  $C \rightarrow B$  (this FD must hold in  $R_1$  because it is in the original set of FDs and its attributes are contained within  $R_1$ ). So, we follow the BCNF decomposition algorithm and decompose  $R_1$  further using  $C \rightarrow B$ :  $R_{11} = (AC; \{A \rightarrow C\})$  and  $R_{12} = (BC; \{C \rightarrow B\})$ .

Now we still have two MVDs left. Note that  $R = BC \bowtie ABDEFG$  projects onto  $R_2$  as  $R_2 = BC \bowtie BDE$ , and it violates 4NF there because  $B = BC \cap BDE$  is not a superkey of  $R_2$ . So, we can use this MVD to decompose  $R_2$  into  $(BC; \{C \rightarrow B\})$  and  $(BDE; \{\})$ . Similarly,  $R = EF \bowtie FGABCD$  projects onto  $R_3$  as  $R_3 = EF \bowtie FG$ . This makes  $R_3$  violate 4NF, and we decompose it into  $(EF; \{E \rightarrow F\})$  and  $(FG; \{\})$ .

The resulting decomposition is not dependency preserving. For instance, the FD  $A \rightarrow B$ , which was present in the original schema, is now not derivable from the FDs that are attached to the schemas in the decomposition. ■

**The fifth normal form.** We are not going to cover the fifth normal form in this book. Suffice it to say that it exists but that the database designer usually need not be concerned with it. 5NF is similar to 4NF in that it is based on join dependencies, but unlike 4NF it seeks to preclude all nontrivial JDs (not just the binary ones) that are not entailed by a superkey.

## 6.10 Advanced 4NF Design\*

The 4NF design algorithm outlined in Section 6.9 was intended to familiarize you with MVDs and 4NF, but it only scratches the surface of the 4NF design process. In this section, we provide more in-depth information, explain the main difficulties in designing database schemas in the presence of both FDs and MVDs, and outline the solutions. In particular, we explain why the 4NF decomposition algorithm does

not truly solve the redundancy problem and why BCNF might be inadequate in the presence of MVDs. We refer you to the literature for more details.

### 6.10.1 MVDs and Their Properties

Multivalued dependencies are binary join dependencies. However, unlike general join dependencies they have a number of nice algebraic properties similar to those of FDs. In particular, a set of syntactic rules, analogous to Armstrong's axioms for FDs, exists for finding MVDs entailed by a given MVD set. These rules have a particularly simple form when we use a special notation for MVDs: It is customary to represent the multivalued dependency of the form  $\bar{R} = \bar{V} \bowtie \bar{W}$  over a relation schema  $R = (\bar{R}, D)$  as  $\bar{X} \twoheadrightarrow \bar{Y}$ , where  $\bar{X} = \bar{V} \cap \bar{W}$  and  $\bar{X} \cup \bar{Y} = \bar{V}$  or  $\bar{X} \cup \bar{Y} = \bar{W}$ . Hence,  $\bar{X} \twoheadrightarrow \bar{Y}$  is synonymous with  $\bar{R} = \bar{X} \bar{Y} \bowtie \bar{X}(\bar{R} - \bar{Y})$ .

Take a moment to understand the intuition behind this notation. An MVD arises when a single value of one attribute, for example, A, is related to a set of values of attribute B and a set of values of attribute C. Attribute A, contained in  $\bar{X}$ , can be thought of as an independent variable whose value determines (hence the symbol  $\twoheadrightarrow$ ) the associated sets of values of both B and C, one of which is contained in  $\bar{Y}$  and the other in the complement of  $\bar{X} \bar{Y}$ . For example, the MVD in the PERSON relation (6.13), SSN PhoneN  $\bowtie$  SSN ChildSSN, can be expressed as  $\text{SSN} \twoheadrightarrow \text{PhoneN}$  or  $\text{SSN} \twoheadrightarrow \text{ChildSSN}$ .

In addition, it is often convenient to combine MVDs that share the same left-hand side. For example,  $\bar{X} \twoheadrightarrow \bar{Y}$  and  $\bar{X} \twoheadrightarrow \bar{Z}$  can be represented as  $\bar{X} \twoheadrightarrow \bar{Y} | \bar{Z}$ . It is simple to show that such a pair of MVDs is equivalent to a join dependency of the form  $\bar{X} \bar{Y} \bowtie \bar{X} \bar{Z} \bowtie \bar{X}(\bar{R} - \bar{Y} \bar{Z})$ . The representation  $\bar{X} \twoheadrightarrow \bar{Y} | \bar{Z}$  is convenient not only for the inference system but also as a device that shows where the redundancy is: if the attributes of  $\bar{X}$ ,  $\bar{Y}$ , and  $\bar{Z}$  are contained within one relational schema, the associations between  $\bar{Y}$  and  $\bar{Z}$  are likely to be stored redundantly. We saw this problem in the context of the PERSON relation on page 228 and will come back to it later.

With this notation, we now present an inference system that can be used to decide entailment for *both* FDs and MVDs. The extended system contains Armstrong's axioms for FDs plus the following rules.

**FD-MVD glue.** These rules mix FDs and MVDs.

- *Replication.*  $\bar{X} \rightarrow \bar{Y}$  entails  $\bar{X} \twoheadrightarrow \bar{Y}$ .
- *Coalescence.* If  $\bar{W} \subset \bar{Y}$  and  $\bar{Y} \cap \bar{Z} = \emptyset$ , then  $\bar{X} \twoheadrightarrow \bar{Y}$  and  $\bar{Z} \rightarrow \bar{W}$  entail  $\bar{X} \rightarrow \bar{W}$ .

**MVD-only rules.** Some of these rules are similar to rules for FDs; some are new.

- *Reflexivity.*  $\bar{X} \twoheadrightarrow \bar{X}$  holds in every relation.
- *Augmentation.*  $\bar{X} \twoheadrightarrow \bar{Y}$  entails  $\bar{X} \bar{Z} \twoheadrightarrow \bar{Y}$ .
- *Additivity.*  $\bar{X} \twoheadrightarrow \bar{Y}$  and  $\bar{X} \twoheadrightarrow \bar{Z}$  entail  $\bar{X} \twoheadrightarrow \bar{Y} \bar{Z}$ .
- *Projectivity.*  $\bar{X} \twoheadrightarrow \bar{Y}$  and  $\bar{X} \twoheadrightarrow \bar{Z}$  entail  $\bar{X} \twoheadrightarrow \bar{Y} \cap \bar{Z}$  and  $\bar{X} \twoheadrightarrow \bar{Y} - \bar{Z}$ .

- *Transitivity.*  $\bar{X} \twoheadrightarrow \bar{Y}$  and  $\bar{Y} \twoheadrightarrow \bar{Z}$  entail  $\bar{X} \twoheadrightarrow \bar{Z} - \bar{Y}$ .
- *Pseudotransitivity.*  $\bar{X} \twoheadrightarrow \bar{Y}$  and  $\bar{Y}\bar{W} \twoheadrightarrow \bar{Z}$  entail  $\bar{X}\bar{W} \twoheadrightarrow \bar{Z} - (\bar{Y}\bar{W})$ .
- *Complementation.*  $\bar{X} \twoheadrightarrow \bar{Y}$  entails  $\bar{X} \twoheadrightarrow \bar{R} - \bar{X}\bar{Y}$ , where  $\bar{R}$  is the set of all attributes in the schema.

These rules first appeared in [Beeri et al. 1977], but [Maier 1983] provides a more systematic and accessible introduction to the subject. The rules are *sound* in the sense that in any relation where a rule premise holds, the consequent of the rule holds as well. For example, replication follows using the same reasoning that we used for losslessness: if  $\bar{X} \rightarrow \bar{Y}$  then the decomposition of  $\bar{R}$  into  $\bar{R}_1 = \bar{X}\bar{Y}$  and  $\bar{R}_2 = \bar{X}(\bar{R} - \bar{Y})$  is lossless; hence,  $\bar{R} = \bar{X}\bar{Y} \bowtie \bar{X}(\bar{R} - \bar{Y})$  and so  $\bar{X} \twoheadrightarrow \bar{Y}$ .

A remarkable fact, however, is that given a set,  $\mathcal{S}$ , that consists of MVDs and FDs and a dependency,  $d$  (which can be either an FD or an MVD),  $\mathcal{S}$  entails  $d$  if and only if  $d$  can be derived by a purely syntactic application of the above rules (plus Armstrong's axioms) to the dependencies in  $\mathcal{S}$ . A similar property for FDs alone was earlier called *completeness*. Proving the soundness of the above inference rules is a good exercise (see Exercise 6.24). Completeness is much harder to prove. The interested reader is referred to [Beeri et al. 1977; Maier 1983].

## 6.10.2 The Difficulty of Designing for 4NF

The inference rules for MVDs are useful because they can help eliminate redundant MVDs and FDs. Also, as in the case of FDs alone, using nonredundant dependency sets can improve the design produced by the 4NF decomposition algorithm described on page 232. However, even in the absence of redundant dependencies, things can go awry. We illustrate some of the problems on a number of examples. Three issues are considered: loss of dependencies, redundancy, and design using both FDs and MVDs.

**A contracts example.** Consider the schema

---

CONTRACTS(Buyer, Vendor, Product, Currency)

---

where a tuple of the form (John Doe, Acme, Paper Clips, USD) means that buyer John Doe has a contract to buy paper clips from Acme, Inc., using U.S. currency. Suppose that our relation represents contracts of an international network of buyers and companies. Although the contract was consummated in dollars, if Acme sells some of its products in Euros (perhaps it is a European company), it may be convenient to store the contract in two tuples: one with the financial information expressed in USD, the other with information expressed in Euros. In general, CONTRACTS satisfies the rule that, if a company accepts several currencies, each contract is described in each one. This can be expressed using the following combined MVD:

---

Buyer Vendor  $\twoheadrightarrow$  Product | Currency

---

To be explicit, this MVD means

$$\text{CONTRACTS} = (\text{Buyer } \text{Vendor } \text{Product}) \rightsquigarrow (\text{Buyer } \text{Vendor } \text{Currency})$$

The second rule of our international network example is that if two vendors supply a certain product, both accept a certain currency, and a buyer of that product has a contract to buy that product with one of the two vendors, then this buyer must have a contract for purchasing that product with the other vendor as well. For example, if, in addition to the above tuple, CONTRACTS contained *(Mary Smith, OfficeMin, Paper Clips, USD)*, then it must also contain the tuples *(John Doe, OfficeMin, Paper Clips, USD)* and *(Mary Smith, Acme, Paper Clips, USD)*. This type of constraint is expressed using the following MVD:

$$\text{Product } \text{Currency} \rightarrow\!\!\! \rightarrow \text{Buyer } | \text{Vendor}$$

6.17

Let us now attempt a design using the 4NF decomposition algorithm. If we first decompose using the dependency (6.16), we get the following lossless decomposition:

$$\begin{array}{l} (\text{Buyer}, \text{Vendor}, \text{Product}) \\ (\text{Buyer}, \text{Vendor}, \text{Currency}) \end{array}$$

6.18

Observe that once this decomposition is done, the second MVD can no longer be applied because no join dependency holds in either one of the above schemas.<sup>5</sup> The situation here is very similar to the problem we faced with the BCNF decomposition algorithm. Some dependencies might get lost in the process. In our case, it is the dependency (6.17). The same problem exists if we first decompose using the second dependency above, but in this case we lose (6.16).

Unlike losing FDs during BCNF decomposition, losing MVDs is potentially a more serious problem because the result might still harbor redundancy even if every relation in the decomposition is in 4NF! To see this, consider the following relation for the CONTRACTS schema:

Buyer	Vendor	Product	Currency
B <sub>1</sub>	V <sub>1</sub>	P	C
B <sub>2</sub>	V <sub>2</sub>	P	C
B <sub>1</sub>	V <sub>2</sub>	P	C
B <sub>2</sub>	V <sub>1</sub>	P	C

<sup>5</sup> This may not be obvious because we have not discussed the tools for verifying such facts. However, in this particular example, our claim can be checked directly using the definition of the natural join. We again recommend [Maier 1983] as a good reference for learning about such techniques.

It is easy to check that this relation satisfies MVDs (6.16) and (6.17). For instance, to verify (6.16) take the projections on decomposition schema (6.18).

Buyer	Vendor	Product
B <sub>1</sub>	V <sub>1</sub>	P
B <sub>2</sub>	V <sub>2</sub>	P
B <sub>1</sub>	V <sub>2</sub>	P
B <sub>2</sub>	V <sub>1</sub>	P

Buyer	Vendor	Currency
B <sub>1</sub>	V <sub>1</sub>	C
B <sub>2</sub>	V <sub>2</sub>	C
B <sub>1</sub>	V <sub>2</sub>	C
B <sub>2</sub>	V <sub>1</sub>	C

Joining these two relations (using the natural join) clearly yields the original relation for CONTRACTS. A closer look shows that the above relations still contain a great deal of redundancy. For instance, the first relation twice says that product P is supplied by vendors V<sub>1</sub> and V<sub>2</sub>. Furthermore, it twice says that P is wanted by buyers B<sub>1</sub> and B<sub>2</sub>. The first relation seems to beg for further decomposition into (Buyer, Vendor) and (Vendor, Product), and the second relation begs to be decomposed into (Buyer, Currency) and (Vendor, Currency). Alas, none of these wishes can be granted because none of these decompositions is lossless (for example, Vendor is not a key of the first relation). As a result, decomposition (6.18) suffers from the usual update anomalies even though each relation is in 4NF! Furthermore, since 4NF implies BCNF, even BCNF does not guarantee complete elimination of redundancy in the presence of MVDs!

**A dictionary example.** For another example, consider a multilingual dictionary relation, DICTIONARY(English, French, German), which provides translations from one language to another. As expected, every term has a translation (possibly more than one) into every language, and the translations are independent of each other. These constraints are easily captured using MVDs.

---

English → French | German  
 French → English | German  
 German → English | French

---

6.19

The problem, as before, is that applying any one of these MVDs in the 4NF decomposition algorithm loses the other two dependencies, and the resulting decomposition exhibits the usual update anomalies.

**A multilingual thesaurus example.** Let us enhance the previous example so that every term is now associated with a unique concept and each concept has an associated description. For the purpose of this example, ignore the language used for the

description. The corresponding schema becomes DICTIONARY(Concept, Description, English, French, German) and the dependencies are

---

English → Concept
French → Concept
German → Concept
Concept → Description
Concept → English   French   German

---

6.20

An example of a concept is A5329 with description “homo sapiens” and translations {human, man}, {homme}, and {Mensch, Mann}.

The 4NF decomposition algorithm suggests that we start by picking up an MVD that violates 4NF and then use it in the decomposition process. Since every FD is also an MVD, we might choose English → Concept first, which yields the schema (English, Concept) and (English, French, German, Description). Using the transitivity rule for MVDs, we can derive the MVD English → French | German | Description and further decompose the second relation into (English, French), (English, German), and (English, Description).

The resulting schema has two drawbacks. First, it is lopsided toward English whereas the original schema was completely symmetric. Second, every one of the bilingual relations, such as (English, French), redundantly lists all possible translations from English to French and back. For example, if  $a$  and  $b$  are English synonyms,  $c$  and  $d$  are French synonyms, and  $a$  translates into  $c$ , then the English-French dictionary (English, French) has all four tuples:  $(a, c)$ ,  $(a, d)$ ,  $(b, c)$ , and  $(b, d)$ .

A better way to use the 4NF decomposition algorithm is to compute the attribute closure (defined on page 204) of the left-hand side of the MVD in (6.20) with respect to functional dependencies in (6.20) and derive the following MVD by the augmentation rule:

---

Concept Description → English   French   German
---

---

We can then apply the 4NF decomposition algorithm using this MVD, which yields the decomposition (Concept, Description, English), (Concept, Description, French), and (Concept, Description, German). We can further decompose each of these relations into BCNF using the FDs alone, splitting off (Concept Description). Not only do we end up with a decomposition into 4NF, but also all dependencies are preserved.

### 6.10.3 A 4NF Decomposition How-To

The above examples make it clear that designing for 4NF is not a straightforward process. In fact, this problem was an active area of research until the early 1980s [Beeri et al. 1978; Zaniolo and Melkanoff 1981; Sciore 1983]. Eventually, all of this work was integrated into a uniform framework in [Beeri and Kifer 1986b]. While we cannot go into the details of this approach, its highlights can be explained with our three examples: contracts, dictionary, and thesaurus.

1. *The anomaly of split left-hand sides.* It is indicative of a design problem when one MVD splits the left-hand side of another, as in our contracts example. We say that an MVD  $X \twoheadrightarrow V \mid W$  splits the left-hand side of the MVD  $Y \twoheadrightarrow K \mid L$  if  $Y \cap V$ , and  $Y \cap W$  are both non-empty sets of attributes. For instance, the MVD (6.17) splits the left-hand side, (Buyer, Vendor), of (6.16), which indicates that Buyer and Vendor are unrelated attributes (every buyer is associated in some tuple with every vendor) and thus should not be in the same relation. This is precisely the reason for the redundancy that we observed in the decomposition of the CONTRACTS relation into (Buyer, Vendor, Product) and (Buyer, Vendor, Currency).

One reason for the problem with this schema might be the incorrectly specified dependencies. Instead of the MVDs given in the CONTRACT schema, the join dependency

---

Buyer Product	$\bowtie$	Vendor Product
	$\bowtie$	Vendor Currency
	$\bowtie$	Buyer Currency

---

seems more appropriate. It simply says that each buyer needs certain products, each vendor sells certain products, a vendor can accept certain currencies, and a buyer can pay in certain currencies. As long as a buyer and a vendor can match on a product and a currency, a deal can be struck. This English-language description matches the requirements in the description of the contracts example, and the designer might simply have failed to recognize that the above JD is all that is needed.

2. *Intersection anomaly.* An intersection anomaly is one in which a schema has a pair of MVDs of the form  $\bar{X} \twoheadrightarrow \bar{Z}$  and  $\bar{Y} \twoheadrightarrow \bar{Z}$  but there is no MVD  $\bar{X} \cap \bar{Y} \twoheadrightarrow \bar{Z}$ . Notice that our dictionary example has precisely this sort of anomaly: there are MVDs English  $\twoheadrightarrow$  French and German  $\twoheadrightarrow$  French, but there is no dependency  $\emptyset \twoheadrightarrow$  French. [Beeri and Kifer 1986b] argue that this is a design problem that can be rectified by inventing new attributes. In our case, the attribute Concept is missing. In fact, the thesaurus example was constructed out of the dictionary example by adding this very attribute<sup>6</sup> plus the dependencies that relate it to the old attributes. Perhaps somewhat unexpectedly, this type of anomaly can be corrected completely automatically—the new attribute and the associated dependencies can be invented by a well-defined algorithm [Beeri and Kifer 1986a, 1987].
3. *Design strategy.* Assuming that the anomaly of split left-hand sides does not arise,<sup>7</sup> a dependency-preserving decomposition of the schema  $R = (\bar{R}; \mathcal{D})$  into fourth normal form can be achieved in five steps:
  - (a) Compute attribute closure,  $X^+$ , of the left-hand side of every MVD  $X \twoheadrightarrow Y$  using the FDs entailed by  $\mathcal{D}$ . Replace every  $X \twoheadrightarrow Y$  with  $X^+ \twoheadrightarrow Y$ .

<sup>6</sup> The other attribute, Description, was added to illustrate a different point. Ignore it for the moment.

<sup>7</sup> Any such anomaly means that the dependencies are incorrect or incomplete.

- (b) Find the minimal cover of the resulting set of MVDs. It turns out that such a cover is unique if  $\mathcal{D}$  does not exhibit the anomaly of split left-hand sides.
- (c) Use the algorithm of [Beeri and Kifer 1986a, 1987] to eliminate intersection anomalies by adding new attributes.
- (d) Apply the 4NF decomposition algorithm using MVDs only.
- (e) Apply the BCNF design algorithm within each resulting schema using FDs only.

Every relation in the resulting decomposition is in 4NF and no MVD is lost on the way, which guarantees that no redundancy is present in the resulting schemas. Moreover, if the decompositions in the last stage are dependency preserving, so is the overall five-step process.

The transition from the dictionary example to the thesaurus example and then to the final decomposition of the thesaurus example is an illustration of this five-step process. Let us enhance the dictionary slightly by adding the **Description** attribute and the FDs  $\text{English} \rightarrow \text{Description}$ ,  $\text{German} \rightarrow \text{Description}$ , and  $\text{French} \rightarrow \text{Description}$  (so the dictionary example now contains four attributes). Then we can obtain a BCNF decomposition as follows:

- Apply steps (a) and (b). (Step (b) applies vacuously, as the set of MVDs is already minimal.) The resulting MVDs are  $\text{English Description} \twoheadrightarrow \text{French} \mid \text{German}$ , etc.
- Apply step (c). According to the algorithm in [Beeri and Kifer 1986a], this introduces a new attribute, **Concept**, with the exact set of dependencies depicted in the thesaurus example (6.20), except that the last MVD has a closed left-hand side:  $\text{Concept Description} \twoheadrightarrow \text{English} \mid \text{French} \mid \text{German}$ . (Of course, the algorithm does not propose the name for the newly invented attribute — this is a job for the database designer.)
- Apply step (d)—perform the 4NF decomposition with respect to the above MVD and then step (e)—apply the BCNF design process within each of the resulting schemas.

The result has four schemas, as explained in the thesaurus example: (**Concept**, **English**), (**Concept**, **French**), (**Concept**, **German**), and (**Concept**, **Description**), where **English**, **French**, and **German** are keys in the first three schemas and **Concept** in the last.

## 6.11 Summary of Normal Form Decomposition

We summarize some of the properties of the normal form decomposition algorithms discussed.

- *Third normal form* schemas might have some redundancy. The decomposition algorithm that we discussed generates 3NF schemas that are lossless and dependency preserving. It does not take multivalued dependencies into account.
- *Boyce-Codd* decompositions do not have redundancy if only FDs are considered. The decomposition algorithm we discussed generates BCNF schemas that are

lossless but that might not be dependency preserving. (As we have shown, some schemas do not have Boyce-Codd decompositions that are both lossless and dependency preserving.) It does not take multivalued dependencies into account, so redundancy due to such dependencies is possible.

- *Fourth normal form* decompositions do not have any nontrivial multivalued dependencies. The algorithm we sketched generates 4NF schemas that are lossless but that might not be dependency preserving. It attempts to eliminate redundancies associated with MVDs, but it does not guarantee that all such redundancies will go away.

Note that none of these decompositions produces schemas that have all of the properties we want.

## 6.12 Case Study: Schema Refinement for the Student Registration System

Having spent all that effort studying the relational normalization theory, we will now put the new knowledge to good use and verify our design for the Student Registration System as outlined in Section 4.8. The good news is that we did a pretty good job of converting the E-R diagram in Figure 4.33, page 114, into the relations in Figures 4.34 and 4.35, so most of the relations turn out to be in Boyce-Codd normal form. However, you did not struggle through this chapter in vain—read on!

To determine whether a schema is in a normal form we need to collect all FDs relevant to it. One source is the PRIMARY KEY and the UNIQUE constraints. However, there might be additional dependencies that are not captured by these constraints or the E-R diagram. They can be uncovered only by careful examination of the schema and of the specifications of the application, a process that requires much care and concentration. If no new dependencies are found, all FDs in the schema are the primary and the candidate keys (or the FDs entailed by them), so the schema is in BCNF. If additional FDs are uncovered, we must check if the schema is in a desirable normal form and, if not, make appropriate changes.

In our case, we can verify that all relation schemas in Figure 4.35, except CLASS, are in BCNF, as they have no FDs that are not entailed by the keys. This verification is not particularly hard because these schemas have six or fewer attributes. It is harder in the case of CLASS, which has ten.

We illustrate the process using the CLASS schema. Along the way, we uncover a missing functional dependency and then normalize CLASS. First, let us list the key constraints specified in the CREATE TABLE statement for that relation.

1. CrsCode SectionNo Semester Year → ClassTime
2. CrsCode SectionNo Semester Year → Textbook
3. CrsCode SectionNo Semester Year → Enrollment
4. CrsCode SectionNo Semester Year → MaxEnrollment
5. CrsCode SectionNo Semester Year → ClassroomId

## CHAPTER 6 Database Design with the Relational Normalization Theory

6. CrsCode SectionNo Semester Year → InstructorId
7. Semester Year ClassTime InstructorId → CrsCode
8. Semester Year ClassTime InstructorId → Textbook
9. Semester Year ClassTime InstructorId → SectionNo
10. Semester Year ClassTime InstructorId → Enrollment
11. Semester Year ClassTime InstructorId → MaxEnrollment
12. Semester Year ClassTime InstructorId → ClassroomId
13. Semester Year ClassTime ClassroomId → CrsCode
14. Semester Year ClassTime ClassroomId → Textbook
15. Semester Year ClassTime ClassroomId → SectionNo
16. Semester Year ClassTime ClassroomId → Enrollment
17. Semester Year ClassTime ClassroomId → MaxEnrollment
18. Semester Year ClassTime ClassroomId → InstructorId

Verifying that additional dependencies hold in a large schema can be difficult: one has to consider every subset of the attributes of CLASS that is not a superkey and check if it functionally determines some other attribute. This “check” is not based on any concrete algorithm. The decision that a certain FD does or does not hold in a relation is strictly a matter of how the designer understands the semantics of the corresponding entity in the real-world enterprise that is being modeled by the database, and it is inherently error prone. However, research is being conducted to help with the problem. For instance, FDEXPERT [Ram 1995] is an expert system that helps database designers discover FDs using knowledge about typical enterprises and their design patterns.

Unfortunately, we do not have an expert system handy, so we do the analysis the hard way. Consider the following candidate FD:

---

ClassTime ClassroomId InstructorId → CrsCode

---

It is easy to see why this FD does not apply: different courses can be taught by the same instructor in the same room at the same time—if all this happens in different semesters and years. Many other FDs can be rejected through a similar argument. However, since in Section 4.8 we assumed that at most one textbook can be used in any particular course, the following FD is an appropriate addition to the set of constraints previously specified for CLASS:

---

CrsCode Semester Year → Textbook

---

Although the textbook used in a course can vary from semester to semester, if a certain course is offered in a particular semester and is split in several sections because of large enrollment, all sections use the same textbook.<sup>8</sup>

It is now easy to see the problem with the design of CLASS: the left-hand side of the above dependency is not a key, and Textbook does not belong to any key either. For these reasons, CLASS is not in 3NF. The 3NF synthesis algorithm on page 224 suggests that the situation can be rectified by splitting the original schema into the following pair:

- CLASS1, with all the attributes of CLASS, except Textbook, and FDs 1, 3–7, 9–13, 15–18 (these numbers refer to the numbered list of FDs on page 241.)
- TEXTBOOKS(CrsCode, Semester, Year, Textbook), with the single FD  
 $\text{CrsCode Semester Year} \rightarrow \text{Textbook}$

Both of these schemas are in BCNF—we can verify by direct inspection that all of their FDs are entailed by key constraints. The 3NF synthesis algorithm also guarantees that the above decomposition is lossless and dependency preserving.

Let us now consider a more realistic situation in which classes can have more than one recommended textbook and all sections of the class in a particular semester use the same set of textbooks. In this case, FD (6.21) does not hold, of course. Observe that the textbooks used in any particular class are independent of meeting time, instructor, enrollment, and so forth. This situation is similar to the one in Section 6.9 relative to the PERSON relation shown in (6.14): here, the independence of the attribute Textbook from the attributes ClassTime, InstructorId, and so forth, is formally represented through the following multivalued dependency:

---


$$\begin{aligned} & (\text{CrsCode Semester Year ClassTime SectionNo} \\ & \quad \text{InstructorId Enrollment MaxEnrollment ClassroomId}) \\ & \qquad \bowtie (\text{CrsCode Semester Year Textbook}) \end{aligned}$$


---

Like the schema of the PERSON relation, CLASS is in BCNF; even so, it contains redundancy because of the above multivalued dependency. The solution to the problem is to try for a higher normal form—4NF—and, fortunately, this is easy using the algorithm in Section 6.9 on page 232. We simply need to decompose CLASS using the above dependency, which yields the following lossless decomposition (losslessness is guaranteed by the 4NF decomposition algorithm):

- CLASS1(CrsCode, Semester, Year, ClassTime, SectionNo, InstructorId, Enrollment, MaxEnrollment, ClassroomId) with the FDs 1, 3–7, 9–13, 15–18
- TEXTBOOKS(CrsCode, Semester, Year, Textbook) with no FDs

<sup>8</sup> This rule might not be true of all universities, but it is certainly true of many.

Note that the only difference between this schema and the one obtained earlier under the one-textbook-per-class assumption is the absence, in the second schema, of the FD

---

CrsCode Semester Year → Textbook

---

The result of applying relational normalization theory to the preliminary design for the Student Registration System developed in Section 4.8 is a lossless decomposition where every relation is in 4NF (and thus in BCNF as well). Luckily, this decomposition is dependency preserving, since every FD specified for the schema is embedded in one of the relations in the decomposition—something that is not always achievable with 4NF and BCNF design.

## **6.13 Tuning Issues: To Decompose or Not to Decompose?**

In this chapter, we have learned a great deal about the schema decomposition theory. However, this theory was motivated by concerns that redundancy leads to consistency-maintenance problems in the presence of frequent database updates. What if most of the transactions are read-only queries? Schema decomposition seems to make query answering harder because associations that existed in one relation before the decomposition might be broken into separate relations afterward.

For instance, finding the average number of hobbies per address is more efficient using the monolithic relation of Figure 4.13 rather than the pair of relations of Figure 6.1 because the latter requires a join before the aggregates can be computed. This is an example of the classic time/space trade-off. Adding redundancy can improve query performance. Such a trade-off has to be evaluated in the context of a particular application if the performance of a frequently executed query is found wanting. The term **denormalization** describes situations in which achieving certain normal forms incurs a punishing performance penalty, and thus perhaps there should be no decomposition.

In general, no one recommendation works in all cases. Sometimes, simulation can help resolve the issue. Here is an incomplete list of conflicting guidelines that need to be evaluated against each particular mix of transactions:

1. Decomposition generally makes answering complex queries less efficient because additional joins must be performed during query evaluation.
2. Decomposition can make answering simple queries more efficient because such queries usually involve a small number of attributes that belong to the same relation. Since decomposed relations have fewer tuples, the tuples that need to be scanned during the evaluation of a simple query are likely to be fewer.
3. Decomposition generally makes simple update transactions more efficient. However, this may not be true for complex update transactions (such as *Raise the*

*salary of all professors who taught every course required for computer science majors)* since they might involve complex queries (and thus might require complex joins).

4. Decomposition can lower the demand for storage space since it usually eliminates redundant data.
5. Decomposition can increase storage requirements if the degree of redundancy is low. For instance, in the PERSON relation of (6.14), suppose that, with few exceptions, most people have just one phone number and one child. In this situation, schema decomposition can actually increase storage requirements without bringing tangible benefits. The same applies to the decomposition of HASACCOUNT in Figure 6.7, which can increase the overhead for update transactions. The reason is that verification of the FD

---

**ClientId OfficeId → AccountNumber**

---

after an update requires a join because the attributes ClientId and OfficeId belong to different relations in the decomposition.

## BIBLIOGRAPHIC NOTES

Relational normal forms and functional dependencies were introduced in [Codd 1970]. Armstrong's axioms and the proof of their soundness and completeness first appeared in [Armstrong 1974], although more accessible exposition can be found in [Ullman 1988; Maier 1983]. An efficient algorithm for entailment of FDs first appeared in [Beeri and Bernstein 1979]. A general test for lossless decompositions was first developed in [Beeri et al. 1981]. The algorithm for synthesizing the third normal form is due to [Bernstein 1976].

The fourth normal form was introduced in [Fagin 1977], which also presents a naive decomposition algorithm and explores the relationship between 4NF and BCNF. The fifth normal form is discussed in [Beeri et al. 1977], but we recommend [Maier 1983] as a more systematic introduction to the subject. The survey in [Kanelakis 1990] is also a good starting point. Other papers on 4NF are [Beeri et al. 1978; Zaniolo and Melkanoff 1981; Sciore 1983]. Eventually, all of this work on designing 4NF schemas in the presence of FDs and MVDs was extended and integrated into a uniform framework in [Beeri and Kifer 1986a, 1986b, 1987]. More recent works on 4NF are [Vincent and Srinivasan 1993; Vincent 1999].

In-depth coverage of the relational design theory is provided in texts such as [Mannila and Raäihä 1992; Atzeni and Antonellis 1993].

As illustrated in Section 6.12, one of the most difficult obstacles to applying the results discussed in this chapter to database design is finding the right set of dependencies to use in the schema normalization process. We mentioned the FDEXPERT system [Ram 1995], which helps discover functional dependencies using

knowledge about various types of enterprises. Extensive work has also been done on the algorithms for discovering FDs, MVDs, and inclusion dependencies using the techniques from *machine learning* and *data mining* [Huhtala et al. 1999; Kantola et al. 1992; Mannila and Raäihä 1994; Flach and Savnik 1999; Savnik and Flach 1993].

## EXERCISES

- 6.1 The definition of functional dependencies does not preclude the case in which the left-hand side is empty—that is, it allows FDs of the form  $\{ \} \rightarrow A$ . Explain the meaning of such dependencies.
- 6.2 Give an example of a schema that is not in 3NF and has just two attributes.
- 6.3 What is the smallest number of attributes a relation key can have?
- 6.4 A table, ABC, has attributes A, B, and C, and a functional dependency  $A \rightarrow BC$ . Write an SQL CREATE ASSERTION statement that prevents a violation of this functional dependency.
- 6.5 Prove that every 3NF relation schema with just two attributes is also in BCNF. Prove that every schema that has *at most* one nontrivial FD is in BCNF.
- 6.6 If the functional dependency  $X \rightarrow Y$  is a key constraint, what are X and Y?
- 6.7 Can a key be the set of all attributes if there is at least one nontrivial FD in a schema?
- 6.8 The following is an instance of a relation schema. Can you tell whether the schema includes the functional dependencies  $A \rightarrow B$  and  $BC \rightarrow A$ ?

A	B	C
1	2	3
2	2	2
1	3	2
4	2	3

- 6.9 Prove that Armstrong's transitivity axiom is sound—that is, every relation that satisfies the FDs  $\bar{X} \rightarrow \bar{Y}$  and  $\bar{Y} \rightarrow \bar{Z}$  must also satisfy the FD  $\bar{X} \rightarrow \bar{Z}$ .
- 6.10 Prove the following *generalized transitivity rule*: If  $\bar{Z} \subseteq \bar{Y}$ , then  $\bar{X} \rightarrow \bar{Y}$  and  $\bar{Z} \rightarrow \bar{W}$  entail  $\bar{X} \rightarrow \bar{W}$ . Try to prove this rule in two ways:
  - Using the argument that directly appeals to the definition of FDs, as in Section 6.4
  - By deriving  $\bar{X} \rightarrow \bar{W}$  from  $\bar{X} \rightarrow \bar{Y}$  and  $\bar{Z} \rightarrow \bar{W}$  via a series of steps using Armstrong's axioms
- \*6.11 We have shown the *soundness* of the algorithm in Figure 6.3—that if  $A \in closure$  then  $A \in \bar{X}_f^+$ . Prove the *completeness* of this algorithm; that is, if  $A \in \bar{X}_f^+$ , then  $A \in closure$  at the end of the computation. Hint: Use induction on the length of derivation of  $X \rightarrow A$  by Armstrong's axioms.

- 6.12 Suppose that  $R = (\bar{R}, \mathcal{F})$  is a relation schema and  $R_1 = (\bar{R}_1; \mathcal{F}_1), \dots, R_n = (\bar{R}_n; \mathcal{F}_n)$  is its decomposition. Let  $r$  be a valid relation instance over  $R$  and  $r_i = \pi_{\bar{R}_i}(r)$ . Show that  $r_i$  satisfies the set of FDs  $\mathcal{F}_i$  and is therefore a valid relation instance over the schema  $R_i$ .
- 6.13 Let  $\bar{R}_1$  and  $\bar{R}_2$  be sets of attributes and  $\bar{R} = \bar{R}_1 \cup \bar{R}_2$ . Let  $r$  be a relation on  $\bar{R}$ . Prove that  $r \subseteq \pi_{\bar{R}_1}(r) \bowtie \pi_{\bar{R}_2}(r)$ . Generalize this result to decompositions of  $\bar{R}$  into  $n > 2$  schemas.
- 6.14 Suppose that  $R = (\bar{R}; \mathcal{F})$  is a schema and that  $R_1 = (\bar{R}_1; \mathcal{F}_1), R_2 = (\bar{R}_2; \mathcal{F}_2)$  is a binary decomposition such that neither  $(\bar{R}_1 \cap \bar{R}_2) \rightarrow \bar{R}_1$  nor  $(\bar{R}_1 \cap \bar{R}_2) \rightarrow \bar{R}_2$  is implied by  $\mathcal{F}$ . Construct a relation,  $r$ , such that  $r \subset \pi_{\bar{R}_1}(r) \bowtie \pi_{\bar{R}_2}(r)$ , where  $\subset$  denotes strict subset. (This relation, therefore, shows that at least one of these FDs is necessary for the decomposition of  $R$  to be lossless.)
- 6.15 Suppose that  $R_1, \dots, R_n$  is a decomposition of schema  $R$  obtained by a sequence of binary lossless decompositions (beginning with a decomposition of  $R$ ). Prove that  $R_1, \dots, R_n$  is a lossless decomposition of  $R$ .
- 6.16 Prove that the loop in the BCNF decomposition algorithm of Figure 6.9 has the property that the database schema at each subsequent iteration has strictly fewer FDs that violate BCNF than has the schema in the previous iteration.
- \*6.17 Prove that the algorithm for synthesizing 3NF decompositions in Section 6.8.2 yields schemas that satisfy the conditions of 3NF. (*Hint:* Use the proof-by-contradiction technique. Assume that some FD violates 3NF and then show that this contradicts the fact that the algorithm synthesized schemas out of a minimal cover.)
- 6.18 Consider a database schema with attributes  $A, B, C, D$ , and  $E$  and functional dependencies  $B \rightarrow E, E \rightarrow A, A \rightarrow D$ , and  $D \rightarrow E$ . Prove that the decomposition of this schema into  $AB, BCD$ , and  $ADE$  is lossless. Is it dependency preserving?
- 6.19 Consider a relation schema with attributes  $ABCWXYZ$  and the set of dependencies  $\mathcal{F} = \{XZ \rightarrow ZYB, YA \rightarrow CG, C \rightarrow W, B \rightarrow G, XZ \rightarrow G\}$ . Solve the following problems using the appropriate algorithms.
- Find a minimal cover for  $\mathcal{F}$ .
  - Is the dependency  $XZA \rightarrow YB$  implied by  $\mathcal{F}$ ?
  - Is the decomposition into  $XZYAB$  and  $YABCW$  lossless?
  - Is the above decomposition dependency preserving?
- 6.20 Consider the following functional dependencies over the attribute set  $ABCDEFGHI$ :

$$\begin{array}{ll}
 A \rightarrow E & BE \rightarrow D \\
 AD \rightarrow BE & BDH \rightarrow E \\
 AC \rightarrow E & F \rightarrow A \\
 E \rightarrow B & D \rightarrow H \\
 BG \rightarrow F & CD \rightarrow A
 \end{array}$$

Find a minimal cover, then decompose into lossless 3NF. After that, check if all the resulting relations are in BCNF. If you find a schema that is not, decompose it into a lossless BCNF. Explain all steps.

- 6.21 Find a projection of the following set of dependencies on the attributes  $AFE$ :

$$\begin{array}{ll} A \rightarrow BC & E \rightarrow HG \\ C \rightarrow FG & G \rightarrow A \end{array}$$

- 6.22 Consider the schema with the attribute set  $ABCDEFH$  and the FDs depicted in (6.12), page 224. Prove that the decomposition  $(AD; A \rightarrow D)$ ,  $(CE; C \rightarrow E)$ ,  $(FA; F \rightarrow A)$ ,  $(EF; E \rightarrow F)$ ,  $(BHE; BH \rightarrow E)$  is not lossless by providing a concrete relation instance over  $ABCDEFH$  that exhibits the loss of information when projected on this schema.
- 6.23 Consider the schema  $BCDFGH$  with the following FDs:  $BG \rightarrow CD$ ,  $G \rightarrow F$ ,  $CD \rightarrow GH$ ,  $C \rightarrow FG$ ,  $F \rightarrow D$ . Use the 3NF synthesis algorithm to obtain a lossless, dependency-preserving decomposition into 3NF. If any of the resulting schemas is not in BCNF, proceed to decompose them into BCNF.
- \*6.24 Prove that all rules for inferring FDs and MVDs given in Section 6.10 are sound. In other words, for every relation,  $r$ , which satisfies the dependencies in the premise of any rule,  $R$ , the conclusion of  $R$  is also satisfied by  $r$  (e.g., for the augmentation rule, prove that if  $\bar{X} \twoheadrightarrow \bar{Y}$  holds in  $r$  then  $\bar{X}\bar{Z} \twoheadrightarrow \bar{Y}$  also holds in  $r$ ).
- 6.25 Prove that if a schema,  $S = (\bar{S}, \mathcal{F})$ , is in 3NF, then every FD in  $\mathcal{F}^+$  (not only those that are in  $\mathcal{F}$ ) satisfies the 3NF requirements.
- 6.26 If  $X = \{A, B\}$  and  $F = \{A \rightarrow D, BC \rightarrow EJ, BD \rightarrow AE, EJ \rightarrow G, ADE \rightarrow H, HD \rightarrow J\}$ , what is  $X_F^+$ ? Are the FDs  $AB \rightarrow C$  and  $AE \rightarrow G$  entailed by  $F$ ?
- 6.27 Using only Armstrong's axioms and the FDs

- 
- (a)  $AB \rightarrow C$   
 (b)  $A \rightarrow BE$   
 (c)  $C \rightarrow D$
- 

give a complete derivation of the FD  $A \rightarrow D$ .

- 6.28 Consider a decomposition  $R_1, \dots, R_n$  of  $R$  obtained via steps 1, 2, and 3 (but not step 4) of the 3NF synthesis algorithm on page 224. Suppose there is an attribute  $A$  in  $R$  that does not belong to any of the  $R_i$ ,  $i = 1, \dots, n$ . Prove that  $A$  must be part of every key of  $R$ .
- 6.29 Consider the schema  $R = (ABCDEFGH, \{BE \rightarrow GH, G \rightarrow FA, D \rightarrow C, F \rightarrow B\})$ .
- Can there be a key that does not contain  $D$ ? Explain.
  - Is the schema in BCNF? Explain.
  - Use one cycle of the BCNF algorithm to decompose  $R$  into two subrelations. Are the subrelations in BCNF?
  - Show that your decomposition is lossless.
  - Is your decomposition dependency preserving? Explain.
- 6.30 Find a minimal cover of the following set of FDs:  $AB \rightarrow CD$ ,  $BC \rightarrow FG$ ,  $A \rightarrow G$ ,  $G \rightarrow B$ ,  $C \rightarrow G$ . Is the decomposition of  $ABCDFG$  into  $ABCD$  and  $ACFG$  lossless? Explain.

- 6.31 Let  $\bar{X}, \bar{Y}, \bar{S}, \bar{R}$  be sets of attributes such that  $\bar{S} \subseteq \bar{R}$  and  $\bar{X} \cup \bar{Y} = \bar{R}$ . Let  $r$  be a relation over  $\bar{R}$  that satisfies the nontrivial MVD  $\bar{R} = \bar{X} \bowtie \bar{Y}$  (i.e., neither set  $\bar{X}$  or  $\bar{Y}$  is a subset of the other).
- Prove that if  $\bar{X} \cap \bar{Y} \subseteq \bar{S}$ , then the relation  $\pi_{\bar{S}}(r)$  satisfies the MVD  $\bar{S} = (\bar{S} \cap \bar{X}) \bowtie (\bar{S} \cap \bar{Y})$ .
  - Suppose  $\bar{X}, \bar{Y}, \bar{S}$ , and  $\bar{R}$  satisfy all the above conditions, except that  $\bar{X} \cap \bar{Y} \not\subseteq \bar{S}$ . Give an example of  $r$  that satisfies  $\bar{R} = \bar{X} \bowtie \bar{Y}$  but does not satisfy  $\bar{S} = (\bar{S} \cap \bar{X}) \bowtie (\bar{S} \cap \bar{Y})$ .
- 6.32 Consider a relation schema over the attributes  $ABCDEFG$  and the following MVDs:

---


$$\begin{aligned} ABCD &\bowtie DEFG \\ CD &\bowtie ABCEFG \\ DFG &\bowtie ABCDEG \end{aligned}$$


---

Find a lossless decomposition into 4NF.

- 6.33 For the attribute set  $ABCDEFG$ , let the MVDs be:

---


$$\begin{aligned} ABCD &\bowtie DEFG \\ ABCE &\bowtie ABDFG \\ ABD &\bowtie CDEFG \end{aligned}$$


---

Find a lossless decomposition into 4NF. Is it unique?

- 6.34 Consider a decomposition  $R_1, \dots, R_n$  of  $R$  obtained through 3NF synthesis. Suppose that  $R_i$  is *not* in BCNF and let  $X \rightarrow A$  be a violating FD in  $R_i$ . Prove that  $R_i$  must have another FD,  $Y \rightarrow B$ , which will be lost if  $R_i$  is further decomposed with respect to  $X \rightarrow A$ .
- \* 6.35 This exercise relies on a technique explained in the optional Section 6.10. Consider a relation schema over the attributes  $ABCDEFGHI$  and the following MVDs and FDs:

$$\begin{array}{ll} D \rightarrow AH & D \twoheadrightarrow BC \\ G \rightarrow I & C \twoheadrightarrow B \\ & G \twoheadrightarrow ABCE \end{array}$$

Find a lossless and dependency-preserving decomposition into 4NF.



# 7

## Triggers and Active Databases

In Chapter 3, we discussed triggers in the context of reactive constraints in databases. However, triggers have other uses as well. For example, they arise naturally in applications that require **active databases**—databases that must react to various external events. In these applications, the general classes of possible external events are known but their exact timings are not. This is what makes triggers a good paradigm for these applications.

Although triggers were not a part of the SQL-92 standard, a number of database vendors include (proprietary, nonstandard) support for triggers in their products. Triggers are a part of the SQL:1999 standard, and we discuss that part of the standard in this chapter. More information on triggers in SQL:1999 can be found in [Gulutzan and Pelzer 1999].

### 7.1 What Is a Trigger?

A **trigger** is an element of the database schema that has the following structure:

---

ON *event* IF *precondition* THEN *action*

---

where **event** is a request for the execution of a particular database operation (e.g., insert a row in a table whose rows represent students registered for a course), **precondition** is an expression that evaluates to true or false (e.g., the class is full), and **action** is a statement of what needs to be done when the trigger is **fired**, that is, when the event occurs and the precondition is true (e.g., delete something from the database or send e-mail to the administrator). Because triggers are built out of the above three ingredients, they are also called **event-condition-action**, or **ECA**, rules.

Triggers fill a number of roles in database processing, including

1. **Constraint maintenance.** In Section 3.3.8, we discussed triggers that are used to maintain the foreign-key and semantic constraints. The most common form of a trigger of this kind uses the **ON DELETE** and **ON UPDATE** clauses, which are attached to foreign keys. In the same section we saw an example of a

trigger intended to enforce a semantic constraint, which prevents dropping of a course after the grade is given. More generally, triggers can be used to maintain **ASSERTION** constraints of SQL.

2. **Business rules.** A business rule is a concise formal statement of a basic principle that underlies a business process in an enterprise. For instance, a business rule encoded as a database trigger could state that if an international money transfer is made into a client's account then an e-mail message should be sent to the client. Thus, insertion of a tuple of type "international money transfer" would trigger the action of insertion of an appropriate message into an e-mail queue (which might also be a relation in the same database).
3. **Monitoring.** Complex physical objects, such as power plants, spaceships, aircraft, etc., are monitored by sensor networks, which record their measurements in a database. Since insertion of each new record in the database is an event, triggers can be used, indirectly, to monitor the state of such physical objects. For instance, if a sensor records an elevated level of carbon monoxide, then the ventilation system should be turned on and the record of this secondary event inserted into the log.
4. **Maintenance of auxiliary cached data.** Materialized views, discussed in Section 5.2.9, is one example of such a use. A trigger can update a materialized view each time a change is made to the base tables on which the trigger depends.
5. **Simplified application design.** Separating core program logic from exception handling can drastically simplify certain applications. In cases where exceptions can be modeled as update operations on a database, triggers are an ideal vehicle for such separation.

## 7.2 Semantic Issues in Trigger Handling

Surprisingly, a number of complex issues lurk behind the conceptual simplicity of the notion of a trigger. First, several types of triggers are possible, each of which might be useful for different applications. Second, we will soon discover many nuances in how and when triggers are applied. Third, at any given point in time several triggers might be activated—how should a DBMS decide which to apply and in what order? Different choices can lead to different executions.

Finally, execution of a trigger might enable other triggers. Therefore, a single event can cause a chain reaction of trigger firing, and there is no guarantee that the process will ever stop. Chain reaction may be indicative of a design problem if it cannot be shown to always terminate. To prevent infinite executions, each DBMS has a limit on the depth of such chain reactions; for instance, if the depth exceeds 32, an exception is raised, the chain reaction stops, and all of the changes made by the original update statement and the triggers are rolled back. However, this limit is a safety valve, not a feature, and trigger systems should not be designed to rely on it. Some techniques for preventing chain reaction will be discussed in Section 7.4.

**Trigger consideration.** A trigger is activated when the triggering event is requested. The **consideration** of a trigger refers to when, after activation, the precondition specified in the trigger is checked. To see why consideration is an issue, assume that when the triggering event is requested, the triggering precondition is true and so the trigger can fire. However, moments later the precondition might become false (because of updates made by this or other transactions). If the precondition is not checked immediately, the trigger will not fire.

Consider the following trigger, whose purpose is to ensure that student registration does not exceed course capacity:

---

```
ON inserting a row in course registration table
IF over course capacity
THEN abort registration transaction
```

---

When a student attempts to insert her name in the course registration table, the course might be full. Thus, if the precondition is checked when the registration attempt is made, the student's request will be rejected. However, at about the same time another student might execute a transaction to drop the course (or the registrar might have increased the course capacity), and this second transaction might commit before the first one. Therefore, if the trigger precondition is checked at the time the registration transaction commits, rather than at the event time, our student will happily register for the course. In this example, deferring the consideration of trigger preconditions might be a suitable policy.

However, if our database is monitoring a nuclear power plant and the triggering event is a pressure increase while the precondition is that the pressure not exceed a certain limit, then in all likelihood the immediate consideration of the trigger precondition is a better idea.

In summary, there are at least two useful strategies: a trigger can be considered **immediately** when the triggering event is requested, or consideration can be **deferred** until the transaction commits.

Trigger consideration is actually a little more subtle. Suppose that a trigger,  $T$ , is activated by an event,  $e$ , that affects the relation  $R$ , and let  $C$  be the condition associated with  $T$ . Many systems (SQL:1999 included) make it possible for  $C$  to take into account the state of  $R$  immediately *before*  $e$  takes place and also immediately *after*  $e$  has been executed. Therefore, if  $C$  uses only these two states of  $R$  and does not refer to any other relation in the database, the immediate and the deferred considerations of  $T$  yield the same result. Moreover, if  $C$  refers only to the *before state* of  $R$ , we can say that  $C$  is evaluated *before*  $e$  takes place! But if  $C$  does take into account database relations other than  $R$ , the two consideration modes might yield different results.

**Trigger execution.** If trigger consideration is deferred, trigger execution is necessarily also deferred until the end of the triggering transaction. However, when triggers are considered immediately we have at least two options. We can execute the trigger

immediately after its consideration, or we can defer execution until the end of the triggering transaction. Again, for a nuclear reactor database, immediate execution might be the way to go, but in less critical situations deferred execution might be a better option.

**Brain Teaser:** What would immediate execution under deferred consideration mean?

With immediate execution, there are the following further possibilities. The trigger can be executed *after* the triggering event (an **after trigger**), *before* it (a **before trigger**), or *instead* of it (an **instead-of trigger**). At first glance, the last two possibilities seem quite strange. How can an action caused by a real-life event execute before or instead of that event? The answer lies in the fact that the event is a request to the DBMS issued by a transaction, so it is quite possible for the DBMS to ignore the request and execute the trigger instead. Or the system might execute the trigger first and then allow the requested action to occur.

SQL:1999 supports only before and after triggers, but some vendors (e.g., Oracle) support instead-of triggers as well. These triggers can be useful in a number of scenarios, the most common being maintenance of views. In this scenario, the events of insertion, deletion, and update on a view can be monitored by triggers. When, say, a tuple is inserted into a view, the trigger is activated and performs appropriate insertions into the base tables of the view *instead of* inserting the tuple directly into the view. (Recall from Section 5.3.4 that in most cases direct update of a view is not even feasible because such operation is ambiguous.) Example 7.3.5 illustrates this type of trigger.

**Brain Teaser:** Do before triggers make any sense under deferred execution?

**Trigger granularity.** The issue here is what constitutes an event. **Row-level granularity** assumes that a change to a single row is an event, and changes to different rows are viewed as separate events that might cause the trigger to be executed multiple times. In contrast, **statement-level granularity** assumes that events are statements, such as **INSERT**, **DELETE**, and **UPDATE**, *not* the individual tuple-level changes they make. Thus, for instance, an **UPDATE** statement that makes no changes (because the condition in its **WHERE** clause affects no tuples currently in the database) is an event that can cause a trigger to execute!

At row-level granularity, a trigger might need to know the old and the new values of the affected tuple so it can test the precondition properly. In the case of a salary increase, for example, the old tuple contains the old salary and the new one contains the new salary. If both values are available, the trigger can verify that the increase does not exceed 10% or can apply corrective actions as appropriate. Row-level triggers usually provide access to the old and the new values of the affected tuple through special variables.

At statement-level granularity, updates are collected in temporary structures, such as OLD TABLE and NEW TABLE. This allows the trigger to query both tables and act on the basis of the results.

**Trigger conflicts.** It is possible for an event to activate several triggers at once. For instance, when a student registers for a course, the following two triggers might be considered:

---

```
ON inserting a row in course registration table
  IF over course capacity
    THEN notify registrar about unmet demands
```

---

```
ON inserting a row in course registration table
  IF over course capacity
    THEN put on waiting list
```

---

In such situations, an important question is which trigger should be considered first. Two alternatives exist.

- *Ordered conflict resolution.* Evaluate trigger preconditions in turn. When a condition is evaluated and found to be true, the corresponding trigger is executed; when that execution is complete, the next trigger is considered. In our case, the student might accept one of the alternative courses and abandon the request to add the course that is full. Therefore, by the time the second trigger is considered, its precondition is no longer true, and the trigger will not fire. One common way to order triggers is according to the times when their enabling events occur.
- *Group conflict resolution.* Evaluate all trigger preconditions at once and then schedule for execution all those whose preconditions are true. In this case, all scheduled triggers will be executed (one after another or concurrently), even if the preconditions attached to some triggers might become false shortly after their evaluation.

With the first option, the system can decide on trigger ordering or it can pick triggers at random. With the second option, trigger ordering is not necessary since all triggers can be scheduled to run concurrently, although most DBMSs do order triggers anyway.

**Triggers and integrity constraints.** In Chapter 3, we discussed the possibility of updates to the database that might violate referential integrity constraints. We saw that SQL has a way of specifying compensating actions (such as ON DELETE CASCADE) that the DBMS should take to restore integrity. These actions can be viewed as special triggers with very strict semantics. At the end of the execution, the integrity of the database must be restored. The situation is complicated by the fact that a compensating action might activate other triggers that can cause violations

of referential integrity. In this case, the scheduling of all of these triggers must have the goal of ultimately restoring the integrity constraint. The problem of trigger scheduling does not have an obvious solution. We will discuss how this issue is resolved in SQL:1999 in the next section.

## 7.3 Triggers in SQL:1999

Convergence of an agreeable syntax and semantics for triggers in the current SQL:1999 standard involved a rather long and painful process. First, the various database vendors already had triggers in their systems, so the standard had to offer sufficient benefits to convince the vendors to change their implementations. Second, as we have seen, the semantic issues associated with triggers are not trivial, and the standard would not have been accepted unless it offered reasonable solutions to the problems discussed earlier.

Armed with a new understanding of the issues associated with trigger handling, we can now approach the SQL:1999 standard systematically:

- *Triggering events.* An event can be the execution of an SQL INSERT, DELETE, and UPDATE statement as a whole or a change to individual rows made by such statements.
- *Trigger precondition.* Any condition allowed in the WHERE clause of SQL.
- *Trigger action.* An SQL query, a DELETE, INSERT, UPDATE, ROLLBACK, or SIGNAL statement, or a program written in the language of SQL's *persistent stored modules* (SQL/PSM), which smoothly integrates procedural control statements with SQL query and update statements. We discuss SQL/PSM in Chapter 8.
- *Trigger conflict resolution.* Ordered—SQL:1999 assumes that all triggers are ordered and executed in some implementation-specific way. Since the order is likely to be different from one database product to another, applications must be designed so that they do not rely on trigger ordering.
- *Trigger consideration.* Immediate—the preconditions of all triggers activated by an event are checked immediately when the event is requested.
- *Trigger execution.* Immediate—execution can be specified to be before or after the triggering event.
- *Trigger granularity.* Row-level and statement-level granularities are both available.

Here is the general syntax of SQL:1999 triggers. Constructs in square brackets are optional; clauses in curly brackets specify a choice of one of the constructs separated by vertical lines.

---

```
CREATE TRIGGER trigger-name
  {BEFORE | AFTER}
    {INSERT | DELETE | UPDATE [ OF column-name-list ]}
  ON table-name
    [ REFERENCING [ OLD AS var-to-refer-to-old-tuple ]
      [ NEW AS var-to-refer-to-new-tuple ] ]
```

---

```

[ OLD TABLE AS name-to-refer-to-old-table ] ]
[ NEW TABLE AS name-to-refer-to-new-table ] ]
[ FOR EACH { ROW | STATEMENT } ]
[ WHEN (precondition) ]
statement-list

```

---

The syntax of SQL:1999 triggers closely follows the model discussed in Section 7.2. A trigger has a name; it is activated by certain events (specified by the INSERT-DELETE-UPDATE clause); it can be defined as a BEFORE or an AFTER trigger (indicating whether the precondition is to be checked in the state that exists before or after the event); and it can have a precondition (specified by the WHEN clause). The clauses FOR EACH ROW and FOR EACH STATEMENT specify the trigger granularity. If FOR EACH ROW is specified, the trigger is activated by the changes to every individual tuple in the table watched by that trigger. If FOR EACH STATEMENT is specified (which is the default), the trigger is activated once per execution of an INSERT, DELETE, or UPDATE statement on the table being monitored, regardless of the number of tuples changed by that execution (it will be activated even if no changes occurred).

The *statement list* following the WHEN clause defines the actions to be executed when the trigger is fired. Usually, these actions are SQL statements (that insert, delete, or modify tuples), but in general they can be statements written in SQL/PSM, which can include SQL statements intermixed with if-then-else statements, loops, local variables, and so forth. We discuss SQL/PSM in Chapter 8.

The REFERENCING clause is the means of referring to the pre-update and the post-update contents of the relation *table name*. This information can be used both in the WHEN condition and in the statement list that follows.

There are two types of references, depending on the granularity of the trigger. If the trigger has row-level granularity, we can use the clauses OLD AS and NEW AS, which define tuple variables to be used to refer to the old and the new value of the tuple that caused trigger activation. If the event is an INSERT, OLD is not applicable; if it is a DELETE, NEW is not applicable. If the trigger has statement-level granularity, SQL:1999 provides access to the old and the new values of the table affected by the triggering statement. Thus, the clause OLD TABLE names the table that contains the old state of the tuples affected by the update whereas the clause NEW TABLE defines the name under which the new state of these tuples can be accessed.

**Note.** NEW AS and OLD AS specify tuple variables that range *only over the tuples affected by the update*. That is, in tuple insertion, NEW AS refers to the inserted tuple. In tuple modification, it refers to the new state of the modified tuple. OLD AS refers to deleted tuples or to the old states of modified tuples.

Likewise, OLD TABLE and NEW TABLE contain *only the tuples affected by the update*, not the entire old and new states of the table. If  $r$  was the state of a table before the update, the state after the update is  $(r - \text{old table}) \cup \text{new table}$ .

Finally, SQL:1999 imposes certain restrictions on what BEFORE and AFTER triggers can do.

**BEFORE triggers.** All BEFORE triggers execute entirely before the triggering events. They are not allowed to modify the database, but can only test the precondition specified in the WHEN clause and either accept or abort the triggering transaction. Since BEFORE triggers cannot modify the database, they cannot activate other triggers.

A typical use of BEFORE triggers is to preserve application-specific data integrity. For instance, the following trigger makes sure that course enrollment limits are never exceeded.

**Example 7.3.1 (Business Rule Enforced with a BEFORE Trigger).** Let us assume that, in addition to the already familiar relation TRANSCRIPT, the database includes a relation CRSLIMITS with the attributes CrsCode, Semester, and Limit (with their usual meanings). The following is a trigger that enforces course enrollment limits by monitoring tuple insertions into the TRANSCRIPT relation.

---

```
CREATE TRIGGER ROOMCAPACITYCHECK
  BEFORE INSERT ON TRANSCRIPT
    REFERENCING NEW AS N
  FOR EACH ROW
  WHEN
    ((SELECT COUNT(T.StudId) FROM TRANSCRIPT T
      WHERE T.CrsCode = N.CrsCode AND T.Semester = N.Semester)
     >=
    (SELECT L.Limit FROM CRSLIMITS L
      WHERE L.CrsCode = N.CrsCode AND L.Semester = N.Semester))
  ROLLBACK
```

---

Observe that an INSERT statement can insert several tuples involving different courses. The trigger specifies row granularity, so each insertion is treated as a separate event. If the course is filled to capacity, the insertion is rejected. ■

Note that the first SQL statement in the WHEN clause refers simultaneously to the new TRANSCRIPT tuples (through the tuple variable N) and to all tuples in the relation TRANSCRIPT (through the variable T). What state of the relation TRANSCRIPT is assumed while checking the validity of the WHEN condition? In the case of tuples referenced by N, the answer is clear. It must be the new state for each referenced tuple. However, it is less obvious what state is referenced by T. The answer is that BEFORE triggers assume that all referenced tables are in their old state while AFTER triggers assume the new state for each relation.

**AFTER triggers.** AFTER triggers execute entirely after the triggering event has applied its changes to the database. They are allowed to make changes to the database and thus can activate other triggers (which can cause a chain reaction, as explained earlier). In this way, AFTER triggers serve as an extension of the application logic. They can take care of various events automatically, thereby relieving application programmers of the need to code all of these event handlers in each application.

**Example 7.3.2 (Business Rule Enforced with an AFTER Trigger).** The following trigger enforces a dynamic constraint that caps any salary raises performed by a single transaction at 5%. We assume that the database has a relation called EMPLOYEE with an attribute named Salary.

---

```

CREATE TRIGGER LIMITSALARYRAISE
AFTER UPDATE OF Salary ON EMPLOYEE
REFERENCING OLD AS O
      NEW AS N
FOR EACH ROW
WHEN (N.Salary - O.Salary > 0.05 * O.Salary)
    UPDATE EMPLOYEE
        SET Salary = 1.05 * O.Salary
    WHERE Id = O.Id

```

---

Whenever the Salary attribute in an EMPLOYEE tuple is updated, the trigger causes the DBMS to compare its old and new values and, if the raise exceeds the cap, to adjust the salary increase to just 5%. If the raise does not exceed the cap or if the update is a salary decrease, the trigger does not fire. Note that the tuple variables O and N in the above statement always *refer to the same tuple* that was affected by the database update that activated the trigger. The difference is that O refers to the old state of that tuple and N refers to the new state. ■

Notice that when the trigger LIMITSALARYRAISE fires, its action overrides the effect of the original event that triggered LIMITSALARYRAISE. Furthermore, execution of the action is itself a triggering event for LIMITSALARYRAISE! However, the new event does not lead to a chain reaction. When the trigger is checked the second time, the salary actually decreases (to become exactly 5% above the original). Thus, the WHEN condition is false, and the trigger does not fire a second time.

We have seen several examples of triggers that have the granularity of a single row. However, some applications require that a trigger be fired only once per statement, after all updates specified in the statement have been processed. We illustrate the use of statement-level triggers with the following examples.

**Example 7.3.3 (Statement-Level Trigger).** Suppose that after each salary raise we want to record the new average salary for all employees. We can achieve this with the help of the following trigger:

---

```

CREATE TRIGGER RECORDNEWAVG
AFTER UPDATE OF Salary ON EMPLOYEE
FOR EACH STATEMENT
    INSERT INTO Log
        VALUES (CURRENT_DATE,
                (SELECT AVG(Salary) FROM EMPLOYEE))

```

---

When the trigger is executed, it inserts a record into the table LOG that gives the new average salary. The record also indicates the date on which the average was calculated (`CURRENT_DATE` is a built-in SQL function that returns the current date). Since it does not make sense to compute a new average after every individual salary change, statement-level granularity is better suited here than is row-level granularity. ■

The next example illustrates the use of statement-level triggers for maintaining inclusion dependencies. We discussed inclusion dependencies in Chapter 3 as a useful generalization of foreign-key constraints, which occur frequently in practical settings. One such dependency, *No professor can be scheduled to teach a course that has no registered students*, was given in (3.1) on page 45. It is a referential integrity constraint that is *not* based on foreign keys, and its representation in SQL requires the use of assertions (see (3.4) on page 52). We now show how AFTER triggers can help maintain this constraint in the presence of updates.<sup>1</sup>

**Example 7.3.4 (Maintenance of Inclusion Dependencies).** The key idea is to construct an SQL view, `IDLETEACHING`, that includes precisely those tuples from the `TEACHING` relation that describe course offerings with no corresponding tuples in the `TRANSCRIPT` relation. In other words, the view contains precisely the tuples that violate the inclusion dependency. Designing such a view definition is left as an exercise.

The trigger works as follows: after one or more students drop a class (or several classes), the trigger deletes all tuples found in `IDLETEACHING` from `TEACHING`.

---

```
CREATE TRIGGER MAINTAINCOURSESNONEMPTY
  AFTER DELETE, UPDATE OF CrsCode, Semester ON TRANSCRIPT
  FOR EACH STATEMENT
    DELETE FROM TEACHING
    WHERE
      EXISTS (SELECT *
              FROM IDLETEACHING T
              WHERE Semester = T.Semester
                AND CrsCode = T.CrsCode)
```

---

7.1

Similarly, we can construct a trigger to maintain the inclusion dependency when tuples are added to the `TEACHING` relation. This trigger should abort any transaction that tries to insert a tuple into `TEACHING` if there is no corresponding tuple in `TRANSCRIPT`. ■

Note that the above trigger might cause a chicken-and-egg problem. We cannot assign a course to a professor until somebody registers for it. However, in most schools the course schedule for the next semester is published before any student registers for any course, so some triggers might need to be created and then

<sup>1</sup> The same technique can be used to emulate the `ON DELETE` and `ON UPDATE` clauses of foreign-key constraints in systems that are not SQL-92 compliant.

destroyed. The above trigger, for instance, might need to be in effect *only* between the deadline for adding courses and the deadline for dropping them.

**Example 7.3.5 (INSTEAD OF Triggers).** Although INSTEAD OF triggers are not part of the SQL standard, this advanced feature is included in a number of database products. In this example we illustrate the approach used in the Oracle DBMS. Consider the following view:

---

```
CREATE VIEW WORKSIN(ProfId,DeptName) AS
  SELECT P.Id, D.Name
  FROM PROFESSOR P, DEPARTMENT D
 WHERE P.DeptId = D.DeptId
```

---

and suppose that the following operation is performed on the view:

---

```
DELETE FROM WORKSIN
WHERE Id = 111111111
```

---

Since the contents of a view is not stored in the database, such an operation must be translated into appropriate operations on the base tables of the view, PROFESSOR and DEPARTMENT. However, in general, there may be several translations: we could delete the department where the professor with Id 111111111 works; we can delete the professor; or we can set the DeptId field in the professor's tuple to NULL. There is no way to automatically decide which of these three possibilities is the right one. However, INSTEAD OF triggers offer a way for the application designer to specify the appropriate course of action. For instance,

---

```
CREATE TRIGGER WORKSINTRIGI
  INSTEAD OF DELETE ON WORKSIN
  REFERENCING OLD AS O
  FOR EACH ROW
    UPDATE PROFESSOR
    SET DeptId = NULL
    WHERE Id = O.ProfId
```

---

is a trigger that directs the DBMS to set the DeptId field to NULL whenever a deletion operation on the view WORKSIN is performed. ■

**Summary of the trigger evaluation procedure.** Suppose that an event,  $e$ , occurs during the execution of a database update statement,  $S$ , and this event activates a set of triggers,  $T = \{T_1, \dots, T_k\}$ . Then we can summarize the procedure for trigger processing as follows:

1. Put the newly activated triggers on the trigger queue,  $Q$ .
2. Suspend the execution of  $S$ .

3. Compute OLD and NEW if row-level granularity is used, or OLD TABLE and NEW TABLE if statement-level granularity is used.
4. Consider all BEFORE triggers in T. Execute those whose preconditions are true, and place all AFTER triggers whose preconditions are true on Q.
5. Apply the updates specified in S to the database.
6. Consider each AFTER trigger on Q according to the (implementation-dependent) priority, and execute it immediately if the triggering condition is true. If the execution of a trigger activates new triggers, execute this algorithm recursively, starting with step 1.
7. Resume the execution of statement S.

**Triggers and foreign-key constraints.** When the event that activates a trigger is invoked on a relation that has foreign-key constraints with ON DELETE and ON UPDATE clauses, the compensating actions specified in these clauses are likely to cause updates of their own. As a result, the exact semantics of the system becomes quite complicated. In fact, it took several iterations for the designers of the SQL:1999 standard to find a satisfactory solution.

One might ask why the actions attached to foreign-key constraints are not treated as regular triggers. The answer is that these constraints *are* triggers, but they have special semantics—they are intended to rectify states that violate foreign-key constraints, and it is desirable to capture this semantics in the trigger-evaluation procedure. To do so, we modify step 5:

- 5'. Apply the updates specified in S to the database (as before). For each FOREIGN KEY statement violated by the current (new) state, let act denote the associated compensating action (i.e., CASCADE, SET DEFAULT, SET NULL, or NO ACTION). Note that act is an event that can in turn activate other triggers, which we denote as  $S = \{S_1, \dots, S_n\}$ . Then
  - (a) Consider all triggers in S. Execute the BEFORE triggers whose preconditions are true, and place on Q all AFTER triggers whose preconditions are true.
  - (b) Apply the updates specified in act.

Note that in step 5'(b), we did not say whether the unprocessed triggers in S should be placed in front of Q or appended to it. The reason for this is that SQL processes triggers according to their implementation-dependent priority.

Observe that execution of AFTER triggers in step 6 can activate other triggers and also cause violation of foreign-key constraints. In this case, steps 1 through 6 are invoked recursively.

The above algorithm is designed to handle very complex interactions of triggers and foreign-key constraints—interactions that might involve dozens of triggers. We illustrate the algorithm on a simple example that involves just two triggers and one foreign-key constraint.

**Example 7.3.6 (Interaction of Triggers and Foreign-Key Constraints).** Let us assume that courses mentioned in the TRANSCRIPT relation must also be listed in the COURSE

relation. Both of these relations are described in Figure 3.4 on page 38. The foreign-key constraint between these relations, which we call **CHECKCOURSEVALIDITY**, can be expressed as follows:

---

```
CREATE TABLE TRANSCRIPT (
    StudId   INTEGER,
    CrsCode  CHAR(6),
    Semester CHAR(6),
    Grade    grades,
    PRIMARY KEY (StudId, CrsCode, Semester),
    CONSTRAINT CHECKCOURSEVALIDITY
        FOREIGN KEY (CrsCode) REFERENCES COURSE (CrsCode)
        ON DELETE CASCADE
        ON UPDATE CASCADE )
```

---

**CHECKCOURSEVALIDITY** deletes or updates all **TRANSCRIPT** tuples if the corresponding **COURSE** tuple is deleted or updated.

Suppose, in addition, that there is an **AFTER** trigger, **WATCHCOURSEHISTORY**, that records all changes to the tuples in the **COURSE** relation. We leave it to Exercise 7.5 to define this trigger in SQL. Finally, the trigger **MAINTAINCOURSES-NONEMPTY** (7.1) is also part of the database scheme. Note that in this example we do not consider other foreign-key constraints (in particular, those associated with the **TEACHING** relation—including such constraints would make a much more complex example).

Suppose now that some course code is changed in the relation **COURSE**, specifically CS305 becomes CS405 beginning with fall 2000. This change activates the trigger **WATCHCOURSEHISTORY** and the **CHECKCOURSEVALIDITY** foreign-key constraint. Because **WATCHCOURSEHISTORY** is an **AFTER** trigger, it is placed on **Q**, and the trigger-processing algorithm handles the foreign-key constraint first as required in step 5'(b). Therefore, all tuples in the **TRANSCRIPT** relation that have CS305 in them are changed to refer to CS405.

This change activates the second trigger, **MAINTAINCOURSESNONEMPTY**. Since CS305 has been changed to CS405, the professor who is listed as teaching CS305 in fall 2000 is left without a class. In other words, CS305 is listed in **TEACHING** for the fall 2000 semester, but **TRANSCRIPT** does not have any corresponding tuples. Therefore, the **WHEN** condition in **MAINTAINCOURSESNONEMPTY** is true, and the trigger can be executed.

Because **MAINTAINCOURSESNONEMPTY** is an **AFTER** trigger, it is put on **Q** (step 5'(b)), which already contains **WATCHCOURSEHISTORY**. The order in which these two triggers in the queue are actually fired depends on the implementation of the particular DBMS being used and cannot be predicted.<sup>2</sup> When all triggers eventually

<sup>2</sup> Most vendors use scheduling strategies based on time stamps that reflect the time of trigger consideration. In our case, such time stamp ordering favors **WATCHCOURSEHISTORY**.

fire, the record about the course change goes into the history log and the teaching assignment for CS305 is deleted.

Note that the interaction of the two triggers and the foreign-key constraint described above might not yield the intended result in this example. For instance, it might be more reasonable to update the teaching assignment of CS305 to a teaching assignment of CS405. ■

## 7.4 Avoiding a Chain Reaction

The possibility of a never-ending chain reaction in trigger execution is a serious concern. As mentioned earlier, commercial DBMSs impose an *a priori* upper limit on the length of chain reactions. However, relying on this upper limit is not a good idea since it is hard to predict the final outcome.

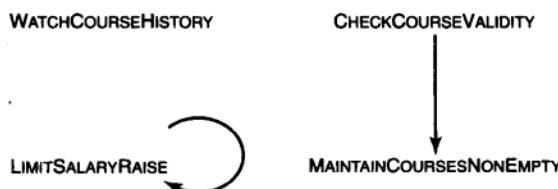
Trigger systems in which chain firing terminates in all cases are called **safe**. Unfortunately, there is no algorithm that can tell whether any given set of triggers is safe. However, there are conditions that are sufficient to guarantee safety (i.e., if the conditions are satisfied, the triggers are safe), but they are not necessary (i.e., a set of triggers might be safe but not satisfy the conditions). The fact that there is no algorithm to test safety implies that there can be no verifiable necessary *and* sufficient condition for safety.

In view of this sorry state of affairs, we present one condition that is sufficient to guarantee safety but that rejects many perfectly safe trigger systems. A **triggering graph** is a graph whose nodes are triggers (or foreign-key constraints as a special case). An arc goes from trigger  $T$  to trigger  $T'$  if and only if execution of  $T$  is an event that can activate  $T'$ .

It is easy to see that one can use a simple syntactic analysis to determine whether one trigger might activate another trigger. Indeed, the events that enable a trigger are listed in the BEFORE/AFTER clause in the trigger definition (or in the ON DELETE/UPDATE clause in foreign-key constraint definitions); the events *caused* by the triggers can be determined from the statements in the trigger body. Figure 7.1 shows the triggering graph for some of the triggers discussed in this chapter.

Clearly, if the trigger graph is *acyclic*, it is not possible for the triggers to invoke each other in a nonterminating manner. By this criterion, the triggers **WATCHCOURSEHISTORY**, **MAINTAINCOURSESNONEMPTY**, and **CHECKCOURSEVALIDITY** cannot be involved in a chain reaction.

**FIGURE 7.1** Cyclic trigger graph. The cycle does not cause a chain reaction.



Even though this method can certify the safety of some systems, it fails in many cases. Indeed, the part of our triggering graph that involves `LIMITSALARYRAISE` is cyclic because syntactic analysis shows that its execution activates this very trigger again. However, this trigger will fire only once because its `WHEN` condition will be false on the second invocation. This analysis exposes a major weakness of the triggering graph method. It does not take into account the semantics of the triggering conditions associated with the triggers. For example, if one can verify that no cycle in the triggering graph can be traversed infinitely many times (as in our example), the trigger system is safe.

There are a number of enhancements to the triggering graph methods, but they are outside of the scope of this book.

## BIBLIOGRAPHIC NOTES

The main concepts underlying triggers in databases are described in [Paton et al. 1993]. The algorithm for integration of triggers with foreign-key constraints originates in [Cochrane et al. 1996]. The syntax of SQL:1999 triggers is described in recent guides to SQL, such as [Gulutzan and Pelzer 1999].

There is a vast body of literature on active databases; the information on SQL:1999 triggers provided here is only the tip of an iceberg. The interested reader is referred to [Widom and Ceri 1996] for a comprehensive study.

## EXERCISES

- 7.1 Explain the semantics of the triggers that are available in the DBMS that is used for your course project. Describe the syntax for defining these triggers.
- 7.2 Give the exact syntactic rules for constructing the triggering graphs from the sets of SQL triggers and foreign-key constraints.
- 7.3 Design a trigger that complements the trigger `MAINTAINCOURSESNONEMPTY` (see (7.1) on page 260) by precluding the insertion of tuples into the relation `TEACHING` when there are no corresponding tuples in the `TRANSCRIPT` relation.
- 7.4 Design a trigger that works like `MAINTAINCOURSESNONEMPTY` but is a row-level trigger.
- 7.5 Define the trigger `WATCHCOURSEHISTORY` that uses a table `LOG` to record all changes that transactions make to the various courses in the `COURSE` relation.
- 7.6 Define triggers that fire when a student drops a course, changes her major, or when her grade average drops below a certain threshold. (For simplicity, assume that there is a function, `grade_avg()`, which takes a student Id and returns the student average grade.)
- 7.7 Consider the `IsA` relationship between `STUDENT(Id,Major)` and `PERSON(Id, Name)`. Write the triggers appropriate for maintaining this relationship: when a tuple is deleted from `PERSON`, the tuple with the same Id must be deleted from `STUDENT`;

when a tuple is inserted into STUDENT, check whether a corresponding tuple exists in PERSON and abort if not. (Do not use the ON DELETE and ON INSERT clauses provided by the FOREIGN KEY statement.)

- 7.8 Consider a brokerage firm database with relations HOLDINGS(AccountId, StockSymbol, CurrentPrice, Quantity) and BALANCE(AccountId, Balance). Write the triggers for maintaining the correctness of the account balance when stock is bought (a tuple is added to HOLDINGS or Quantity is incremented), sold (a tuple is deleted from HOLDINGS or Quantity is decremented), or a price change occurs.

Solve the problem using both row-level and statement-level triggers. Give an example of a situation when row-level triggers are more appropriate for the above problem and when statement-level triggers are more appropriate.

- 7.9 Consider an enterprise in which different projects use parts supplied by various suppliers. Define the appropriate tables along with the corresponding foreign-key constraints. Define triggers that fire when a project changes a supplier for a part; when a supplier discontinues a part; or when a project stops using a part.
- 7.10 Consider triggers with immediate consideration and deferred execution. What do OLD AS and NEW AS refer to during consideration and during execution?
- 7.11 Give an example of an application where SQL:1999 triggers could be used for a purpose other than just maintaining integrity constraints.

# 8

## Using SQL in an Application

In the previous chapters, we discussed SQL as an interactive language. You type in a query, anxiously listen to your hard drive, and then see the results appear on your screen (or more likely scroll by on the screen too quickly to be read). This mode of execution, called **direct execution**, was part of the original vision of SQL.

In most transaction processing applications, however, SQL statements are part of an application program written in some conventional language, such as C, Cobol, Java, or Visual Basic, and the program executes on a computer different from the one on which the database server resides. In this chapter, we discuss some advanced features of SQL that address the issues involved in this type of execution. Our goal is to present the basic concepts involved, not to cover all the syntactic options.

### 8.1 What Are the Issues Involved?

We are interested in creating programs that involve a mixture of SQL statements and statements from a conventional language. The SQL statements enable the program to access a database. The conventional language, called the **host language**, supplies features that are unavailable in SQL. These features include control mechanisms, such as the **if** and **while** statements, assignment statements, and error handling.

In our discussion of how SQL statements can be included in a host language, we must deal with two issues, discussed here:

1. Prior to executing an SQL statement, a **preparation** step is performed. Preparation involves parsing the statement and then making a *query execution plan*, which determines the sequence of steps necessary for statement execution. In what order will tables be joined? Should the tables be sorted first? What indices will be used? What constraints will be checked? Because the execution of a single SQL statement can involve considerable computational and I/O resources, it is essential that it be carefully planned. The query execution plan is designed by the DBMS using the database schema and the structure of the statement: the statement type (e.g., **SELECT**, **INSERT**), the tables and columns accessed, and the column domains. Factors such as the number of rows in a table might also

be taken into account in query optimization. The SQL statement is executed according to the sequence of steps outlined in the plan.

2. SQL constructs can be included in an application program in two different ways:

- (a) **Statement-level interface (SLI).** The SQL constructs appear as new statement types in the program. The program is then a mixture of statements in two languages: the host language and the new statement types. Before the program can be compiled by the host language compiler, the SQL constructs must be processed by a **precompiler**, which translates the constructs into calls to host language procedures. The entire program can then be compiled by the host language compiler. At run time, these procedures communicate with the DBMS, which takes the actions necessary to cause the SQL statements to be executed.

The SQL constructs can take two forms. In the first, referred to as **embedded SQL**, they are ordinary SQL statements (e.g., SELECT, INSERT). In the second, they are directives for preparing and executing SQL statements, but the SQL statements *appear in the program as the values of string variables that are constructed by the host language portion of the program at run time*. Since in this case the actual SQL statements to be executed might not be known at compile time, this form is referred to as **dynamic SQL**. This is in contrast to embedded SQL, where the SQL statements are known at compile time and are written directly into the program. Hence, embedded SQL is also referred to as **static SQL**. SQL-92 defines a standard for embedded SQL. We also discuss SQLJ—a version of SLI designed specifically for Java—which was standardized in SQL:2003.

- (b) **Call-level interface (CLI).** Here, unlike static and dynamic SQL, the application program is written entirely in the host language. As with dynamic SQL, SQL statements are the values of string variables constructed at run time. These variables are passed as arguments to host language procedures provided by the CLI. Since no special syntax is used, no precompiler is needed.

We discuss two CLIs in this chapter: **JDBC** (Java DataBase Connectivity), which is specifically designed for the Java language, and **ODBC** (Open DataBase Connectivity), which can be used with many languages. JDBC was standardized in SQL:2003 and has an interface very similar to ODBC in SQL:1999.

## 8.2 Embedded SQL

Embedded SQL is a statement-level interface that allows SQL statements to be embedded in a host language program. The schema of the database to be accessed by the program must be known at the time the program is written so that the SQL statements can be constructed. For example, the programmer must know the names of tables and the names and domains of columns.

Before the compilation of the program by the host language compiler, a precompiler (usually supplied by the vendor of the DBMS) scans the application program and locates the embedded SQL statements. These statements are not part of the host language, so they cannot be processed by the host language compiler. Instead they are set off by a special syntax so that the precompiler can recognize them. The pre-

compiler translates each statement into a sequence of subroutine calls in the host language to a run-time library, which can be processed by the host language compiler at the next stage. Later, when the program is run and the SQL statement is to be executed, the subroutines are called and they send the SQL statement (that was originally embedded in the application) to the DBMS, which prepares and executes it.

It would be reasonable for the precompiler to check the form of each SQL statement and prepare a query execution plan since that would eliminate a significant source of run-time overhead. However, most precompilers do not do this. Preparation requires the precompiler to communicate with the DBMS (to determine the schema of the database that the statement is accessing), and this communication might not be possible at compile time. Furthermore, since the query execution plan might depend on the size of tables, the closer in time the preparation is to the execution, the better.

In the best of all possible worlds, the embedded SQL constructs would be written in some standardized version of SQL (e.g., SQL-92 or SQL:1999), and the precompiler for each DBMS would perform any necessary translation to the dialect of SQL recognized by that DBMS. In the real world, however, most precompilers do not perform such translations, and the SQL constructs must be written in the exact dialect of the DBMS being accessed. In practice, then, the DBMS, as well as the database schema, must be known at the time the program is written.

Requiring the application program to use the exact dialect of the DBMS can be a disadvantage if, at some later time, it becomes necessary to change to a different DBMS with a different dialect. In some situations, however, using the dialect of the DBMS can be an advantage. Many DBMSs contain proprietary extensions to SQL. If the SQL embedded in the host language is exactly SQL-92, those extensions are not available to the programmer. Of course, if the proprietary extensions supported by a particular DBMS are used in an application, the difficulty of changing to a different DBMS at a later time increases.

The SQL standard requires that all implementations of embedded SQL provide precompilers for at least seven host languages: Ada, C, COBOL, Fortran, M (formerly known as MUMPS), Pascal, and PL/I. In practice, precompilers are available for other languages as well.

Figure 8.1 is a fragment of a C program with embedded SQL statements. Each embedded SQL statement is preceded by the words EXEC SQL, so it can be located by the precompiler. We use the syntax of SQL-92, but be aware that many database vendors use their own dialect of SQL.

All examples in this chapter come from the following two schemas:

---

```
CLASS(CrsCode:CHAR(6), Semester:CHAR(6),
      Enrollment:INTEGER, ProfId:CHAR(9), Room:CHAR(10))
The Key of CLASS: {CrsCode, Semester}
TRANSCRIPT(StudId:INTEGER, CrsCode:CHAR(6), Semester:CHAR(6),
            Grade:CHAR(1))
The Key of TRANSCRIPT: {StudId, CrsCode, Semester}
```

---

**FIGURE 8.1** Fragment of an embedded SQL program written in C.

```

EXEC SQL BEGIN DECLARE SECTION;
    unsigned long num_enrolled;
    char *crs_code, *semester;
    :

EXEC SQL END DECLARE SECTION;
    : other host language declarations and statements
    : statements to set the variables semester and crs_code

EXEC SQL SELECT C.Enrollment
    INTO :num_enrolled
    FROM CLASS C
    WHERE C.CrsCode = :crs_code
        AND C.Semester = :semester;
    : the rest of the host language program

```

---

The domains of the attributes `CrsCode`, `Semester`, and `Grade` are the same as in Figure 3.5, page 39. That is, course codes are strings of the form `MAT123` or `CS305`, semesters are strings of the form `F1999` or `S2000`, and grades are letters, such as `A` or `B`.

For the application program as a whole to communicate with the database, host language statements and SQL statements must be able to access common variables. In that way, results computed by the host language portion of the program can be stored in the database, and data extracted from the database can be processed by host language statements.

The first group of statements in the fragment of Figure 8.1 declares variables of the host program, or **host variables**, that are used for that purpose. The declarations are included between `EXEC SQL BEGIN DECLARE SECTION` and `EXEC SQL END DECLARE SECTION` so that they can be easily found and processed by the precompiler. However, the declarations themselves are *not* preceded by `EXEC SQL`. In this way, the declarations can be processed by both the precompiler and the host language compiler.

Host variables are used in the `SELECT` statement shown in Figure 8.1. Note the colon that precedes each use of a host variable in the `SELECT` statement to differentiate it from the table and column names of the database schema. The value of `Enrollment` is returned in the host variable `num_enrolled` and can be accessed by host language statements in the normal way after the `SELECT` statement has been executed.

Since `CrsCode` and `Semester` together form the primary key of `CLASS`, the `SELECT` statement returns a *single* row. This is an important point. If the `SELECT` statement returned more than one row, which one would be used to provide the

value for the variable `num_enrolled`? For this reason, it is an error for a `SELECT INTO` statement to return more than one row. We address the case in which the result consists of multiple rows in Section 8.2.4.

We can think of the host language variables as parameterizing the SQL statement. They are used to communicate scalar values, not table or column names or structured data. Host language variables that occur in `WHERE` clauses correspond to **in parameters**, while those used in `INTO` clauses correspond to **out parameters**. When the statement is executed, the values of the `in` parameters are used to form a complete SQL statement that can be executed by the database manager. Note, however, that the SQL statement can be prepared before the values of the `in` parameters are determined because, for example, table and column names are known (they cannot be parameters). Therefore, the query execution plan used when the statement is first executed can be saved for subsequent executions of the same statement (since only the parameter values differ on each execution). This is an important advantage of embedded SQL. One function of the precompiler is to select routines that, at run time, move values into and out of host language variables and to handle formatting for communication with the DBMS.

### 8.2.1 Status Processing

In the real world, things do not always proceed smoothly. For example, when you attempt to connect to a database on a distant server, the server might be down or it might reject the connection. Or an `INSERT` statement that you attempt to execute might be rejected by the DBMS because it would cause a constraint violation. You might categorize these as error situations since the requested action did not occur. In other situations, an SQL statement might execute correctly and return information describing the outcome of the execution. For example, the `DELETE` statement returns the number of rows deleted. SQL provides two mechanisms for returning information describing such situations to the host program: a five-character string `SQLSTATE` and a `diagnostics area`.

In Figure 8.2, we have added status processing to the fragment shown in Figure 8.1. `SQLSTATE` is declared within the declaration section since it is used for communication between the DBMS and the host language portion of the application. (It is declared as a six-character string when SQL is embedded in C, to account for the additional null character that terminates strings in C.) Note that `SQLSTATE` is not preceded by a colon when used in SQL statements because it is recognized by the preprocessor as a special keyword.

This declaration is required in all embedded SQL programs. (Earlier versions of SQL use a slightly different technique. Status is communicated through an integer variable, `SQLCODE`.) The DBMS sends information to be stored in that string after each SQL statement is executed. The statement can then be followed by a (host language) conditional statement that checks the value of `SQLSTATE`. If that value is 00000, the last SQL statement executed successfully. If not, the particular exception situation can be determined, and appropriate action can be taken. In Figure 8.2, a status message is printed.

**FIGURE 8.2** Adding some status processing.

```

#define OK "00000"
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    unsigned long num_enrolled;
    char *crs_code, *semester;
EXEC SQL END DECLARE SECTION;
    : other statements; get the values for crs_code, semester
EXEC SQL SELECT C.Enrollment
    INTO :num_enrolled
    FROM CLASS C
    WHERE C.CrsCode = :crs_code
        AND C.Semester = :semester;
if (strcmp(SQLSTATE,OK) != 0)
    printf("SELECT statement failed\n");

```

---

Instead of checking status after each SQL statement, we can include a single **WHENEVER** statement anywhere before the first SQL statement is executed.

---

**EXEC SQL WHENEVER SQLERROR GOTO label;**

---

Then any nonzero status in a subsequently executed statement causes a transfer of control to **label**. The **WHENEVER** statement remains in effect until another **WHENEVER** statement is executed.

More detailed information on the outcome of the last executed SQL statement can be retrieved from the diagnostics area using a **GET DIAGNOSTICS** statement. A single SQL statement can raise several exceptions. The diagnostics area records information about all exceptions raised.

Before this becomes a problem, we should mention one confusing issue: the difference in string notation in SQL (including all of its components, such as embedded SQL) and many of the host languages, such as C and Java. In SQL, strings are set in single quotes, while in C and Java they are set in double quotes. Thus, a C program with embedded SQL can have both kinds of notation. For instance, in

---

```

semester = "F2000";
EXEC SQL SELECT C.Enrollment
    INTO :num_enrolled
    FROM CLASS C
    WHERE C.CrsCode = 'CS305'
        AND C.Semester = :semester;

```

---

the string F2000 appears in a regular assignment statement and is processed by the C compiler, while CS305 occurs in an SQL statement and is handled by the SQL preprocessor.

### 8.2.2 Sessions, Connections, and Transactions

We introduce some terminology from the SQL standard. Before an application program that includes SQL statements can perform any database operations, it must establish an **SQL connection** to an **SQL server**. That connection initiates an **SQL session** on the server. Once an SQL session has been established, the application program can execute any number of **SQL transactions**, until it disconnects from the server, breaking the SQL connection and ending the SQL session.

SQL connections are established by executing a CONNECT statement (possibly implicitly), the general form of which is

---

```
CONNECT TO {DEFAULT | db-name-string}  
[AS connection-name-string] [USER user-id-string]
```

---

Phrases in square brackets are optional. Phrases in curly brackets refer to alternatives: one of the enclosed phrases separated by a vertical line must be chosen.

The option *db-name-string* is the name of the data source, *connection-name-string* is the name that *you* give to the connection, and *user-id-string* is the name of a user account; it is used by the data source for authorization. The format used to specify a data source depends on the vendor. It can be a string that identifies the database by name on a local machine or something like

---

```
tcp:postgresql://db.xyz.edu:100/studregDB
```

---

on a remote machine.

A program can execute additional CONNECT statements to different servers, after which the new connection and the new session become current and the previous connection and session become dormant. The program can switch to a dormant connection and session by executing

---

```
SET CONNECTION TO {DEFAULT | connection-name-string}
```

---

SQL connections and SQL sessions are terminated by executing (possibly implicitly)

---

```
DISCONNECT {DEFAULT | db-name-string}
```

---

### 8.2.3 Executing Transactions

There is no explicit SQL-92 statement that initiates a transaction. A transaction is initiated automatically when the first SQL statement that accesses the database is executed within a session.

In SQL:1999, transactions can also be initiated by using a **START TRANSACTION** statement that initiates a transaction and specifies certain of its characteristics similarly to the **SET TRANSACTION** statement, described below, in this section.

In SQL-92, transactions can be terminated with either **COMMIT** or **ROLLBACK**. The next SQL statement (after **COMMIT** or **ROLLBACK**) immediately starts a new transaction. This is referred to as **chaining**.

SQL:1999 also has **COMMIT AND CHAIN** and **ROLLBACK AND CHAIN** statements, which start a new transaction immediately after the commit or rollback completes without waiting until the start of the next SQL statement.

The default mode of execution for transactions is **READ/WRITE**, meaning that the transaction can both read and make changes to the database. Alternatively, it can be restricted to **READ ONLY** access to protect the database from unauthorized changes.

In Section 2.3, we pointed out that, although only serializable schedules guarantee correct execution for all applications, less demanding levels of isolation can often be used for a particular application to improve performance. Hence, the default isolation level is **SERIALIZABLE**, but other levels are offered as well. We discuss these levels at some length in Chapter 13.

If a mode of execution other than the default mode is wanted, the **SET TRANSACTION** statement can be used. For example,

---

```
SET TRANSACTION READ ONLY
ISOLATION LEVEL READ COMMITTED
DIAGNOSTICS SIZE 6;
```

---

sets the mode to **READ ONLY** and the isolation level to **READ COMMITTED**. The following isolation levels are defined:

---

```
READ UNCOMMITTED
READ COMMITTED
REPEATABLE READ
SERIALIZABLE
```

---

The **DIAGNOSTICS SIZE** clause determines the number of exception conditions (caused by the last executed SQL statement) that can be described at one time in the diagnostics area.

Figure 8.3 illustrates the use of connection and transaction statements as well as status processing. After the declarations, the next set of statements makes a connection to the server. The program does not use explicit statements for beginning

**FIGURE 8.3** The use of connection and transaction statements in a C program with embedded SQL statements whose purpose is to deregister a student from a course.

```
#define OK "00000"
EXEC SQL BEGIN DECLARE SECTION;
    unsigned long stud_id;
    char *crs_code, *semester;
    char SQLSTATE[6];
    char *dbName;
    char *connectName;
    char *userId;
EXEC SQL END DECLARE SECTION;

// Get values for dbName, connectName, userId
dbName = "studregDB";
connectName = "conn1";
userId = "ji21";

: other statements

EXEC SQL CONNECT TO :dbName AS :connectName USER :userId;
if (strcmp(SQLSTATE,OK) != 0)
    exit(1);

: get the values for stud_id, crs_code, etc.

EXEC SQL DELETE FROM TRANSCRIPT
    WHERE StudId = :stud_id
        AND Semester = :semester
        AND CrsCode = :crs_code;

if (strcmp(SQLSTATE,OK) != 0)
    EXEC SQL ROLLBACK;
else {
    EXEC SQL UPDATE CLASS
        SET Enrollment = (Enrollment - 1)
        WHERE CrsCode = :crs_code
            AND Semester = :semester;

    if (strcmp(SQLSTATE,OK) != 0)
        EXEC SQL ROLLBACK;
    else
        EXEC SQL COMMIT;
}
EXEC SQL DISCONNECT :connectName;
```

a transaction; instead, a transaction is implicitly started once the connection is established.

The figure shows a fragment of the program that deregisters a student from a course. We assume that the host language variable `semester` contains the current semester, which can be determined through a call to the operating system, such as `time()`. The program makes two modifications to the database state: deleting the row of `TRANSCRIPT` that indicates that the student is registered in the course, and decrementing the `Enrollment` attribute of the course's tuple in `CLASS`. If either the `DELETE` statement or the `UPDATE` statement fails (and hence the value of `SQLSTATE` is not "00000" when the statement completes), the `ROLLBACK` command is executed. Otherwise, the `COMMIT` command is executed. Then the transaction disconnects.

If the `DELETE` statement fails, no modification has been made to the database, which might lead us to think that the `ROLLBACK` command is not needed. However, the system must be notified that the transaction has completed so that, for example, an appropriate entry can be made in the system log and any locks acquired by the transaction can be released. The log is part of the mechanism the system uses to ensure transaction atomicity.

When an application executes either `EXEC SQL COMMIT` or `EXEC SQL ROLLBACK`, it is requesting that the database server,  $S$ , to which the application has a current connection, commit or roll back any changes it has made to the database at  $S$ . However, the application program might be executing a transaction that does more than just access a single database server. For example, by establishing several connections and switching among them, it might be accessing several database servers, or it might be putting the results of its computation into a local file system. The `COMMIT` and `ROLLBACK` statements are sent over the connection to  $S$  and therefore do not apply to these tasks.

If the program is connected to more than one database, the transactions at each can be separately committed or rolled back. However, the global transaction consisting of all of the separate transactions might not be atomic (for example if the transaction at one database commits and the transaction at another database is rolled back).

### 8.2.4 Cursors

One of the advantages of SQL as a database language is that its statements can deal with entire tables. Thus, a `SELECT` statement might return a table, which we refer to as the **query result** or **result set**. When the statement is executed in direct or interactive mode rather than embedded in an application program, the result set scrolls out on the screen. The following `SELECT` statement, for example, returns the IDs and grades of all students enrolled in a particular course in a given semester.

---

```
EXEC SQL SELECT T.StudId, T.Grade
  FROM TRANSCRIPT T
 WHERE T.Semester = :semester
   AND T.CrsCode = :crs_code;
```

---

Suppose that we want to include such a statement in a host language program. The number of rows in the result set is not known until the statement is executed, so we face the problem of allocating storage within the program for an unknown number of rows. For example, if an array is to be used, how large should the array be?

This problem points up a fundamental difference between SQL and the host language. The fundamental unit dealt with by an SQL statement is a set of tuples, whereas the fundamental unit dealt with by a statement in the host language is a variable. This difference is often called an **impedance mismatch**.

The SQL mechanism for solving this problem is the **cursor**, which allows the application program to deal with one row in a result set at a time. Think of a cursor as a pointer to a row in the result set. A **FETCH** statement fetches the row pointed to by the cursor and assigns the attribute values in the row to host language variables in the program. In this way, variables need be allocated only for a single row. From the database schema, we know the types of the values in each row and so can declare variables of the appropriate type.

Figure 8.4 is a fragment of an embedded SQL program that uses cursors. The **DECLARE CURSOR** statement declares the name of the cursor as **GETENROLLED**, specifies it as **INSENSITIVE** (a qualification we discuss shortly), and associates it with a particular **SELECT** statement. It does not, however, cause that statement to be executed. The statement is executed when the **OPEN** statement is executed.

In the example, the associated **SELECT** statement is parameterized. Only tuples whose attributes **CrsCode** and **Semester** match the values stored in the host variables **crs\_code** and **semester** are selected. When the **OPEN** statement is executed, parameter substitution takes place and then the **SELECT** statement is executed. Hence, changes to the values of the parameters made after the cursor is opened have no effect on the tuples that are retrieved through it. **OPEN** positions the cursor prior to the first row in the result set.

When the **FETCH** statement is executed, the cursor is advanced. Thus, the **FETCH** statement in Figure 8.4 points the cursor to the first row in the result set, and the values in that row are fetched and stored in the host language variables **stud\_id** and **grade**. The **CLOSE** statement closes the cursor. (In this example, only the first row of the result set is retrieved—clearly an artificial situation. The next example is more realistic.)

Each of the SQL statements in the program has a number of options. The general form of the **DECLARE CURSOR** statement is

---

```
DECLARE cursor-name [ INSENSITIVE ] [ SCROLL ] CURSOR FOR
  table-expression
  [ ORDER BY order-item-comma-list ]
  [ FOR { READ ONLY | UPDATE [ OF column-comma-list ] } ]
```

---

where *table-expression* is generally a table, view, or **SELECT** statement.

The option **INSENSITIVE** means that the execution of **OPEN** will effectively create a copy of the rows in the result set and all accesses through the cursor will be to that copy. The SQL standard uses the word “effectively” to mean that the standard

**FIGURE 8.4** Using cursors.

```

#define OK "00000"
EXEC SQL BEGIN DECLARE SECTION;
    unsigned long stud_id;
    char grade[1];
    char *crs_code, *semester;
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
: input values for crs_code, semester, etc.

EXEC SQL DECLARE GETENROLLED INSENSITIVE CURSOR FOR
    SELECT T.StudId, T.Grade
        FROM TRANSCRIPT T
        WHERE T.CrsCode = :crs_code
            AND T.Semester = :semester;

EXEC SQL OPEN GETENROLLED;
if (strcmp(SQLSTATE,OK) != 0) {
    printf("Cannot open cursor\n");
    exit(1);
}
EXEC SQL FETCH GETENROLLED INTO :stud_id, :grade;
if (strcmp(SQLSTATE,OK) != 0){
    printf("Cannot fetch\n");
    exit(1);
}
EXEC SQL CLOSE GETENROLLED;

```

---

does not specify how the INSENSITIVE option must be implemented, but whatever implementation is used must have the same effect as if a separate copy had been made. This type of returned data is sometimes called a **snapshot**.

INSENSITIVE cursors have very intuitive semantics. The selection over the base tables implied by the SELECT statement is performed when OPEN is executed, and the result set is computed and stored. This copy can then be browsed at a later time using the cursor. The situation is shown in Figure 8.5 for the cursor of Figure 8.4, where `semester = 'F1997'` and `crs_code = 'CS315'`.

Because an INSENSITIVE cursor accesses a copy of the result set, any modifications to the base tables by other statements in the same transaction made (not through this cursor) after the cursor has been opened will not be seen through the cursor. For example, the transaction might execute

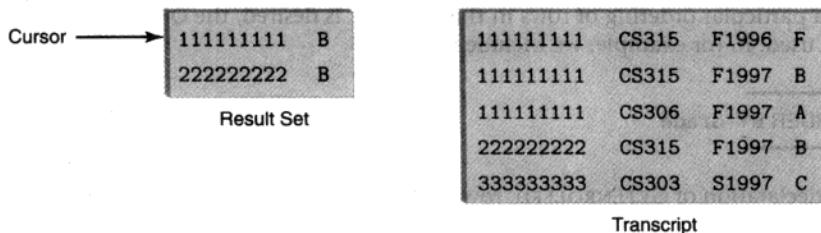
---

```

INSERT INTO TRANSCRIPT
VALUES ('656565656', 'CS315', 'F1997', 'C');

```

---



**FIGURE 8.5** With an insensitive cursor, the result set is effectively calculated when the cursor is opened and the underlying table is not accessed when rows are fetched.

after opening the cursor, thus inserting a new tuple directly (not through the cursor) into TRANSCRIPT. Although crs\_code = 'CS315' and semester = 'F1997' at the time GETENROLLED is opened, the cursor will not retrieve values from the above newly inserted row. This is true even if the transaction executes an UPDATE statement that changes the attributes in one of the tuples in the result set after the cursor has been opened. Similarly, modifications of the base tables by concurrently executing transactions after the cursor has been opened will not be seen through the cursor.

The SQL standard does not specify what effects should be observed when changes are made to the base tables and the INSENSITIVE option has not been selected. Every database vendor is free to implement whatever it deems appropriate. Many vendors use the semantics called KEYSET\_DRIVEN, which is part of ODBC and is described in Section 8.6.

If INSENSITIVE is not specified, the cursor has not been declared READ ONLY, and the SQL query in the cursor declaration satisfies the conditions for an updatable view (see Section 5.3), then the current row of the base table can be updated or deleted through the cursor, and the cursor is said to be "updatable." UPDATE or DELETE statements are used for this purpose, but the WHERE clause is replaced by WHERE CURRENT OF *cursor-name*. Thus, the general syntax is

---

```
UPDATE table-name
SET assignment-comma-list
WHERE CURRENT OF cursor-name
```

---

and

---

```
DELETE
FROM table-name
WHERE CURRENT OF cursor-name
```

---

Because an INSENSITIVE cursor points to a copy of the result set, UPDATE and DELETE statements would have no effect on tables from which the result set was calculated. Hence, to avoid confusion, these operations cannot be performed through an INSENSITIVE cursor.

If a particular ordering of rows in the result set is desired, the ORDER BY clause can be used. If, for example, we include the clause

---

**ORDER BY Grade**

---

in the declaration of GETENROLLED, rows of the result set will be in ascending order of Grade.

The general form of the FETCH statement is

---

**FETCH [ [ row-selector ] FROM ] cursor-name  
INTO target-commalist**

---

where *target-commalist* is a list of host language variables that must match in number and type the list of attributes of the cursor's result set. The *row-selector* determines how the cursor is to be moved over the result set before the next row is fetched. The options are

---

**FIRST  
NEXT  
PRIOR  
LAST  
ABSOLUTE *n*  
RELATIVE *n***

---

If the row selector is NEXT, the cursor is moved to the next row of the result set and that row is fetched into the variables named in *target-commalist*. If the row selector is PRIOR, the cursor is moved to the preceding row and that row is fetched. Similarly, the FIRST row selector causes the cursor to be moved to the first row, and the LAST row selector causes the cursor to be moved to the last row. Finally, ABSOLUTE *n* refers to the *n*th row in the table, and RELATIVE *n* refers to the *n*th row before or after the row to which the cursor is pointing (as determined by a negative or positive *n*). If *row-selector* is omitted, NEXT is assumed. In that case, if the above ORDER BY clause is used in GETENROLLED, rows are fetched in ascending grade order.

The option SCROLL in the declaration of the cursor means that all forms of the FETCH statement are allowable. If SCROLL is not specified, only NEXT is allowable.

In Figure 8.6, we extend the example of Figure 8.4 so that all of the students enrolled in a particular course can be processed. The FETCH statement is now in a loop that terminates when the status returned indicates that execution was unsuccessful. The conditional statement following the loop checks for a "no data" condition (SQLSTATE has value "02000"), indicating that the result set has been completely scanned. It calls an error-handling routine if this is not the case.

**FIGURE 8.6** Using a cursor to scan a table.

```
#define OK "00000"
#define EndOfScan "02000"
EXEC SQL BEGIN DECLARE SECTION;
    unsigned long stud_id;
    char grade[1];
    char *crs_code;
    char *semester;
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE GETENROLLED INSENSITIVE CURSOR FOR
    SELECT T.StudId, T.Grade
        FROM TRANSCRIPT T
        WHERE T.CrsCode = :crs_code
            AND T.Semester = :semester
    FOR READ ONLY;

    : get values for crs_code, semester

EXEC SQL OPEN GETENROLLED;
if (strcmp(SQLSTATE,OK) != 0) {
    printf("Cannot open cursor\n");
    exit(1);
}

EXEC SQL FETCH GETENROLLED INTO :stud_id, :grade;
while (strcmp(SQLSTATE,OK) == 0) {
    : process the values in stud_id and grade
    EXEC SQL FETCH GETENROLLED INTO :stud_id, :grade;
}

if (strcmp(SQLSTATE,EndOfScan) != 0) {
    printf("Something fishy: error before end-of-scan\n");
    exit(1);
}

EXEC SQL CLOSE GETENROLLED;
```

### 8.2.5 Stored Procedures on the Server

Many DBMS vendors allow **stored procedures** to be included as elements of the database schema. These procedures can then be invoked by an application at a client site and executed at the server site. Among the advantages of stored procedures are the following:

- Since the procedure executes at the server, only its results need be transmitted from the server back to the application program. For example, a stored procedure might use a cursor to scan a large result set and analyze the rows to produce a single value that is returned to the application program. By contrast, if the cursor is used from within the application program, the entire result set must be returned to the application program for analysis, thereby increasing communication costs and response time.
- The SQL statements within a stored procedure can be prepared before the application is executed since the procedure is part of the schema stored at the server. By contrast, preparation of embedded SQL statements is generally done at run time. Hence, even if a stored procedure contains only a single SQL statement, that statement will execute more efficiently in the procedure than if it had been embedded directly in the application.

This advantage can become a disadvantage because query plans tend to go stale when there are significant database changes after the preparation has been computed. Thus, “old” stored procedures might avoid the overhead of query preparation but incur run-time overhead due to out-of-date query plans. Some vendors (e.g., Sybase) provide an option, **WITH RECOMPILE**, that can be specified at the time of the procedure call. The application can thus periodically recompile stored procedures and keep query execution plans up to date.

- Authorization can be checked by the DBMS at the level of the stored procedure using the **GRANT EXECUTE** statement, which extends the **GRANT** statement introduced in Section 3.3. Thus, even the users who are not authorized to access particular relations in the database might be authorized to execute certain procedures that contain statements that access those relations. For example, both the registration and grade-changing transactions might invoke stored procedures that use a **SELECT** statement to access the same tuples in a particular table, but one stored procedure can be executed only by students and the other only by faculty.

In addition, a stored procedure can control what the user can do beyond the capabilities of the SQL **GRANT** statement. For example, the code within the stored procedure can enforce the requirement that only the student can execute a transaction to register himself.

- The application programmer need not know the details of the database schema since all database accesses can be encapsulated within the procedure body. For example, the registrar’s office might supply the procedure body for the registration transaction. The application programmer need only know how to call it.

- Maintenance of the system is simplified since only one copy of the procedure, stored on the server, need be maintained and updated. By contrast, if the code contained in a procedure is part of a number of application programs, all of those copies have to be maintained and updated.
- The physical security of the code for the procedure is enhanced because the code is stored on the server rather than with the application program.

The original SQL-92 standard did not support stored procedures, but this support was added retroactively in 1996. We illustrate the language of stored procedures through an example.

Figure 8.7 shows the DDL declaration of a stored procedure for the transaction that deregisters a student from a course. We assume that the application program will connect to the DBMS before calling the procedure and will disconnect after the procedure returns.

The procedure body is written in the **SQL Persistent Stored Modules** language (**SQL/PSM**), as specified by the expanded SQL-92 standard.<sup>1</sup> The standard also provides for stored procedures written in other languages, such as C. Note that in this context SQL/PSM is simply another host language in which SQL statements are embedded. The procedure in the example has three *in* parameters, indicated by the keyword IN, and two *out* parameters, indicated by the keyword OUT. The standard also allows parameters that can be used both ways (INOUT).

The body of the procedure is enclosed in a BEGIN-END block. The option ATOMIC ensures that the entire block executes as a single atomic unit (i.e., it either executes to completion or the partial results of the execution are rolled back). Next follows a series of variable declarations, which are given initial value using the DEFAULT statement (which is optional). Note that neither the parameters nor the host variables (i.e., PSM variables declared within the stored procedure) have the colon (:) prefix. This is because SQL/PSM is a unified language whose compiler understands the host variable declarations, the control statements, and the SQL query and update statements. Our example illustrates the use of the variables both within and outside of the query and update statements. In particular, their value can be changed with the SET clause, and they can be part of arithmetic and string expressions.

PSM is a powerful, full-blown programming language that is well integrated with the rest of SQL. In our brief discussion, we omit many features, such as the looping constructs, the case statement, and cursors. A detailed treatment of PSM and stored procedures appears in [Melton 1997]; here we only touch upon error handling in PSM, which is somewhat different from that in embedded SQL.

Rather than have the program check the variable SQLSTATE after each update statement (or using the WHENEVER statement), in SQL/PSM, condition handlers are declared for different values of SQLSTATE. A condition handler is a program that gets executed when an SQL statement terminates with a value for SQLSTATE that

<sup>1</sup> Other vendors provide similar languages that precede SQL/PSM and differ from it in various ways. Oracle has PL/SQL, Microsoft and Sybase offer Transact-SQL, and Informix has the SPL language.

**FIGURE 8.7** A stored procedure that deregisters a student from a course.

```

CREATE PROCEDURE Deregister( IN  crs_code CHAR(6),
                             IN  semester CHAR(6),
                             IN  student_id INTEGER,
                             OUT status INTEGER,
                             OUT statusMsg CHAR VARYING(100))

BEGIN ATOMIC
    DECLARE message CHAR VARYING(50)
        DEFAULT 'Houston, we have a problem: ';
    DECLARE Success INTEGER DEFAULT 0;
    DECLARE Failure INTEGER DEFAULT -1;

    IF 1 <> (SELECT COUNT(*) FROM CLASS C
               WHERE C.Semester = semester AND C.CrsCode = crs_code)
    THEN
        SET statusMsg = 'Course not offered';
        SET status = Failure;
    ELSE
        BEGIN -- Block limits the scope of error handler
            DECLARE UNDO HANDLER FOR SQLEXCEPTION
                BEGIN
                    SET statusMsg = message || 'cannot delete';
                    SET status = Failure;
                END
            DELETE FROM TRANSCRIPT
                WHERE StudId = student_id
                    AND Semester = semester
                    AND CrsCode = crs_code;
        END
        BEGIN -- Block limits the scope of error handler
            DECLARE UNDO HANDLER FOR SQLEXCEPTION
                BEGIN
                    SET statusMsg = message || 'cannot update';
                    SET status = Failure;
                END
            UPDATE CLASS
                SET Enrollment = (Enrollment - 1)
                WHERE Semester = semester
                    AND CrsCode = crs_code;
        END
        -- Normal termination
        SET status = Success;
        SET statusMsg = 'OK';
    END IF;
END;

```

---

matches one of the values associated with that condition handler. In our case, we have two handlers associated with SQLEXCEPTION, which is a condition that matches any *error code* (an SQLSTATE value that does *not* begin with 00, 01, or 02). Each handler's scope is delimited by a BEGIN/END block, which allows us to associate different handlers with different SQL statements. Both handlers are UNDO handlers, which means that the DBMS will roll back the effects of the stored procedure and exit after the execution of the handler. UNDO handlers can occur only inside BEGIN ATOMIC blocks. If we specify CONTINUE handlers instead, the execution proceeds after the handler has been executed as if no error occurred. We can specify EXIT instead of UNDO, in which case the procedure will exit after the execution of the condition handler but the changes made by the procedure will *not* be rolled back.

In direct (interactive) SQL (and inside another stored procedure), a stored procedure can be executed using the SQL statement

---

```
CALL procedure_name(argument-commalist) ;
```

---

In a host program with embedded SQL, a CALL statement is preceded by EXEC SQL. In that case, the procedure arguments are host language variables preceded by a colon (:). For example, to execute the stored procedure Deregister(), we might use

---

```
EXEC SQL CALL Deregister(:crs_code,:semester,:stud_id);
```

---

where crs\_code, semester, and stud\_id are host variables.

## 8.3 More on Integrity Constraints

A consistent transaction moves the database from an initial to a final state, both of which satisfy all integrity constraints. However, a constraint might be false in an intermediate state during transaction execution; for example, in the case of referential integrity, if the reference to a row is added before the row itself. Similarly, the state produced by the DELETE statement in the procedure Deregister (Figure 8.7) violates the integrity constraint that the number of students listed as enrolled in a course in TRANSCRIPT be equal to the NumEnrolled attribute value for the course in CLASS. If the DBMS checks constraints immediately after each statement is executed, the DELETE would be rejected.

To deal with this situation, SQL allows the application to control the mode of each constraint. If a constraint is in **immediate mode**, it is checked immediately after the execution of any SQL statement in the transaction that might make it false. If it is in **deferred mode**, it is not checked until the transaction requests to commit.

- If constraint checking for a particular constraint is immediate and an SQL statement causes the constraint to become false, the offending SQL statement is rolled back and an appropriate error code is returned through SQLSTATE. The transaction can retrieve the name of the violated constraint from the diagnostics area.

- If constraint checking for a particular constraint is deferred, the constraint is not checked until the transaction requests to commit. If the constraint is found to be false at that time, the transaction is aborted and an appropriate error code returned. Deferred constraint checking is obviously preferable to immediate checking if integrity constraints are violated in intermediate transaction states.

When a constraint is initially defined, it can be specified with options. For example, a table constraint conforms to the rule

---

```
[ CONSTRAINT constraint-name ] CHECK conditional-expression
    [ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
    [ { DEFERRABLE | NOT DEFERRABLE } ]
```

---

The first option gives the initial mode of the constraint. Thus, if INITIALLY DEFERRED is specified, the constraint is checked in the deferred mode until the mode is changed by an explicit SET CONSTRAINTS statement. The second option tells whether or not the constraint can be deferred by a subsequent SET CONSTRAINTS statement. The options NOT DEFERRABLE and INITIALLY DEFERRED are considered contradictory and cannot be specified together.

A DEFERRABLE constraint can be in IMMEDIATE or DEFERRED mode at different times. The mode switch is performed with the following statement:

---

```
SET CONSTRAINTS { constraint-list | ALL } { DEFERRED | IMMEDIATE }
```

---

where *constraint-list* is a list of constraint names, given in CONSTRAINT statements.

## 8.4 Dynamic SQL

With static SQL, an SQL statement to be executed is designed and embedded in the application program at the time the program is written. All of the details of the statement (e.g., whether it is SELECT or INSERT), schema information (e.g., attribute and table names referred to in the statement), and host language variables used as *in* or *out* parameters are known at compile time.

In some applications, not all of this information is known when the program is written. To handle this situation, SQL defines a syntax for including **directives** in a host language program to construct, prepare, and execute an SQL statement. The statement is constructed by the host language portion of the program at run time. The directives are collectively referred to as *dynamic SQL* to distinguish them from static SQL and to indicate that SQL statements can be (dynamically) constructed at run time. Since, as with static SQL, the directives use a syntax that sets them apart from the host language, dynamic SQL is also a statement-level interface. Static and dynamic SQL use the same syntax, so they can be processed by the same precompiler. An application program can include both static and dynamic SQL constructs.

The constructed SQL statement appears in the program as the value of a host language variable of type string and is passed to the DBMS at run time as the

argument of a dynamic SQL directive for preparation. Once prepared, the statement can be executed. As with static SQL, the statement must be constructed in the dialect understood by the target DBMS.

Suppose that, for example, your university has a single student registration system that allows a student to register for any course at any of its campuses. Assume that each campus has its own course database with its own table- and attribute-naming conventions. When a student executes the registration interaction, the application program might construct, at run time, the appropriate SQL statements to perform the registration at the specified campus. For example, it might have string representations of the appropriate SELECT statements for each campus stored in a file. The correct string is retrieved from the file at run time, assigned to a host language variable, and then prepared and executed. Or the program might use a skeleton of an appropriate SQL statement, which was prepared in advance, and then fill in appropriate table and attribute names at run time.

In the above example, there might be some commonality among the schemas and the SQL statements that must be executed to register a student at all campuses. Hence, the application program might know something about the SQL statements that it is executing. As another example, consider an application that monitors a terminal and allows the user to input an arbitrary SQL statement for execution at some database manager. The application now has no advance information about the SQL statements that it is sending to the database but must simply take the string that has been input to a variable and send it to the DBMS for processing. Similarly, consider an application in which a spreadsheet is connected to a database. At run time, the user might specify that the value of a particular entry in the spreadsheet is some expression involving database items that must be retrieved with queries. The queries might be expressed by the user in some graphical notation, but the application translates this notation into SELECT statements. Again, it has no advance information about the SQL statement to be executed and, possibly, none about the schema of the database the statement is accessing.

This lack of information can create a problem since the domains of the *in* and *out* parameters of the SQL statement must be known so that host language variables of the appropriate type can be used for parameter passing. For situations in which this information is not available to the application program at compile time, dynamic SQL provides directives that allow the program to query the DBMS at run time to obtain schema information.

### 8.4.1 Statement Preparation in Dynamic SQL

We illustrate the idea of dynamically constructed SQL statements with the following example:<sup>2</sup>

<sup>2</sup> For readers who need help with C, the function `scanf()` reads user input and puts the result in the variable `column`. The function `sprintf()` substitutes the value of the variable `column` for the format symbol `%s` and puts the result in the variable `my_sql_stmt`. The backslash in the `SELECT` clause indicates that the string continues on the following line.

```
printf("Which column of CLASS would you like to see?");
scanf("%s", column); // get user input (Enrollment or Room)
// Incorporate user input into SQL statement
sprintf(my_sql_stmt,
        "SELECT C.%s FROM CLASS C \
         WHERE C.CrsCode = ? AND C.Semester = ?",
         column);
EXEC SQL PREPARE st1 FROM :my_sql_stmt;
EXEC SQL EXECUTE st1
    INTO :some_string_var
    USING :crs_code, :semester;
```

---

Here, in addition to the fact that the values of CrsCode and Semester are not known at compile time, the exact form of the SELECT statement is also not known at that time since the column to be retrieved by the query depends on what the user inputs at run time. The PREPARE statement sends the query string (in the variable my\_sql\_stmt) to the database manager for preparation and assigns the name st1 to the prepared statement. Note that st1 here is an SQL variable (used only in SQL statements), not a host language variable, so it is not preceded with a colon (:).

The EXECUTE statement causes the statement named st1 to be executed. The string has two *in* parameters marked with ?. The host language variables whose values are to be substituted for these parameters are named in the USING clause. In addition, the host variable to receive the result is named in the INTO clause. The ? marker is called a **dynamic parameter**, or **placeholder**, and can be used in SELECT, INSERT, UPDATE, and DELETE statements. Once prepared, st1 can be executed many times with different host language variables as arguments. The query execution plan created by the PREPARE statement is used for all subsequent executions during the current session.

Note that, just like SELECT INTO, EXECUTE INTO requires that the query result be a single row. If the result has more than one row, a cursor must be used instead of EXECUTE INTO. We describe cursors over dynamic SQL statements in Section 8.4.3.

Parameter passing in dynamic SQL is different from that in static SQL. Placeholders, instead of the names of host language variables, are used in the string to be prepared, and the INTO clause is now attached to the EXECUTE statement instead of the SELECT statement. Why is parameter passing different in this case?

- With static SQL, the names of the host language variables serving as parameters are provided to the precompiler in the WHERE and INTO clauses of the SQL statements. The precompiler parses these clauses at compile time. The variables are described in the compiler's symbol table that is used to translate variable names to addresses (recall that declarations in the DECLARE SECTION are processed by both the precompiler and the host language compiler). The symbol table entries contain the mapping between variable names and addresses plus the type information needed by the precompiler to generate the code for convert-

ing data items from the database representation to these variables and back. This code is executed in the host language program at the time the SQL statement is executed.

- With dynamic SQL, as in the above example, the SQL statement might not be available to the precompiler. Thus, if host language variables to be used as parameters were embedded in the statement, they could not be processed using information contained in the symbol table. To make parameter information available at compile time, it is supplied in one of two ways: through the *SQLDA* mechanism, explained on page 291, and by supplying the input and output variables in the clauses **USING** and **INTO**, as in our example. In the latter case, the precompiler generates the code for fetching and storing the argument values from and to these variables for communication with the DBMS.

Applications should be designed using static SQL whenever possible since dynamic SQL is generally less efficient. The separation of preparation and execution implies added communication and processing costs—although, if the statement is executed multiple times, the added cost can be prorated over the executions because preparation need be done only once. Moreover, this cost can be eliminated in some cases. With certain SQL statement to be executed only once, we can combine preparation and execution using the **EXECUTE IMMEDIATE** directive:

---

```
EXEC SQL EXECUTE IMMEDIATE
  'INSERT INTO TRANSCRIPT '
  || 'VALUES (''656565656'', ''CS315'', ''F1999'', ''C'') ';
```

---

Note the treatment of strings. **INSERT INTO** is part of the dynamic SQL statement, so we are using single quotes to denote strings. Since the statement is long, it is split into two strings, which are concatenated with the usual SQL concatenation operator, **||**. To include a quote symbol in a string, it must be doubled, as in the case of **''656565656''**. This enables the SQL parser to parse the string correctly. In the result, each occurrence of **''** is replaced with a single quote, thus producing a valid SQL statement.

More generally, as with **EXECUTE**, the SQL statement can be constructed in a host language string variable, in which case the **EXECUTE IMMEDIATE** statement takes the form

---

```
EXEC SQL EXECUTE IMMEDIATE :my_sql_stmt;
```

---

Note the **:** prepended to the variable **my\_sql\_stmt**. As before, it indicates that **my\_sql\_stmt** is a host language variable rather than an SQL variable.

**EXECUTE IMMEDIATE** is merely a shortcut that combines the **PREPARE** and **EXECUTE** statements into one and does not preserve the execution plan after the statement has been executed. This shortcut imposes additional syntactic restrictions, some logical and some not. For instance, it does not allow an associated **INTO** clause.

Therefore, the statement to be executed cannot have any *out* parameters (i.e., it cannot retrieve any data) and so cannot be a SELECT statement.

EXECUTE IMMEDIATE also does not allow an associated USING clause, but this is not a serious limitation. The need for USING in the EXECUTE statement comes from the fact that the statement is prepared once, with dynamic *in* parameters marked as ?, and then is executed many times with different arguments. Since EXECUTE IMMEDIATE does preparation and execution in one step (and the prepared statement is not saved for posterity), the special dynamic parameters are not needed. We can simply *plug* the appropriate parameter values into the SQL statement (represented as a string in the host language) using the host language facilities and then pass the fully constructed statement to EXECUTE IMMEDIATE.

As in static SQL, SQLSTATE is used to return the status of PREPARE, EXECUTE, EXECUTE IMMEDIATE, and all other dynamic SQL statements.

#### 8.4.2 Prepared Statements and the Descriptor Area \*

Even though the query in the example of Section 8.4.1 on page 288 is constructed at run time and the name of the output column is not known at compile time, the example is still fairly simple because the application knows that the query target list contains exactly one attribute name and that the WHERE clause has exactly two dynamic parameters. Knowing the number of outputs and inputs thus allows us to use the EXECUTE statement and provide concrete variable names for the USING and INTO clauses. The precompiler can then supply such niceties as automatic format conversion. For example, if the user inputs Enrollment, which is an integer, conversion to the string format is automatic: the precompiler determines that the INTO variable is of type string. Since the DBMS provides, at run time, the type of the value returned by the SELECT statement, the nature of the conversion can be determined at that time.

Suppose now that, at run time, the application allows the user to specify the number of attributes in the target list and the condition in the WHERE clause. In this case we do not know the number of inputs and outputs at design time and will not be able to use the form of the EXECUTE statement described in Section 8.4.1 since we do not know how many variables to supply in the INTO and USING clauses at the time of writing the program.

To deal with this situation, dynamic SQL provides a run-time mechanism that the application program can use to request from the DBMS information describing the parameters of a statement. For example, suppose that an application allows the user to query any 1-tuple relation in the database:<sup>3</sup>

---

```
printf("Which table would you like to inspect?");
scanf("%s", table); // get user input (e.g., Class or Transcript)
// Incorporate user input into SQL statement
```

<sup>3</sup> Our example uses the EXECUTE statement, which can handle only 1-tuple queries. For the general case, a cursor and the FETCH statement are needed. We discuss the use of cursors in dynamic SQL in Section 8.4.3.

---

```
sprintf(my_sql_stmt,
       "SELECT * FROM %s WHERE COUNT(*)=1",
       table);
```

---

This statement has no input parameters but has an indeterminate number of output parameters because the table to be used in the FROM clause is not known in advance. Thus, it is not known how many table attributes (i.e., *out* parameters) are represented by the \* in the SELECT clause. Although the application knows nothing about these parameters, once the statement has been prepared, the DBMS knows all there is to know about them and can provide this information to the application. It does this through a descriptor area. The application first requests that the DBMS allocate a **descriptor area**, sometimes called an **SQLDA**, in which parameter information can be stored. After the statement has been prepared, the application can then request that the DBMS populate the descriptor with the parameter information. For the above example, we populate the descriptor area as follows:

---

```
EXEC SQL PREPARE st FROM :my_sql_stmt;
EXEC SQL ALLOCATE DESCRIPTOR 'st_output' WITH MAX 21;
EXEC SQL DESCRIBE OUTPUT st
    USING SQL DESCRIPTOR 'st_output';
```

---

Here **st\_output** is an SQL variable that references the descriptor area that has been allocated. The **ALLOCATE DESCRIPTOR** statement creates space in the database manager, which must be sufficient to describe at most 21 parameters of the statement (as specified by the **WITH MAX** clause). The descriptor can be thought of as a one-dimensional array with an entry for each parameter, together with a count of the *actual* number of parameters. Each entry has a fixed structure consisting of components that describe a particular parameter, such as its name, type, and value. All fields are initially undefined. The **DESCRIBE** statement causes the DBMS to populate the *i*th entry of the descriptor **st\_output** with meta-information about the *i*th *out* parameter of the prepared statement **st** (which includes name, type, and length of the parameter). It also stores the number of these parameters in the descriptor.

Returning to our example, the application causes the prepared statement **st** to be executed using the directive

---

```
EXEC SQL EXECUTE st
    INTO SQL DESCRIPTOR 'st_output';
```

---

Execution causes the value of the *i*<sup>th</sup> attribute of the row returned to be stored in the *value field* of the *i*<sup>th</sup> entry in **st\_output**. To retrieve the meta-information about the attributes as well as their values, the application can use the **GET DESCRIPTOR** statement. Typically this is done in a loop that inspects each column in the retrieved row, as illustrated in Figure 8.8. In the end, the program calls **DEALLOCATE DESCRIPTOR** to free the space occupied by the descriptor **st\_output**.

**FIGURE 8.8** Example of using GET DESCRIPTOR.

```
int collength, coltype, colcount;
char colname[255];
// Arrange variables for different types of data
char stringdata[1024];
int intdata;
float floatdata;
: variable declarations for other types

// Store the number of columns in colcount
EXEC SQL GET DESCRIPTOR 'st_output' :colcount = COUNT;
for (i=0; i < colcount; i++) {
    // Get meta-information about the ith attribute
    // Note: type is represented by an integer constant, such as
    // SQL_CHAR, SQL_INTEGER, SQL_FLOAT, defined in a header file
    EXEC SQL GET DESCRIPTOR 'st_output' VALUE :i
        :coltype = TYPE,
        :collength = LENGTH,
        :colname = NAME;
    printf("Column %s has value: ", colname);
    switch (coltype) {
        case SQL_CHAR:
            EXEC SQL GET DESCRIPTOR 'st_output' VALUE :i :stringdata = DATA;
            printf("%s\n", stringdata); // print string value
            break;
        case SQL_INTEGER:
            EXEC SQL GET DESCRIPTOR 'st_output' VALUE :i :intdata = DATA;
            printf("%d\n", intdata); // print integer value
            break;
        case SQL_FLOAT:
            EXEC SQL GET DESCRIPTOR 'st_output' VALUE :i :floatdata = DATA;
            printf("%f\n", floatdata); // print floating point value
            break;
        : other cases
    } // switch
} // for loop
```

Situations in which the SQL statement to be executed contains an unknown number of input parameters are rare. When they do occur, the application can use ALLOCATE and DESCRIBE INPUT to set up a descriptor area for the *in* parameters specified as ? placeholders. As before, this descriptor is essentially an array with an entry for each placeholder. The application then uses the DESCRIBE INPUT statement

to request that the DBMS populate the descriptor area with the information about each *in* parameter: for example, its type, length, and name. In this case the value has to be supplied by the application using the SET DESCRIPTOR statement. We do not discuss the details of this procedure and refer the reader to SQL manuals such as [Date and Darwen 1997; Melton and Simon 1992].

### 8.4.3 Cursors

Like SELECT INTO, EXECUTE INTO has the problem that the result of the query must be a single tuple. A more likely situation is that the result of a query is a relation and the cursor mechanism is needed to scan it. Fortunately, cursors can be defined for prepared statements in dynamic SQL as they are in static SQL, although the syntax is slightly different. For instance, the following is equivalent to the program in Figure 8.4 in static SQL (for brevity we have not shown the status checks in this case):

---

```
my_sql_stmt = "SELECT T.StudId, T.Grade \
               FROM Transcript T \
               WHERE T.CrsCode = ? \
               AND T.Semester = ?";
EXEC SQL PREPARE st2 FROM :my_sql_stmt;
EXEC SQL DECLARE GETENROLL INSENSITIVE CURSOR FOR st2;
EXEC SQL OPEN GETENROLL USING :crs_code, :semester;
EXEC SQL FETCH GETENROLL INTO :stud_id, :grade;
EXEC SQL CLOSE GETENROLL;
```

---

As with static SQL, the DECLARE CURSOR statement has the options INSENSITIVE and SCROLL, while the FETCH statement has the option that allows a row selector (FIRST, NEXT, PRIOR, etc.) to be specified for scrollable cursors. UPDATE and DELETE statements can also be performed through a dynamic cursor that is not insensitive.

### 8.4.4 Stored Procedures on the Server

Some DBMSs allow stored procedures to be called using dynamic SQL. To call the stored procedure of Figure 8.7, we might use

---

```
my_sql_stmt = "CALL Deregister(?, ?, ?)";
EXEC SQL PREPARE st3 FROM :my_sql_stmt;
EXEC SQL EXECUTE st3
    USING :crs_code, :semester, :stud_id;
```

---

The PREPARE statement prepares the call to the *Deregister* procedure. The EXEC SQL EXECUTE statement calls the procedure and supplies the values of host language variables as arguments.

## 8.5 JDBC and SQLJ

JDBC<sup>4</sup> is an API to the database manager that provides a call-level interface for the execution of SQL statements from a Java language program. As in dynamic SQL, an SQL statement can be constructed at run time as the value of a string variable. JDBC was developed by Sun Microsystems and is an integral part of the Java language.

In contrast, SQLJ is a statement-level interface to Java, analogous to static embedded SQL. Unlike JDBC, it was developed by a consortium of companies and has become a separate ANSI standard. In the end, both JDBC and SQLJ were included as part of SQL:2003.

Both JDBC and SQLJ are designed to access databases over the Internet and are much more portable than the various implementations of embedded and dynamic SQL. As part of SQL:2003, JDBC and SQLJ are described in the document titled *Object Language Bindings*.

### 8.5.1 JDBC Basics

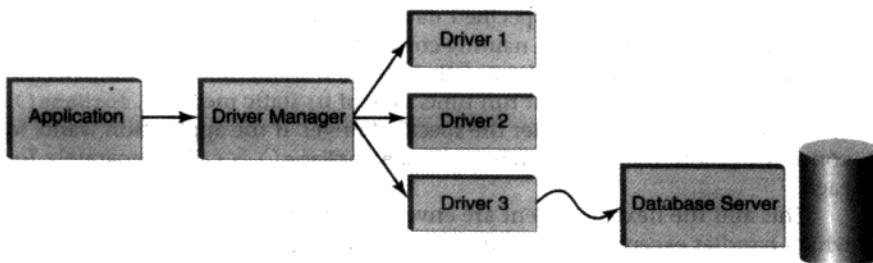
Recall that, in dynamic and static SQL, the target DBMS must be known at compile time since the SQL statements must use the target's dialect. With dynamic SQL, the schema need not be known at compile time. By contrast, in JDBC neither the DBMS nor the schema need be known at compile time. Applications can use core SQL dialect that all JDBC drivers are required to support, and, as with dynamic SQL, JDBC supplies features that allow the application to request information about the schema from the DBMS at run time.

The mechanism that allows JDBC programs to deal with different DBMSs at run time is shown in Figure 8.9. The application communicates with a DBMS through a JDBC module called a **driver manager**. When the application first connects to a particular DBMS, the driver manager chooses another JDBC module, called a **driver**, to pass SQL statements specified in the JDBC calls to that DBMS. JDBC maintains a separate driver for most commonly used DBMSs. The manager chooses the one corresponding to the particular DBMS being accessed. When an SQL statement is to be executed, the application program sends a string representation of the statement to the appropriate driver, which performs any necessary reformatting and then sends the statement to the DBMS, where it is prepared and executed.

The software architecture of JDBC consists of a set of predefined object classes, such as `DriverManager` and `Statement`. The methods of these object classes provide the call-level interface to the database. A JDBC program must first load these predefined object classes, then create appropriate instances of the object types, and then use the appropriate methods to access the database. The general structure of JDBC calls within a Java program is shown in Figure 8.10. We explain each statement as it appears:

- `import java.sql.*` imports all of the classes in the package `java.sql` and hence makes the JDBC API available within the Java program. The classes

<sup>4</sup> JDBC is a trademark of Sun Microsystems, which claims that it is not an acronym. Nevertheless, it is often assumed to stand for "Java DataBase Connectivity."



**FIGURE 8.9** Connecting to a database through JDBC.

**FIGURE 8.10** Skeleton of procedure calls needed for JDBC.

```

import java.sql.*;
.

String url,userId,password;
Connection con1;
.

try {
    // Use the right driver for your database
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // load the driver
    con1 = DriverManager.getConnection(url, userId, password);
} catch (ClassNotFoundException e) {
    System.err.println("Cannot load driver\n");
    System.exit(1);
} catch (SQLException e) {
    System.err.println("Cannot connect\n");
    System.exit(1);
}
Statement stat1 = con1.createStatement(); // create a statement object
String myQuery = ... some SELECT statement ...
ResultSet res1 = stat1.executeQuery(myQuery);
    : process results
stat1.close(); // free up the statement object
con1.close(); // close connection
  
```

Connection, Statement, DriverManager, and ResultSet that occur in the figure are all in this package, as are PreparedStatement, CallableStatement, ResultSetMetadata, and SQLException. (CallableStatement is a subclass of PreparedStatement, which in turn is a subclass of Statement.)

- `Class.forName()` loads the specified database driver and registers it with the driver manager. Although the naming convention might not be intuitive, there is a class in the `java.lang` package, called `Class`, with methods that allow a class to be loaded into a program at run time. One of its static methods, `forName()`, can be used to load and register the specified driver. If the application wants to connect to more than one database, `Class.forName()` is called separately for each.

This and the next statement are enveloped with the `try/catch` construct, which handles exceptions. We return to exception handling in Section 8.5.5.

- `DriverManager.getConnection()` uses the static method `getConnection()` of the class `DriverManager` to connect to the DBMS at the given address. The method tests each of the database drivers that have been loaded to see if any are capable of establishing a connection to that DBMS. If so, it
  1. Establishes the connection using the specified user Id and password
  2. Creates a `Connection` object and assigns it to the variable `con1`, declared earlier

The parameter `url` contains the URL (uniform resource locator) of the target DBMS. This URL is obtained from the database administrator and looks something like

---

`jdbc:odbc:http://server.xyz.edu/sturegDB:8000`

---

The prefix `jdbc` specifies the main protocol for connecting to the database (JDBC, not surprisingly). The second component, `odbc`, specifies the subprotocol (which is vendor and driver specific). Next comes the address of the database server, followed by the communication port number on which the server is listening.

- `con1.createStatement()` uses the `createStatement()` method of the `Connection` object `con1` to create a `Statement` object and assign it to the `Statement` variable `stat1`.
- `stat1.executeQuery()` prepares and executes the `SELECT` statement provided as an argument, using the `Statement` object `stat1`. The SQL statement can have no *in* parameters, and the result set it returns is stored in the `ResultSet` object `res1`, which is created by the `executeQuery()` method. This method is analogous to the `EXECUTE IMMEDIATE` directive in dynamic SQL since it combines both preparation and execution. A major difference is that the JDBC method creates an object for returning data, which `EXECUTE IMMEDIATE` cannot do. We discuss the `ResultSet` class in Section 8.5.3.

To execute an `UPDATE`, `DELETE`, or `INSERT` statement (or a DDL statement), the appropriate form is

---

`stat1.executeUpdate(... some SQL statement ...);`

---

This function returns an integer denoting how many rows are affected (or 0 for a DDL statement, such as CREATE).

- `stmt.close()` and `con1.close()` deallocate the `Statement` object and close the connection (deallocating the `Connection` object), respectively. After a statement has been executed, the `Statement` object that supported it need not be closed but can be reused to support another statement.

### 8.5.2 Prepared Statements

The call to `executeQuery()` in Figure 8.10 both prepares and executes the specified statement. To prepare a statement and then execute it separately, the appropriate calls are

---

```
PreparedStatement ps1 =
    con1.prepareStatement(... SQL preparable statement ...);
```

---

which returns a `PreparedStatement` object that is assigned to `ps1`, followed by either

---

```
ResultSet res1 = ps1.executeQuery();
```

---

or in the case of an update statement,

---

```
int n = ps1.executeUpdate();
```

---

where `executeUpdate()` returns an integer denoting how many rows were updated.

`PreparedStatement` is a subclass of the class `Statement`. Note that both classes have methods with the names `executeQuery()` and `executeUpdate()`, but that the methods for `PreparedStatement` have no arguments (because a prepared statement knows which query it is).

As in dynamic SQL, the string argument of `prepareStatement()` can contain dynamic *in* parameters marked with the ? placeholders. Also as with dynamic SQL, the placeholders must be given concrete values before execution. This is done using the `setXXX()` methods. For instance,

---

```
ps1.setInt(1, someIntVar);
```

---

replaces the first ? placeholder with the value stored in the host language variable `someIntVar`. `PreparedStatement` has a number of `setXXX()` methods, where `XXX` specifies the type (e.g., `Int`, `Long`, `String`) to supply *in* arguments of different types.

### 8.5.3 Result Sets and Cursors

The execution of a query statement stores its result (the *out* arguments) in the specified `ResultSet` object. The rows in that result set are retrieved using a cursor.

**FIGURE 8.11** Fragment of JDBC program using a cursor.

```

import java.sql.*;
.
.
Connection con2;
try {
    // Use appropriate URL and JDBC driver
    String url = "jdbc:odbc:http://server.xyz.edu/sturegDB:800";
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    con2 = DriverManager.getConnection(url, "pml", "36.ty");
} catch ... // catch exceptions

// Use the "+" operator, for readability
String query2 = "SELECT T.StudId, T.Grade " +
    "FROM TRANSCRIPT T " +
    "WHERE T.CrsCode = ? " +
    "AND T.Semester = ?";

PreparedStatement ps2 = con2.prepareStatement(query2);
ps2.setString(1, "CS308");
ps2.setString(2, "F2000");
ResultSet res2 = ps2.executeQuery();

long studId;
String grade;
while (res2.next()) {
    studId = res2.getLong("StudId");
    grade = res2.getString("Grade");
    : process the values in studId and grade
}

ps2.close();
con2.close();

```

---

A JDBC cursor is implemented using the `next()` method of class `ResultSet`. When invoked on a `ResultSet` object, it scans the entire set tuple by tuple. In Figure 8.11, the result set object is stored in the variable `res2`, and `res2.next()` advances the cursor to the next row.

The program prepares the query, supplies arguments, and then executes the query. It then uses a while loop to retrieve all of the rows of the result set. The method `next()` moves the cursor and returns `false` when there are no more rows to return. In each iteration of the while loop, the result tuples are retrieved using calls to `getLong()` and `getString()`, which come in two forms: those that take attribute names as a parameter and those that take positional arguments. Thus, we can use

`getLong(1)` to obtain the student Id (since `StudId` is the first attribute in the result set `res2`) and `getString(2)` to obtain the grade (since `Grade` is the second attribute). The class `ResultSet` has `getXXX()` methods for all primitive types supported by Java.

JDBC defines three result set types. They differ in their support of scrolling and sensitivity.

1. A *forward-only* result set, as its name implies, is not scrollable (the cursor can move only in the forward direction). It uses the default cursor type (`INSENSITIVE` or non-`INSENSITIVE`) of the underlying DBMS.
2. A *scroll-insensitive* result set is scrollable and uses an `INSENSITIVE` cursor of the specified DBMS so that changes made to the underlying tables after the result set is computed (either by the transaction that created the result set or by other transactions) are not seen in the result set.
3. A *scroll-sensitive* result set is scrollable and uses a non-`INSENSITIVE` cursor. As we discussed in Section 8.2.4, the SQL standard does not define any required behavior when the `INSENSITIVE` option has not been selected. Database vendors are free to implement whatever semantics they deem appropriate, and JDBC generally provides the semantics supported in the DBMS it is accessing. Many vendors have implemented the semantics called `KEYSET_DRIVEN`, which is part of the ODBC specification and is described in Section 8.6. In that semantics, row updates and deletes made after the result set is created are visible but inserts are not. JDBC provides a variety of methods for querying the driver to determine what to expect.

If the target DBMS does not support the scrolling or sensitivity properties requested by an application, a warning is issued.

A result set can be read-only or updatable. With an updatable result set, the SQL query on which the result set is based must satisfy the conditions for updatable views (see Section 5.3). For example, the following variant of `createStatement()` creates an instance `s3` of the class `Statement`:

---

```
Statement s3 =
    con1.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                        ResultSet.CONCUR_UPDATABLE);
```

---

If the `executeQuery()` method of `s3` is later invoked, the result set that will be created will be updatable and scroll-sensitive. Consult the description of classes `ResultSet` and `Connection` in your JDK documentation to see other options.

The current row of an updatable result set, `res`, produced by a `SELECT` statement that returns the value of a string attribute, `Name`, might be updated by assigning the value `Smith` to `Name` using

---

```
res.updateString("Name", "Smith");
```

---

As with the methods `setXXX()` and `getXXX()`, there is an `updateXXX()` method for every primitive type.

When the new value of the row has been completely constructed, the underlying table is updated by the execution of

---

```
res.updateRow();
```

---

Not only can rows in an updatable result set be updated and deleted but, in contrast to cursors in static and dynamic SQL, new rows can be inserted through the result set as well. The column values of the row to be inserted are first assembled in a buffer associated with the result set. The method `res.insertRow()` is then called to insert the buffered row in the result set `res` and in the database simultaneously.

#### 8.5.4 Obtaining Information about a Result Set

As with dynamic SQL, information about a result set might not be known when the program is written. JDBC provides mechanisms for querying the DBMS to obtain such information. For example, JDBC provides a class `ResultSetMetaData`, whose methods can be used for this purpose. Thus

---

```
ResultSet rs3 = stmt3.executeQuery("SELECT * FROM TABLE3");
ResultSetMetaData rsm3 = rs3.getMetaData();
```

---

creates a `ResultSetMetaData` object, `rsm3`, and populates it with information about the result set `rs3`. This object can then be queried with such methods as

---

```
int numberOfColumns = rsm3.getColumnCount();
String columnName = rsm3.getColumnName(1);
String typeName = rsm3.getColumnTypeName(1);
```

---

The first method returns the number of columns in the result set, and the last two return the name and type of column 1. Using these methods, even without knowing the schema of a result set, one can iterate over the columns of each row in a loop, examine their types, and fetch the data. For instance, if `rsm3.getColumnTypeName(2)` returns "Integer", the program can call `rs3.getInt(2)` to obtain the value stored in the second column of the current row.

JDBC also has a class, `DatabaseMetaData`, which can be queried for information about the schema and other database information.

#### 8.5.5 Status Processing

Status processing in JDBC uses the standard exception-handling mechanism of Java. The basic format is

---

```
try {
    : code that might cause an exception goes here
}
```

```
    catch (SQLException e) {
        System.out.println("Bad things have happened:\n");
        System.out.println("Message: " + e.getMessage());
        System.out.println("SQLState: " + e.getSQLState());
        System.out.println("ErrorCode: " + e.getErrorCode());
    };


---


```

In fact, in our examples, all calls to `executeQuery()`, `prepareStatement()`, and the like should have been enveloped with such a `try` statement.

The system *tries* to execute the statements within the `try` clause. Each such statement can contain method calls for Java or JDBC objects, and the declaration of each method can specify that, if certain errors occur during method execution, one or more named exceptions are **thrown**, where the name of the exception denotes the type of error that occurred. For example, the JDBC method `executeQuery()` throws the exception `SQLException` if the DBMS returns an access error during query execution. An access error occurs whenever there is an unsuccessful or incomplete execution of an SQL statement—more precisely, an execution for which `SQLSTATE` has any value other than successful completion (of the form "00XXX"), warning (of the form "01XXX"), or no data ("02000").

If such an exception is thrown within the `try` clause, it is **caught** by the corresponding `catch` clause, which is then executed. In the example, an `SQLException` object, `e`, is created, whose methods can be used to print out an error message, return the value of `SQLSTATE`, or return any vendor-specific error code. When the `catch` clause completes, execution continues with the next statement following the clause.

## 8.5.6 Executing Transactions

By default, the database is in **autocommit mode** when a connection is created. Each SQL statement is treated as a separate transaction, which is committed when that statement is (successfully) completed. To allow two or more statements to be grouped into a transaction, autocommit mode is disabled using

---

```
con4.setAutoCommit(false);
```

---

where `con4` is a `Connection` object.

Initially, each transaction uses the default isolation level of the database manager. The level can be changed with a call such as

---

```
con4.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

---

Serialization levels `TRANSACTION_SERIALIZABLE`, `TRANSACTION_REPEATABLE_READ`, and the like are constants (static integers) defined in class `Connection`.

Transactions can be committed or aborted using the `commit()` or `rollback()` methods of class `Connection`.

---

```
con4.commit();
con4.rollback();
```

---

After a transaction is committed or rolled back, a new one starts when the next SQL statement is executed (or, in the case of the first SQL statement in the program, when that statement is executed). This way of structuring transactions is called *chaining*.

If the program is connected to more than one DBMS, the transactions at each can be separately committed or rolled back. JDBC does not support a commit protocol that ensures that the set of transactions will be globally atomic. However, a Java package, JTS (Java Transaction Service), includes a TP monitor (and an appropriate API, called JTA [Java Transaction API]), which does guarantee an atomic commit of distributed transactions using JDBC. Also J2EE (Java 2 Enterprise Edition) provides transaction services based on JDBC and JTS.

### 8.5.7 Stored Procedures on the Server

JDBC can be used to call a stored procedure if the DBMS supports this feature. For example, to call the stored procedure defined in Figure 8.7 on page 284 we might use the program fragment

---

```
CallableStatement cs5 =
    con5.prepareCall("{call Deregister(?,?,?,?,?)}");
cs5.setString(1, crs_code);
cs5.setString(2, semester);
cs5.setInt(3, stud_id);
cs5.getInt(4, status);
cs5.setString(5, message);
cs5.executeUpdate();
```

---

where `con5` is a `Connection` object.

The first statement declares a `CallableStatement` object with the name `cs5` and assigns to it the stored procedure `Deregister()`. The braces around the construct `{call Deregister(?,?,?,?,?)}` denote that the construct is part of the **SQL escape syntax** and signals the driver that the code within the braces should be handled in a special way. The values of the three *in* parameters of `Deregister()`, obtained from the Java variables `crs_code`, `semester`, and `stud_Id`, are specified

with `setXXX()` method calls. (We omit the details of *out* parameters and return values, which are handled somewhat differently.) The final statement executes the call.

The DBMS might allow a stored procedure to return a result set, perhaps in addition to updating the database. A call to such a procedure is viewed as a query, in which case the last statement in the above program fragment is replaced by

---

```
ResultSet rs5 = cs5.executeQuery();
```

---

JDBC also has facilities for creating a stored procedure (as a string) and sending it to the DBMS.

### 8.5.8 An Example

Figure 8.12 is a fragment of a Java program containing calls to the JDBC API. The program performs roughly the same transaction as that of Figure 8.3, except that, for simplicity, it uses constants in the SQL statements instead of the ? placeholders.

### 8.5.9 SQLJ: Statement-Level Interface to Java

Although call-level interfaces, such as JDBC, can be used in static transaction processing applications (where the database schema and the format of the SQL statements are known at compile time), they are fundamentally less efficient at run time than statement-level interfaces, such as static SQL, because preparation and execution generally involve separate communication with the DBMS. For this reason, a consortium of companies developed a statement-level SQL interface to Java, called SQLJ, which is now an ANSI standard. An important goal of SQLJ is to obtain some of the run-time efficiency of embedded SQL for (static) Java applications while retaining the advantage of accessing DBMSs through JDBC.

SQLJ is analogous to embedded SQL but was designed specifically to be embedded in Java programs. Such programs are translated by a precompiler into standard Java, and the embedded SQLJ constructs are replaced by calls to an SQLJ run-time package, which accesses a database using calls to a JDBC driver. An SQLJ program can connect to multiple DBMSs using different JDBC drivers in this way. As with embedded SQL, the precompiler can also check SQL syntax and the number and types of arguments and results.

We do not discuss the syntax of SQLJ in detail but highlight some of the differences between SQLJ, embedded SQL, and JDBC:

- In contrast to embedded SQL, in which each DBMS vendor supports its own proprietary version of SQL, SQLJ supports a core sublanguage of SQL-92 and is much more portable across vendors. (DBMS vendors can provide proprietary extensions, however.)
- SQL statements appear in a Java program as part of an **SQLJ clause**, which begins with `#SQL` (instead of `EXEC SQL`, as in embedded SQL) and can contain an SQL

FIGURE 8.12 A fragment of a Java program using JDBC.

```
import java.sql.*;
.
.
try {
    // Use the right JDBC driver here
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch (ClassNotFoundException e) {
    return(-1); // Cannot load driver
}
Connection con6 = null;
try {
    String url = "jdbc:odbc:http://server.xyz.edu/sturegDB:8000";
    con6 = DriverManager.getConnection(url,"john","ji21");
} catch (SQLException e) {
    return(-2); // Cannot connect
}
con6.setAutoCommit(false);
Statement stat6 = con6.createStatement();
try {
    stat6.executeUpdate("DELETE FROM TRANSCRIPT " +
                        "WHERE StudId = 123456789 " +
                        "AND Semester = 'F2000' " +
                        "AND CrsCode = 'CS308'" );
} catch (SQLException e) {
    con6.rollback();
    stat6.close();
    con6.close();
    return(-3); // Cannot execute
}
try {
    stat6.executeUpdate("UPDATE CLASS " +
                        "SET Enrollment = (Enrollment - 1) " +
                        "AND Semester = 'F2000' " +
                        "WHERE CrsCode = 'CS308'" );
} catch (SQLException e) {
    con6.rollback();
    stat6.close();
    con6.close();
    return(-4); // Cannot update
}

con6.commit();
stat6.close();
con6.close();
return(0); // Success!
```

**FIGURE 8.13** Use of an iterator in SQLJ.

```

import java.sql.*;
.

#SQL iterator GetEnrolledIter(int studentId, String studGrade);
GetEnrolledIter iter1;

#SQL iter1 = { SELECT T.StudId AS "studentId",
               T.Grade AS "studGrade"
             FROM TRANSCRIPT T
             WHERE T.CrsCode = :crsCode
                   AND T.Semester = :semester };

int id;
String grade;
while (iter1.next()) {
    id = iter1.studentId();
    grade = iter1.studGrade();
    :
    process the values in id and grade
}
iter1.close();

```

---

statement inside curly braces. For example, the **SELECT** statement of Figure 8.1 becomes in SQLJ:

---

```

#SQL {SELECT C.Enrollment
      INTO :numEnrolled
      FROM CLASS C
      WHERE C.CrsCode = :crsCode
            AND C.Semester = :semester};

```

---

- Any Java variable can be included as a parameter in an SQL statement prefixed with `:`, as in static SQL. This method of passing parameters into an SQL construct (which, you will recall, is done at compile time) is considerably more efficient during run time than the method used in JDBC, in which the value of each argument must be bound to a `?` parameter at run time.
- In SQLJ, a query returns an **SQLJ iterator** object instead of a **ResultSet** object. SQLJ iterators are similar to result sets in that they provide a cursor mechanism. In fact, both the SQLJ iterator object and the **ResultSet** object implement the same Java interface `java.util.Iterator`. (SQLJ iterators can be converted into result sets, and vice versa.) An iterator object stores an entire result set and provides methods, such as `next()`, to scan through the rows in the set. Figure 8.13 shows an SQLJ version of the program fragment in Figure 8.6.

The first statement in the figure tells the SQLJ preprocessor to generate Java statements that define a class, `GetEnrolledIter`, which implements the interface `sqlj.runtime.NamedIterator`. This is an interface that extends the standard Java interface, `java.util.Iterator`, and provides the venerable `next()` method. The class `GetEnrolledIter` can be used to store result sets in which each row has two columns: an integer and a string. The declaration gives a Java name to these columns, `studentId` and `studGrade`, and (implicitly) defines the column accessor methods, `studentId()` and `studGrade()`, which can be used to return data stored in the corresponding columns.

The second statement declares an object, `iter1`, in the class `GetEnrolledIter`.

The third statement executes `SELECT` and places the result set in `iter1`. Note that the `AS` clause is used to associate the SQL attribute names in the result set with the column names in the iterator. These names do not have to be the same, but the sequence of columns in the result set and the iterator must correspond in number and type.

The `while` statement fetches the results one at a time into the host variables `id` and `grade` and processes them.

- SQLJ has its own mechanism, which we do not discuss, for defining connection objects and for connecting to a database. A program can have several such connections active at the same time. Unlike in embedded SQL, each individual SQLJ statement can optionally designate a specific database connection to which that clause is to be applied. For example, the `SELECT` statement on page 305 can be rewritten as

---

```
#SQL [db1] {SELECT C.Enrollment
    INTO :num_enrolled
    FROM CLASS C
    WHERE C.CrsCode = :crs_code
    AND C.Semester = :semester};
```

---

to specify that it is to be applied to the (previously defined) database connection named `db1`. If this option is not used, all SQL statements are applied to a default database connection, as in embedded SQL. Recall that, in JDBC, each SQL statement is always associated with a specific database connection *explicitly*, through its `Statement` object. In contrast, in embedded SQL, connection is set in a rather awkward and inflexible way via the `SET CONNECTION` statement.

- Just as static and dynamic embedded SQL statements can be included in the same host language program, SQLJ statements and JDBC calls can be included in the same Java program.

## 8.6 ODBC\*

ODBC (Open DataBase Connectivity) is an API to the DBMS that provides a call-level interface for SQL statement execution. Our presentation is based on the ODBC specification developed by Microsoft, but be aware that some vendors do not support all of the features.

It should also be noted that the SQL standardization body has for quite a long time been working on a specification for a call-level interface, known as SQL/CLI, to replace the bulky dynamic SQL. ODBC is a branch of an earlier version of this specification, and it has much in common with the recently finalized release of SQL/CLI, which is included in SQL:1999. Microsoft has pledged to align ODBC with this newly adopted standard.

The software architecture of an ODBC application is similar to that of JDBC in that it uses a driver manager and a separate driver for each DBMS to be accessed. ODBC is not object oriented, however, and its interface to the DBMS is at a much lower level. For example, in ODBC an application must specifically allocate and deallocate the storage it needs within the driver manager and the driver, whereas in JDBC that storage is automatically allocated when the appropriate objects are created, and deallocated when these objects are no longer needed. Thus, before an ODBC application calls the function `SQLConnect()` to request a connection to a database manager, it must first call the function `SQLAllocConnect()` to request that the driver manager allocate storage for that connection. Later, after the application calls `SQLDisconnect()` to disconnect from the database manager, it must call `SQLFreeConnect()` to deallocate this storage. As a result, ODBC applications are prone to **memory leaks**—an accumulation of garbage memory blocks that occurs when a program fails to free up ODBC structures that are no longer in use.

Figure 8.14 shows the structure of one version of the required function calls as they might appear in a C program.<sup>5</sup> Each function returns a value that denotes success or failure.

- `SQLAllocEnv()` allocates and initializes storage within the driver manager for use as ODBC's interface to the application. It returns an identifying *handle*, *henv*. A **handle** is simply a mechanism the application can use to refer to this data structure. In C, a handle is implemented as a pointer, but other host languages might implement it differently. The environment area is used internally by the ODBC driver manager to store run-time information.
- `SQLAllocConnect()` allocates memory within the driver manager for the connection and returns a connection handle, *hdbc*.

<sup>5</sup> We have simplified the syntax in this and subsequent examples to emphasize the semantics of the ODBC interaction. For example, in reality, string parameters, such as `database_name`, are passed with an accompanying length field.

**FIGURE 8.14** Skeleton of procedure calls needed for ODBC in a C program.

```
SQLAllocEnv(&henv);
SQLAllocConnect(henv, &hdbc);
SQLConnect(hdbc, database_name, userId, password);
SQLAllocStmt(hdbc, &hstmt);
SQLExecDirect(hstmt, ... SQL statement ...);
: process results
SQLFreeStmt(hstmt, fOption);
SQLDisconnect(hdbc);
SQLFreeConnect(hdbc);
SQLFreeEnv(henv);
```

- `SQLConnect()` loads the appropriate database driver and then connects to the DBMS server using previously allocated connection, `hdbc`, the database name, user Id, and password.  
If the application wants to connect to more than one database manager, `SQLAllocConnect()` and `SQLConnect()` are called separately for each manager. The drivers (there might be more than one in this case) maintain separate transactions for each such connection.
- `SQLAllocStmt()` allocates storage within the driver for an SQL statement and returns a handle, `hstmt`, for that statement.
- `SQLExecDirect()` takes a statement handle previously allocated using `SQLAllocStmt()` and a string variable containing an SQL statement and asks the DBMS to prepare and execute the statement. The same handle can be used multiple times to execute different SQL statements.  
The SQL statement can be a data-manipulation statement, such as `SELECT` or `UPDATE`, as well as a DDL statement, such as `CREATE` or `GRANT`. It cannot contain an embedded reference to a host variable, since variable names can be translated to memory addresses only at compile time. Note that `SQLExecDirect()` is related to but is more versatile than `EXECUTE IMMEDIATE` in dynamic SQL. Whereas `EXECUTE IMMEDIATE` cannot return data to the application (see Section 8.4.1), `SQLExecDirect()` can produce a result set using a `SELECT` statement, and this set can then be accessed through a cursor (see Section 8.6.2).
- `SQLDisconnect()` disconnects from the server. This function takes the connection handle as an argument.
- `SQLFreeStmt()`, `SQLFreeConnect()`, and `SQLFreeEnv()` release the handles and free up the corresponding storage space allocated by the corresponding `Alloc` functions.

### 8.6.1 Prepared Statements

Instead of calling `SQLExecDirect()`, the application program can call

---

```
SQLPrepare(hstmt, ... SQL statement ...);
```

---

to prepare the statement and then

---

```
SQLExecute(hstmt);
```

---

to execute it.

As in dynamic SQL, the statement argument of `SQLPrepare()` can contain the ? placeholders. Arguments can be supplied to those parameters using the `SQLBindParameters()` function. For example, the call

---

```
SQLBindParameters(hstmt, 1, SQL_PARAMETER_INPUT,
                  SQL_C_SSHORT, SQL_SMALLINT, &int1);
```

---

binds the first parameter of the statement `hstmt`, which is an *in* parameter, to the host language variable `int1`, where `int1` is of type `short` in the C language. The parameter's value replaces the first ? placeholder in `hstmt` when `SQLBindParameters()` is executed. `SQLBindParameters()` performs the conversion from type `short` in C to type `SMLLINT` in SQL.<sup>6</sup> Since the procedure call is compiled by the host language compiler, references to host language variable `int1` in the parameter list can be resolved at compile time. This contrasts with the use of `SQLExecDirect()`, where references to host variables are not allowed.

### 8.6.2 Cursors

The execution of a `SELECT` statement using `SQLExecDirect()` or `SQLExecute()` does not return any data to the application. Instead, data is returned through a cursor, which is maintained within the ODBC driver and referred to by the statement handle, `hstmt`, returned by `SQLAllocStmt()`. Additional ODBC functions must be called to bring the data into the application.

The program can optionally call `SQLBindCol()`, which binds a particular column of the result set to a specific host variable in the program. For example, we might use the following call to bind the first column to the integer variable `int2`.

---

```
SQLBindCol(hstmt, 1, SQL_C_SSHORT, &int2);
```

---

<sup>6</sup> In a number of ODBC constructs, as in this one, the programmer must specify the desired conversion between the SQL data types used within the DBMS and the C language data types used within the application program.

When `SQLFetch(hstmt)` is then called, the cursor is advanced and the values in the columns of the current row that had been bound to host variables are stored in those variables. If a column has not been bound to a designated host variable, the program can call `SQLGetData()` to retrieve and store its value. For example, to store the value of the second column of the current row in the integer variable `int3` we might use the call

---

```
SQLGetData(hstmt, 2, SQL_C_SSHORT, &int3);
```

---

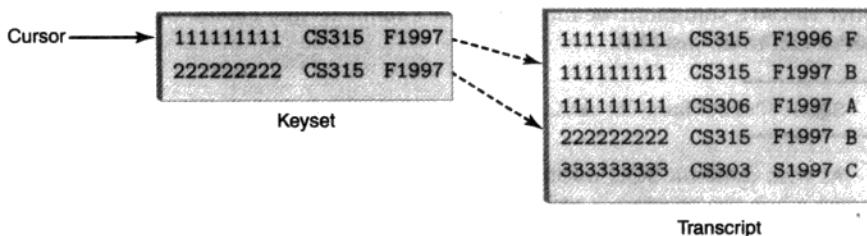
Before executing the `SELECT` statement, the application can use `SQLSetStmtOption()` to specify one of the following three types for the cursor:

1. **STATIC.** As with the `INSENSITIVE` option for cursors in embedded SQL, when the `SELECT` statement is executed, the driver effectively calculates the result set and stores it separately from the base table. A call to `SQLFetch()` fetches a row from the result set. The situation is shown in Figure 8.5, page 279. This type of returned data is called a *snapshot*.
2. **KEYSET\_DRIVEN.** When the `SELECT` statement is executed, the driver effectively constructs a set of pointers to the rows in the base table that satisfy the `WHERE` clause. A call to `SQLFetch()` follows the pointer and fetches the row from the base table. The situation is shown in Figure 8.15. Note the difference between this figure and Figure 8.5, where the result set is calculated at the time the `SELECT` statement is executed and rows are fetched from the result set. This type of returned data is sometimes called a *dynaset*.

Since rows are obtained from the base table, any change to the base table will be seen through the pointers. For example, if a statement in this or a concurrent transaction *modifies* or *deletes* a row that is the target of a pointer, the change will be seen in a subsequent call to `SQLFetch()`. Furthermore, if a statement in this or a concurrent transaction *inserts* a row that satisfies the `WHERE` clause, the new row will not be seen by subsequent calls to `SQLFetch()` (since a pointer to the row is not in the set). Unfortunately, that is not the only anomaly that can occur. A change of an attribute value in a row that previously satisfied the `WHERE` clause can cause the row to no longer satisfy the clause. However, the row will still be visible through the cursor.

3. **DYNAMIC.** The data in the result set is completely dynamic. A statement in this or some other concurrently executing transaction can change, delete, *and* insert a row into the result set after the `SELECT` statement is executed. Those changes will be seen in subsequent calls to `SQLFetch()`.

The ODBC specification calls for the implementation of all of these cursor types. However, the mechanisms made available to ODBC by a particular DBMS might make the implementation of a particular cursor type difficult, so a driver for that DBMS might support only a subset of cursor types. Clearly, **DYNAMIC** cursors are the most difficult to implement, and many drivers do not implement them.



**FIGURE 8.15** Effect of using a KEYSET\_DRIVEN cursor to retrieve records from TRANSCRIPT.

The statement `SQLSetStmtOption()` is used to request a cursor type as follows:

---

```
SQLSetStmtOption(hstmt, SQL_CURSOR_TYPE, Option);
```

---

where *Option* is one of the constants `SQL_CURSOR_STATIC`, `SQL_CURSOR_DYNAMIC`, or `SQL_CURSOR_KEYSET_DRIVEN`.

ODBC also supports positioned updates through a non-`STATIC` cursor. For such an update, the `SELECT` statement must be defined with a `FOR UPDATE` clause. For example,

---

```
SQLExecDirect(hstmt1, "SELECT * \
    FROM EMPLOYEE \
    FOR UPDATE OF Salary");
```

---

uses a previously allocated statement handle, `hstmt1`, to prepare a query whose cursor will allow updates to the `EMPLOYEE` relation through the `Salary` attribute.

Now, suppose we have executed a number of `SQLFetch(hstmt1)` statements and the cursor is now positioned at the employee named Joe Public. To raise Joe's salary by \$1000, we might execute the following statement (where `hstmt2` is a previously allocated statement handle):

---

```
SQLExecDirect(hstmt2, "UPDATE EMPLOYEE \
    SET Salary = Salary + 1000 \
    WHERE CURRENT OF employee_cursor");
```

---

The only problem with the above statement is that the cursor name, `employee_cursor`, required by the `WHERE CURRENT OF` clause, comes out of the blue. In particular, it is not connected in any way to the cursor associated with the `hstmt1` handle that we used to execute the `SELECT` statement. Thus, before executing the

above UPDATE statement we must first give a name to that cursor. ODBC provides a special call to do just this:

---

```
SQLSetCursorName(hstmt1, employee_cursor);
```

---

As with dynamic SQL, information about the result set of a query might not be known when the program is written. ODBC thus provides a number of functions to obtain this information. For example, after a statement has been prepared, the program can call the function `SQLNumResultCols()`, to obtain the number of columns in a result set, and the functions `SQLColAttributes()` and `SQLDescribeCol()`, to provide information about a specific column in a result set.

ODBC also has a number of functions, called **catalog functions**, which return information about the database schema. For example, `SQLTables()` returns, in a result set, the names of all tables, and `SQLColumns()` returns column names.

### 8.6.3 Status Processing

The ODBC procedures we have been discussing are actually functions, which return a value, of type `RETCODE`, indicating whether or not the specified action was successful. Thus, we might have

---

```
RETCODE retcode1;  
:  
retcode1 = SQLConnect(...);  
if (retcode1 != SQL_SUCCESS) {  
    : do something  
}
```

---

Additional information about the error can be found by calling `SQLGetError()`.

### 8.6.4 Executing Transactions

By default, the database is in autocommit mode when a connection is created. To allow two or more statements to be grouped into a transaction, autocommit mode is disabled using

---

```
SQLSetConnectionOption(hdbc,  
                      SQL_AUTOCOMMIT,  
                      SQL_AUTOCOMMIT_OFF);
```

---

where `hdbc` is a connection handle. Initially, each transaction uses the default isolation level of the database manager. This can be changed with a call such as

---

```
SQLSetConnectionOption(hdbc,
                      SQL_TXN_ISOLATION,
                      SQL_TXN_REPEATABLE_READ);
```

---

Transactions can be committed or rolled back using

---



---

```
SQLTransact(henv, hdbc, Action);
```

---

where Action is either SQL\_COMMIT or SQL\_ABORT.

After a transaction is committed or rolled back, a new transaction starts when the next SQL statement is executed (or, in the case of the first SQL statement in the program, when that statement is executed).

If the program is connected to more than one database, the transactions at each database can be separately committed or rolled back. ODBC does not support a commit protocol that ensures that the set of transactions will be globally atomic. However, Microsoft has introduced a new TP monitor, MTS (Microsoft Transaction Server), that includes a transaction manager (and an appropriate API) guaranteeing an atomic commit of distributed transactions using ODBC.

## 8.6.5 Stored Procedures on the Server

ODBC can be used to call a stored procedure if the DBMS supports this feature. For example, to call the stored procedure of Figure 8.7 on page 284 we might begin with the statement

---

```
SQLPrepare(hstmt, "{call Deregister(?,?,?,?,?)}");
```

---

which prepares the call statement. The braces around the call to `Deregister()` denote the SQL escape syntax discussed in Section 8.5.7. The parameters of `Deregister()` can be bound to host variables with `SQLBindParameter()` functions. Then the procedure call can be executed with

---

```
SQLExecute(hstmt);
```

---

As with embedded SQL, if the DBMS allows a stored procedure to return a result set, the application program can retrieve data from the result set using a cursor.

## 8.6.6 An Example

Figure 8.16 is a fragment of a C program containing ODBC procedure calls. It executes the same transaction as that in Figure 8.12. The constant `SQL_NTS`, when supplied as an argument to a procedure (`SQLConnect()` and `SQLExecDirect()` in this example) indicates that the preceding argument is a null-terminated string.

**FIGURE 8.16** A fragment of an ODBC program written in C.

```
HENV henv;
HDBC hdcb;
HSTMT hstmt;
RETCODE retcode;
SQLAllocEnv(&henv);
SQLAllocConnect(henv, &hdcb);
retcode = SQLConnect(hdbc, dbName, SQL_NTS, "john", SQL_NTS, "j121",
                     SQL_NTS);
if (retcode != SQL_SUCCESS){
    SQLFreeEnv(henv);
    return(-1);
}
SQLSetConnectionOption(hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF);
SQLAllocStmt(hdbc, &hstmt);
retcode = SQLExecDirect(hstmt,
                        "DELETE FROM TRANSCRIPT \
                         WHERE StudId = 123456789 \
                           AND Semester = 'F2000' \
                           AND CrsCode = 'CS308'",
                        SQL_NTS);
if (retcode != SQL_SUCCESS) {
    SQLTransact(henv, hdcb, SQL_ABORT);
    SQLFreeStmt(hstmt, SQL_DROP);
    SQLDisconnect(hdbc);
    SQLFreeConnect(hdbc);
    SQLFreeEnv(henv);
    return(-2);
}
retcode = SQLExecDirect(hstmt,
                        "UPDATE CLASS \
                         SET Enrollment = (Enrollment - 1) \
                           WHERE CrsCode = 'CS308'",
                        SQL_NTS);
if (retcode != SQL_SUCCESS) {
    SQLTransact(henv, hdcb, SQL_ABORT);
    SQLFreeStmt(hstmt, SQL_DROP);
    SQLDisconnect(hdbc);
    SQLFreeConnect(hdbc);
    SQLFreeEnv(henv);
    return(-3);
}
SQLTransact(henv, hdcb, SQL_COMMIT);
SQLFreeStmt(hstmt, SQL_DROP);
SQLDisconnect(hdbc);
SQLFreeConnect(hdbc);
SQLFreeEnv(henv);
```

## 8.7 Comparison

We have discussed a variety of techniques for creating programs that can access a database—each has its own advantages and disadvantages. Here we summarize some of the issues involved:

- In some cases, the program contains SQL statements that use a special syntax (static SQL, SQLJ); in others, the statements are values of variables (dynamic SQL, JDBC, ODBC). An advantage of the former is its simplicity, but a disadvantage is that the interaction with the database is fixed at compile time. In some applications, the statement to be executed cannot be determined until run time; in such cases, it is important that the application be able to construct SQL statements dynamically.
- In some cases, the application must use the SQL dialect supported by the particular DBMS to which it is connected, making it difficult to port the application to a different vendor's product. In other cases (ODBC, JDBC), a single dialect or a common core is used in the application, and modules are provided to translate it to each vendor's dialect. Portability is thus enhanced, but the application might not have access to the proprietary features supported by a particular vendor's product.
- A number of factors are involved in assessing the run-time overhead incurred by each technique. These include the cost of communication, preparation, and parameter passing. In some cases, the cost depends on how a technique is implemented, but certain general observations can be made.

With a statement-level interface, the SQL statement has embedded parameter names that can be processed at compile time to generate parameter-passing code. At run time, the statement can be passed to the server for both preparation and execution, so a single communication is sufficient. With a call-level interface and with dynamic SQL, parameter names are not included in the statement because the statement is not available at compile time when parameter names can be mapped to addresses using the symbol table. Instead, they are provided separately so that they can be dealt with at compile time. Except in special cases (e.g., `EXECUTE IMMEDIATE`), therefore, one communication is used to send the statement for preparation and one additional communication is needed for requesting execution. This issue is important, since communication is expensive and time consuming.

Preparation must take place at run time if the SQL statement is constructed dynamically (dynamic SQL, JDBC, and ODBC). However, even with static SQL, preparation is generally done at run time when the statement is submitted for execution. In all of these cases, if the statement is executed many times, the preparation cost can be prorated over each execution and might not be a major factor. The most effective way to avoid run-time preparation is to use stored procedures, in which case the DBMS can be instructed to create and store a query execution plan prior to execution of the application. Often, this is simply a separate plan for each SQL statement in the procedure. More

sophisticated systems create an optimized plan for the procedure as a whole, since the sequence of SQL statements is known. Data structures created for one statement might be preserved for use by the next.

## BIBLIOGRAPHIC NOTES

Embedded and dynamic SQL date back to prehistoric times, and every SQL manual covers them to some degree. The following references are good places to look: [Date and Darwen 1997; Melton and Simon 1992; Gulutzan and Pelzer 1999].

There is vast literature on ODBC. Microsoft publishes an authoritative guide [Microsoft 1997], but there are more accessible books, such as [Signore et al. 1995]. The history and principles behind SQL/CLI, the new SQL standard analogous to (and designed to replace) ODBC, are discussed in [Venkatrao and Pizzo 1995]. SQL stored procedures are discussed in [Eisenberg 1996; Melton 1997]. Information on JDBC and SQLJ is readily available on the Web at [Sun 2000] and [SQLJ 2000], but published references, such as [Reese 2000; Melton et al. 2000], are better places to learn these technologies.

## EXERCISES

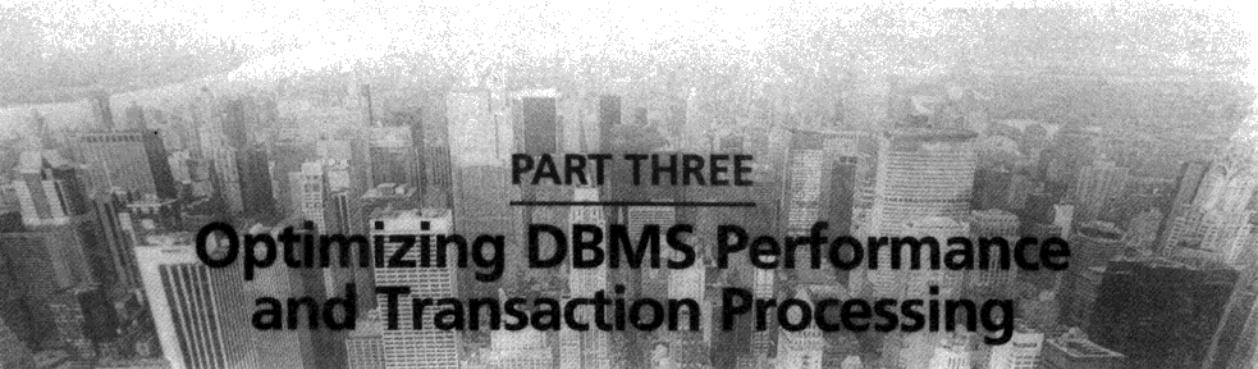
- 8.1 Explain why a precompiler is needed for embedded SQL and SQLJ but not for ODBC and JDBC.
- 8.2 Since the precompiler for embedded SQL translates SQL statements into procedure calls, explain the difference between embedded SQL and call-level interfaces, such as ODBC and JDBC, where SQL statements are specified as the arguments of procedure calls.
- 8.3 Explain why constraint checking is usually deferred in transaction processing applications.
- 8.4 Give an example where immediate constraint checking is undesirable in a transaction processing application.
- 8.5 Explain the advantages and disadvantages of using stored procedures in transaction processing applications.
- 8.6 Write transaction programs in
  - a. Embedded SQL
  - b. JDBC
  - c. ODBC
  - d. SQLJthat implement the registration transaction in the Student Registration System. Use the database schema from Figures 4.34 and 4.35.
- 8.7 Write transaction programs in
  - a. Embedded SQL
  - b. JDBC

- c. ODBC
- d. SQLJ

that use a cursor to print out a student's transcript in the Student Registration System. Use the database schema from Figures 4.34 and 4.35.

- 8.8** Explain the following:
- a. Why do embedded SQL and SQLJ use host language variables as parameters, whereas dynamic SQL, JDBC, and ODBC use the ? placeholders?
  - b. What are the advantages of using host language variables as parameters in embedded SQL compared with ? placeholders?
- 8.9** Explain the advantages and disadvantages of using dynamic SQL compared with ODBC and JDBC.
- 8.10** Write a transaction program in
- a. Embedded SQL
  - b. JDBC
  - c. ODBC
  - d. SQLJ
- that transfers the rows of a table between two DBMSs. Is your transaction globally atomic?
- 8.11** Write a Java program that executes in your local browser, uses JDBC to connect to your local DBMS, and makes a simple query against a table you have created.
- 8.12** Suppose that, at compile time, the application programmer knows all of the details of the SQL statements to be executed, the DBMS to be used, and the database schema. Explain the advantages and disadvantages of using embedded SQL as compared with JDBC or ODBC as the basis for the implementation.
- 8.13** Section 8.6 discusses KEYSET\_DRIVEN cursors.
- a. Explain the difference between STATIC and KEYSET\_DRIVEN cursors.
  - b. Give an example of a schedule in which these cursors give different results even when the transaction is executing in isolation.
  - c. Explain why updates and deletes can be made through KEYSET\_DRIVEN cursors.
- 8.14** Give an example of a transaction program that contains a cursor, such that the value returned by one of its FETCH statements depends on whether or not the cursor was defined to be INSENSITIVE. Assume that this transaction is the only one executing. We are not concerned about any effect that a concurrently executing transaction might have.
- 8.15** Compare the advantages and disadvantages of the exception handling mechanisms in embedded SQL, SQL/PSM, JDBC, SQLJ, and ODBC.





## PART THREE

---

# Optimizing DBMS Performance and Transaction Processing

In this part we will discuss various issues related to optimizing the performance of a database application. This subject is very important in the real world because it consumes a significant amount of the effort of database practitioners.

In Chapter 9 we will discuss the physical organization of databases, including various indexing mechanisms.

In Chapters 10 and 11 we will present the basic ideas of query processing and query optimization.

Chapter 12 discusses various aspects of database tuning, including the choice of indices, cache tuning, and methods for tuning the schema.

In conclusion, Chapter 13 provides an overview of transaction processing, which is one of the most important application areas for databases.



# 9

## Physical Data Organization and Indexing

One important advantage of SQL is that it is declarative. An SQL statement describes a query about information stored in a database, but it does not specify the technique the system should use in executing it. The DBMS itself decides that.

Such techniques are intimately associated with **storage structures**, **indices**, and **access paths**. A table is stored in a file, and the term “storage structure” is used to describe the way the rows are organized in the file. An index is an auxiliary data structure, perhaps stored in a separate file, that supports fast access to the rows of a table. An access path refers to a particular technique for accessing a set of rows. It uses an algorithm based on the storage structure of the table and on a choice among the available indices for that table.

An important aspect of the relational model is that the result of executing a particular SQL statement (i.e., the statement’s effect on the database and the information returned by it) is not affected by the storage structure used to store the table that is accessed, the indices that have been created for that table, or the access path that the DBMS chooses to use to access the table. Thus, when designing the query, the programmer does not have to be aware of these issues.

Although the choice of an access path does not affect the result produced by a statement, it does have a major impact on performance. Depending on the access path used, execution time might vary from seconds to hours, particularly when the statement refers to large tables involving thousands, and perhaps hundreds of thousands, of rows. Furthermore, different access paths are appropriate for different SQL statements used to query the same table.

Because execution time is sensitive to access paths, most database systems allow the database designer and system administrator to determine which access paths should be provided for each table. Generally, this decision is based on the nature of the SQL statements used to access the table and the frequency with which these statements are executed.

In this chapter, we describe a variety of access paths and compare their performance when used in the execution of different SQL statements. We continue this discussion in Chapter 12.

## 9.1 Disk Organization

For the following reasons, databases are generally stored on mass storage devices rather than in main memory:

- **Size.** Databases describing large enterprises frequently contain a huge amount of information. Databases in the gigabyte range are not uncommon, and those in the terabyte range already exist. Such databases cannot be accommodated in the main memory of current machines or of machines that are likely to be available in the near future.
- **Cost.** Even in situations in which the database can be accommodated in main memory (e.g., when its size is in the megabyte range), it is generally stored on mass storage.<sup>1</sup> The argument here is one of economy. The cost per byte of storage in main memory is on the order of one hundred times that of storage on disk.
- **Volatility.** We introduced the notion of durability in Chapter 2 in connection with the ACID properties of transactions. The argument for durability goes beyond transactions, however. It is an important property of any database system. The information describing an enterprise must be preserved in spite of system failures. Unfortunately, most implementations of main memory are **volatile**: the information stored in main memory can be lost during power failures and crashes. Hence, main memory does not naturally support database system requirements. Mass storage, on the other hand, is **nonvolatile** since the stored information survives failures. Hence, mass storage forms the basis of most database systems.

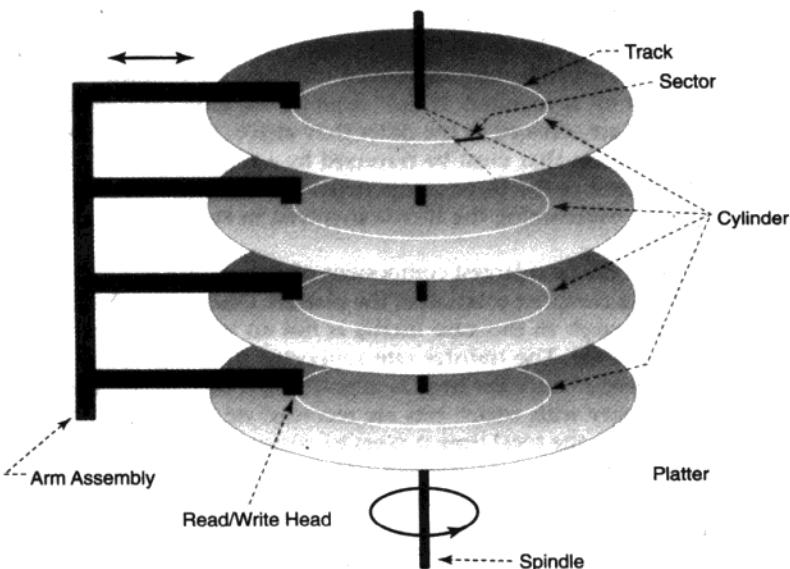
Since data on mass storage devices is not directly accessible to the system processors, accessing an item in the database involves first reading it from the mass storage device into a buffer in main memory—at which point a copy exists on both devices—and then accessing it from the buffer. If the access involves changing the item, the copy on mass storage becomes obsolete and the buffer copy must be used to overwrite it.

The most commonly used form of mass storage is a disk. Since the strategy used to enhance the performance of a database system is based on the physical characteristics of a disk, it is important to review how a disk works in order to understand performance issues.

A disk unit contains one or more circular **platters** attached to a rotating spindle, as shown in Figure 9.1. One, and sometimes both, surfaces of each platter are coated with magnetic material, and the direction of magnetism at a particular spot determines whether that spot records a one or a zero. Since information is stored magnetically, it is durable in spite of power failures.

Each platter (or surface) is accessed through an associated **read/write head** that is capable of either detecting or setting the direction of magnetism on the spot over which it is positioned. The head is attached to an arm, which is capable of moving

<sup>1</sup> In applications requiring very rapid response times, the database is often stored in main memory. Such systems are referred to as **main memory database systems**.



**FIGURE 9.1** Physical organization of a disk storage unit.

radially toward either the center or the circumference of the platter. Since platters rotate, this movement enables the head to be positioned over an arbitrary spot on the platter's surface.

As a practical matter, data is stored in **tracks**, which are concentric circles on the surface of the platter. Each platter has the same fixed number,  $N$ , of tracks, and the storage area consisting of the  $i$ th track on all platters is called an *i*th **cylinder**. Since there is a head for each platter, the disk unit as a whole has an array of heads, and the arm assembly moves all heads in unison. Thus, at any given time all read/write heads are positioned over a particular cylinder. It would seem reasonable to be able to read or write all tracks on the current cylinder simultaneously, but engineering limitations generally allow only a single head to be active at a time. Finally, each track is divided into **sectors**, which are the smallest units of transfer allowed by the hardware.

Before a particular sector can be accessed, the head must be positioned over the beginning of the sector. Hence, the time to access a sector,  $S$ , can be divided into three components:

1. **Seek time.** The time to position the arm assembly over the cylinder containing  $S$
2. **Rotational latency.** The additional time it takes, after the arm assembly is over the cylinder, for the platters to rotate to the angular position at which  $S$  is under the read/write head
3. **Transfer time.** The time it takes for the platter to rotate through the angle subtended by  $S$

The seek time varies depending on the radial distance between the cylinder currently under the read/write heads and the cylinder containing the sector to be read. In the worst case, the heads must be moved from track 1 to track  $N$ ; in the best case, no motion at all is required. If disk requests are uniformly distributed across the cylinders and are serviced in the order they arrive, it can be shown that the average number of tracks that must be traversed by a seek is about  $N/3$ .<sup>2</sup> Seek time is generally the largest of the three components because it involves not only mechanical motion but overcoming the inertia involved in starting and stopping the arm assembly.

Rotational latency is the next-largest component of access time and on average is about half the time of a complete rotation of the platters. Once again, mechanical motion is involved, although in this case inertia is not an issue. Transfer time is also limited by rotation time. The transfer rate supported by a disk is the rate at which data can be transferred once the head is in position; it depends on the speed of rotation and the density with which bits are recorded on the surface. We use the term **latency** to refer to the total time it takes to position the head and platter between disk accesses. Hence, latency is the sum of seek time and rotational latency.

A typical disk stores gigabytes of data. It might have a sector size of 512 bytes, an average seek time of 5 to 10 milliseconds, an average rotational latency of 2 to 5 milliseconds, and a transfer rate of several megabytes per second. Hence, a typical access time for a sector is on the order of 10 milliseconds.

The physical characteristics of a disk lead to the concept of the distance between two sectors, which is a measure of the latency between accesses to them. The distance is smallest if the sectors are adjacent on the same track since latency is zero if they are read in succession. After that, in order of increasing distance, sectors are closest if they are on the same track, on the same cylinder, or on different cylinders. In the last case, sectors on tracks  $n_1$  and  $n_2$  are closer than sectors on tracks  $n_3$  and  $n_4$  if  $|n_1 - n_2| < |n_3 - n_4|$ .

Several conclusions can be drawn from the physical characteristics of a disk unit:

- Most important, disks are extremely slow devices compared to CPUs. A CPU can execute hundreds of thousands of instructions in the time it takes to access a disk sector. Hence, in attempting to optimize the performance of a database system, it is necessary to optimize the flow of information between main memory and disk. While the database system should use efficient algorithms in processing data, the payoff in improved performance pales in comparison with that obtained from optimizing disk traffic. In recognition of this fact, in our discussion of access paths later in this chapter we estimate performance by counting I/O operations and ignoring processor time.
- In applications in which after accessing record  $A_1$ , the next access will, with high probability, be to record  $A_2$ , latency can be reduced if the distance between the

<sup>2</sup> The problem can be stated mathematically: suppose that we have an interval and we place two marks at random somewhere in the interval; how far apart are the marks on average?

sectors containing  $A_1$  and  $A_2$  is small. Thus the performance of the application is affected by the way the data is physically stored on the disk. For example, a sequential scan through a table can be performed efficiently if the table is stored on a single track or cylinder.

- To simplify buffer management, database systems transfer the same number of bytes with each I/O operation. How many bytes should that be? Using the typical numbers quoted earlier, it is apparent that the transfer time for a sector is small compared with the average latency. This is true even if more than one sector is transferred with each I/O operation. Thus, even though from a physical standpoint the natural unit of transfer is the data in a single sector, it might be more efficient if the system transfers data in larger units.

The term **page** generally denotes the unit of data transferred with each I/O operation. A page is stored on the disk in a **disk block**, or simply a block, which is a sequence of adjacent sectors on a track such that the page size is equal to the block size. A page can be transferred in a single I/O operation, with a single latency to position the head at the start of the containing block since there is no need to reposition either the arm or the platter in moving from one sector of the block to the next.

A trade-off is involved in choosing the number of sectors in a block (i.e., the page size). If, when a particular application accesses a record,  $A$ , in a table, there is a reasonable probability that it will soon access another record,  $B$ , in the table that is close to  $A$  (this would be the case if the table was frequently scanned), the page size should be large enough so that  $A$  and  $B$  are stored in the same page. This avoids a subsequent I/O operation when  $B$  is actually accessed. On the other hand, transfer time grows with page size, and larger pages require larger buffers in main memory. Hence, too large a page size cannot be justified, since it is far from certain that  $B$  will actually be accessed. With considerations such as these in mind, a typical page size is 4096 bytes (4 kilobytes).

A checksum is generally computed each time the data in a page is modified and stored with the data in the block. Each time the page is read a checksum over the data is computed and compared with the stored checksum. If the page has been corrupted on the disk it is very unlikely that the two values will be equal, and hence storage failures can be readily detected.

The system keeps an array of page-size buffers, called a **cache**, in main memory. With page I/O, the following sequence of events occurs when an application requests access to a particular record,  $A$ , of a table. We assume that each table is stored in a **data file**. First, the database system determines which of the file's pages contains  $A$ . Then it initiates the transfer of the page from mass storage into one of the buffers in the cache (we modify this part of the description in a moment). At some later time (generally signaled by an interrupt from the mass storage device), the system recognizes that the transfer has completed and copies  $A$  from the page in the buffer to the application.

The system attempts to keep the cache filled with pages that are likely to be referenced in the future. Then, if an application refers to an item contained

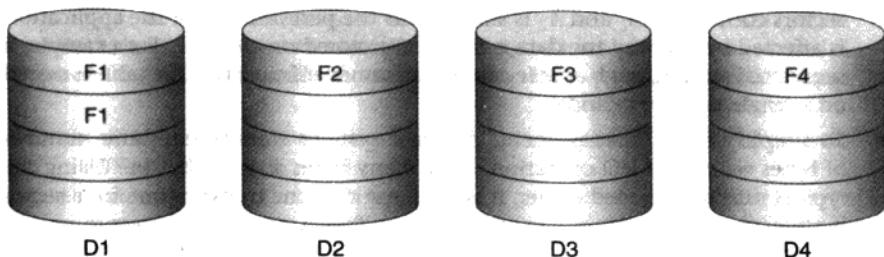


FIGURE 9.2 The striping of a file across four disks.

in a page in the cache, an I/O operation can be avoided and a **hit** is said to have occurred. Hence, proper cache management can greatly improve performance. Cache management is treated in more detail in Section 12.1.

### 9.1.1 RAID Systems

At various points in the book we will discuss a number of techniques that are made available to the application programmer or are provided within the DBMS to overcome the performance bottleneck caused by disk I/O: indices, query optimization, caches, and prefetching. A **RAID (Redundant Array of Independent Disks)** system is another, lower-level, approach to the problem. It consists of an array of disks configured to operate like a single disk. In particular, the RAID system responds to the usual operating system commands of read, write, and so on.

A RAID system has the potential for providing better throughput than a single disk of similar capacity since the individual disks of a RAID can function independently. There are two dimensions to this. If multiple requests access data on different disks of the RAID, they can be handled concurrently. Thus, a RAID involving  $n$  disks has the potential of improving throughput by a factor of  $n$ . The performance of a single request involving a large block of data can also be improved. If the data is spread over several disks, it can be transferred in parallel, reducing the transfer time. Thus, if it is spread over  $n$  disks, the transfer time can be reduced by a factor of  $n$ .

Data that is spread across an array of disks is said to be **striped**. The data is divided into **chunks** for this purpose, where the size of a chunk is configurable and might be a byte, a block, or some multiple of the block size. For example, if a file is striped over four disks, its first chunk, F1, would be stored on the first disk, D1, the second chunk, F2, on the second disk, D2, the third, F3, on D3, and the fourth, F4, on D4. The four chunks are referred to as a stripe. The fifth chunk, F5, is stored on D1 and successive chunks are laid out on the next stripe. The situation is shown in Figure 9.2. Then, if the entire file is to be read, data can be transferred from all four disks simultaneously.

In addition to increasing throughput, an important objective of RAID systems is to use redundant storage to increase the availability of the data by reducing

downtime due to disk failure. Such failures are magnified when an array of disks is used since the mean time to failure of at least one disk in an array of  $n$  is (approximately) the mean time to failure of one disk divided by  $n$ . For example, if a single disk has a mean time to failure of 10 years, the mean time to failure of at least one disk in an array of 50 is about two months—an unacceptable failure rate for most applications. To address this issue, five RAID levels have been defined, with different types of redundancy.

**Level 0.** Level 0 uses striping to increase throughput but does not use any redundancy. The bad news is that if all files are striped, the failure of a single disk destroys them all. Some people would say that Level 0 is not really RAID at all since there is no redundancy—the R in RAID.

**Level 1.** Level 1 does not perform striping but involves **mirroring**. In this case the disks in the array are paired, and the data stored on both elements of a pair is identical. Hence, a write to one (automatically) results in a write to the other, and as long as one disk in each pair is operational, all the data stored in the RAID can be accessed. Note that, assuming both disks in a pair are operational, the pair has twice the read rate of a single disk since both disks can be independently read at the same time (perhaps by different transactions). Although it might seem that doubling the number of required disks would be too expensive for many applications, the sharply decreased prices of disks have made mirroring increasingly attractive. Many transaction processing systems use mirrored disks to store their log, which has a very high requirement for durability.

**Level 3.** The higher levels of RAID perform striping and, in addition, store various types of redundant information to increase availability. In Level 3, the chunks are bytes (the striping is at the byte level), and in addition to the  $n$  disks on which the data is stored, the stripe includes an  $(n + 1)$ st disk that stores the *exclusive or* (XOR) of the corresponding bytes on the other  $n$  disks. It follows that if  $x_{i,j}$  is the  $j$ th bit on disk  $i$  then

$$x_{1,j} \text{ XOR } x_{2,j} \text{ XOR } \cdots \text{ XOR } x_{n+1,j} = 0$$

Thus, whenever a chunk is written on any one of the  $n$  disks, its XOR with the other  $n - 1$  chunks in the stripe is computed and stored as a byte on the  $(n + 1)$ st disk. This disk is sometimes called the **parity disk** since taking the XOR of a set of bytes is called “computing their parity.” In contrast to Level 1, instead of dedicating half of the capacity of the RAID to parity data, only  $1/n + 1$  of the capacity is used in this way.

For example, consider a RAID containing six disks and assume that parity is stored on the sixth disk. If the first bit of each byte on the first five disks is 1 0 1 0 0, then the first bit of the corresponding byte on disk six is 0. Observe that setting the bit on the parity disk equal to the XOR of the bits on the others also makes the bit on *each* disk the XOR of the bits on the others. Thus, if any disk fails, its information can be reconstructed as the XOR of the bits on the other disks. (Note

that this reconstruction method requires knowledge of which disk has failed, but this information can be supplied by the controller for that disk.)

With byte-level striping, a large data segment can be read at a very high transfer rate. This is important in a number of applications. For example, multimedia data must be transferred at a rate that is sufficient to keep up with real-time requirements. Level 3 is particularly useful for such single-user, high-transfer-rate applications.

Writes, however, can be a bottleneck since not only must new data be written to one of the  $n$  disks, but the new value of the parity must be computed and written to the parity disk. Its value is given by

$$\text{new\_parity\_bit} = (\text{old\_data\_bit} \text{ XOR } \text{new\_data\_bit}) \text{ XOR } \text{old\_parity\_bit}$$

Thus, four disk accesses are required to write a single byte: the old value of the byte to be overwritten and the old value of the parity byte must be read, and then the new data byte and the new parity byte must be written. Furthermore, since each write request to the RAID involves a write to the parity disk, that disk can become a bottleneck.

**Level 5.** Level 5 also involves striping and storing parity information, but it differs from Level 3 in two ways:

1. The chunks are disk blocks (or multiples of disk blocks), which is more efficient for some applications. For example, if there are many concurrent transactions, each of which wants to read a chunk of data, their requests can be satisfied in parallel. Thus, Level 5 can be viewed as an efficient way to distribute data over multiple disks to support parallel access.
2. The parity information is itself striped and is stored, in turn, on each of the disks, which eliminates the bottleneck caused by a single parity disk. Thus, in the above example, in the first stripe, the data blocks might be stored on the first five disks and the parity block on the sixth; in the second stripe, the data blocks might be stored on disks 1, 2, 3, 4, and 6 and the parity block on the fifth disk; in the third stripe, the parity block might be stored on the fourth disk, and so on.

**Level 10.** Many hardware vendors provide hybrid RAID levels combining some of the features of the basic RAID levels. One interesting hybrid is Level 10, which combines the disk striping of Level 0 with the mirroring of Level 1. It uses a striped array of  $n$  disks (as in Level 0), in which each of the  $n$  disks is mirrored (as in Level 1), making a total of  $2n$  disks. In other words, it consists of a striped array of  $n$  mirrored disks. Level 10 has the performance benefits of disk striping with the disk redundancy of mirroring. It provides the best performance of all RAID levels, but, as with Level 1, it requires doubling the number of required disks.

**Controller cache.** To further improve the performance of a RAID system, a controller cache can be included. This is a buffer in the main memory of the disk controller that can speed up both reading and writing.

- When reading, the RAID system can transfer a larger portion of data than was requested into the cache. If the program that made the request subsequently requests the additional data, it is immediately available in the buffer—no I/O has to be performed. For example, if a transaction is scanning a table, it will access all the pages in sequence. If  $n$  successive pages are stored in a stripe, and the entire stripe is retrieved when the first page is requested, I/O operations for subsequent pages in the stripe are eliminated. This is an example of prefetching, to which we will return in Chapter 12.
- In a **write-back cache** the RAID reports back that the write is complete as soon as the data is in the cache, before it has actually been written to the disk. Thus, the effective write time appears to be very fast. To implement a write-back cache, there must be some means of protecting the data in case of failures in the cache system. Some write-back caches are mirrored and/or have battery backup in case of electrical failure.
- The overhead of writing to a Level 5 RAID can be alleviated with a cache if all the blocks in a stripe are to be updated. In that case the parity block can be computed in the cache, and then all the blocks can be written to the disks in parallel: no extra disk reads are required. When used in this way the cache is referred to as a **write-gathering cache**.

Of all the RAID levels, Level 5, with a controller cache, is the one most often recommended for high-performance transaction processing applications.

## 9.2 Heap Files

We assume that each table is stored in a separate file, in accordance with some storage structure. The simplest storage structure is a **heap file**. With a heap file, rows are effectively appended to the end of the file as they are created. Thus the ordering of rows in the file is arbitrary. No rule dictates where a particular row should be placed. Figure 9.3 shows table TRANSCRIPT stored as a heap file. Recall that the table has four columns: StudId, CrsCode, Semester, and Grade. We have modified the schema so that grades are stored as real numbers, and we have assumed that four rows fit in a single page.

The important characteristic of a heap file, as far as we are concerned, is the fact that its rows are unordered. We ignore a number of significant issues since they are internal to the workings of the database system and the programmer generally has no control over them. Still, you should be aware of what these issues are. For example, we have assumed in Figure 9.3 that all rows are of the same length, and hence exactly the same number of rows fit in each page. This is not always the case. If, for example, the domain associated with a column is VARCHAR(3000), the number of bytes necessary to store a particular column value varies from 1 to 3000. Thus, the number of rows that will fit in a page is not fixed. This significantly complicates storage allocation. Furthermore, whether rows are of fixed or variable length, each page must be formatted with a certain amount of header information that keeps

666666666	MGT123	F1994	4.0
123454321	CS305	S1996	4.0
987654321	CS305	F1995	2.0
111111111	MGT123	F1997	3.0
page 0			
123454321	CS315	S1997	4.0
666666666	EE101	S1991	3.0
123454321	MAT123	S1996	2.0
234567890	EE101	F1995	3.0
page 1			
234567890	CS305	S1996	4.0
111111111	EE101	F1997	4.0
111111111	MAT123	F1997	3.0
987654321	MGT123	F1994	3.0
page 2			
425360777	CS305	S1996	3.0
666666666	MAT123	F1997	3.0
page 3			

FIGURE 9.3 TRANSCRIPT table stored as a heap file. At most four rows can be fit in a page.

track of the starting point of each row in the page and that locates unused regions of the page. We assume that the logical address of a row in a data file is given by a **row Id (rid)** that consists of a **page number** within the file and a **slot number** identifying the row within the page. The actual location of the row is obtained by interpreting the slot number using the page's header information. The situation is further complicated if a row is too large to fit in a page.

The good thing about a heap file is its simplicity. Rows are inserted by appending them to the end of the file, and they are deleted by declaring the slot that they occupy as empty in the header information of the page in which they are stored. Figure 9.4 shows the file of Figure 9.3 after the records for the student with Id 111111111 were deleted and the student with Id 666666666 completed CS305 in the spring of 1998. Note that deletion leaves gaps in the file and eventually a significant amount of storage is wasted. These gaps cause searches through the file to take longer because more pages have to be examined. Eventually, the gaps become so extensive that the file must be compacted. Compaction can be a time-consuming process if the file is large since every page must be read and the pages of the compacted version written out.

**Access cost.** We can compare the efficiency of accessing a heap file with other storage structures, which we discuss later, by counting the number of I/O operations required to do various operations. Let  $F$  denote the number of pages in the file. First

666666666	MGT123	F1994	4.0
123454321	CS305	S1996	4.0
987654321	CS305	F1995	2.0
			page 0
123454321	CS315	S1997	4.0
666666666	EE101	S1991	3.0
123454321	MAT123	S1996	2.0
234567890	EE101	F1995	3.0
			page 1
234567890	CS305	S1996	4.0
987654321	MGT123	F1994	3.0
			page 2
425360777	CS305	S1996	3.0
666666666	MAT123	F1997	3.0
666666666	CS305	S1998	3.0
			page 3

FIGURE 9.4 TRANSCRIPT table of Figure 9.3 after insertion and deletion of some rows.

consider insertion. Before a row,  $A$ , can be inserted, we must ensure that  $A$ 's key does not duplicate the key of a row already in the table. Hence, the file must be scanned. If a duplicate exists, it will be discovered in  $F/2$  page reads on average, and at that point the insertion is abandoned. The entire file has to be read in order to conclude that no duplicate is present, and then the last page (with  $A$  inserted) has to be rewritten, yielding a total cost of  $F + 1$  page transfers in this case.

A similar situation exists for deletion. If a tuple,  $A$ , with a specified key is present, it will be discovered in  $F/2$  page reads on average, and then the page (with  $A$  deleted) will be rewritten, yielding a cost of  $F/2 + 1$ . If no such tuple is present, the cost is  $F$ . If the condition specifying the tuples to be deleted does not involve a key, the entire file must be scanned since an arbitrary number of rows satisfying the condition can exist in the table.

A heap file is an efficient storage structure if queries on the table involve accessing all rows and if the order in which the rows are accessed is not important. For example, the query

---

```
SELECT *
FROM TRANSCRIPT
```

---

returns the entire table. With a heap storage structure, the cost is  $F$ , which is clearly optimal. Of course, if we want the rows printed out in some specific order, the cost is much greater since the rows have to be sorted before being output. An example of such a query is

---

```
SELECT *
FROM TRANSCRIPT T
ORDER BY T.StudId
```

---

As another example, consider the query

---

```
SELECT AVG(T.Grade)
FROM TRANSCRIPT T
```

---

which returns the average grade assigned in all courses. All rows of the table must be read to extract the grades no matter how the table is stored, and the averaging computation is not sensitive to the order in which reading occurs. Again, the cost,  $F$ , is optimal.

Suppose, however, a query requests the grade received by the student with Id 234567890 in CS305 in the spring of 1996.

---

```
SELECT T.Grade
FROM TRANSCRIPT T
WHERE T.StudId = '234567890' AND
      T.CrsCode = 'CS305' AND T.Semester = 'S1996'
```

---

9.1

Since {*StudId*, *CrsCode*, *Semester*} is the key of TRANSCRIPT, at most one grade will be returned. If TRANSCRIPT has the value shown in Figure 9.3, exactly one grade (namely, 4.0) will be returned. The database system must scan the file pages looking for a row with the specified key and return the corresponding grade. In this case, it must read three pages. Generally, an average of  $F/2$  pages must be read, but if a row with the specified key is not in the table, all  $F$  pages must be read. In either case, the cost is high considering the small amount of information actually requested.

Finally, consider the following queries. The first returns the course, semester, and grade for all courses taken by the student with Id 234567890, and the second returns the Ids of all students who received grades between 2.0 and 4.0 in some course. Since, in both cases, the WHERE clause does not specify a candidate key, an arbitrary number of rows might be returned. Therefore, the entire table must be searched at a cost of  $F$ .

---

```
SELECT T.Course, T.Semester, T.Grade
FROM TRANSCRIPT T
WHERE T.StudId = '234567890'
```

---

9.2

---

```
SELECT T.StudId
FROM TRANSCRIPT
WHERE T.Grade BETWEEN '2.0' AND '4.0'
```

---

**9.3**

The conditions in the WHERE clauses of statements (9.1) and (9.2) are equality conditions since the tuples requested must have the specified attribute values. A search for a tuple satisfying an equality condition is referred to as an **equality search**. The condition in the WHERE clause of statement (9.3) is a **range condition**. It involves an attribute whose domain is ordered and requests that all rows with attribute values in the specified range be retrieved. The actual value of the attribute in a requested tuple is not specified in the range condition. A search for a tuple satisfying a range condition is referred to as a **range search**.

## 9.3 Sorted Files

The last examples ((9.1), (9.2), and (9.3)) illustrate the weakness of heap storage. Even though information is requested about only a subset of rows, the entire table (or half the table) must be searched since without scanning a page we cannot be sure that it does not contain a row in the subset. These examples motivate the need for more sophisticated storage structures, one of which we consider in this section.

Suppose that, instead of storing the rows of a table in arbitrary order, we sort them based on the value of some attribute(s) of the table. We refer to such a structure as a **sorted file**. For example, we might store TRANSCRIPT in a sorted file in which the rows of the table are ordered on StudId, as shown in Figure 9.5. An immediate advantage of this is that if we are using a scan to locate rows having a particular value of StudId, we can stop the scan at the point in the file at which those rows must be located. But beyond that, we can now use a binary search. We explore these possibilities next.

**Access cost.** A naive way to search for the data records satisfying (9.2) is to scan the records in order until the first record having the value 234567890 in the StudId column is encountered. All records having this value would be stored consecutively, and access to them requires a minimum of additional I/O. In particular, if successive pages are stored contiguously on the disk, seek time can be minimized. In the figure, the rows describing courses taken by student 234567890 are stored on a single page. Once the first of these rows is made available in the cache, all others satisfying query (9.2) can be quickly retrieved. If the data file consists of  $F$  pages, an average of  $F/2$  page I/O operations is needed to locate the records—a significant improvement over the case in which TRANSCRIPT is stored in a heap file. Note that the same approach works for query (9.1).

In some cases, it is more efficient to locate the record with a **binary search** technique. The middle page is retrieved first, and its Id values are compared to 234567890. If the target value is found in the page, the record has been located; if the target value is less than (or greater than) the page values, the process is repeated

1111111111	MGT123	F1997	3.0	page 0
1111111111	EE101	F1997	4.0	
1111111111	MAT123	F1997	3.0	
123454321	CS305	S1996	4.0	page 1
123454321	CS315	S1997	4.0	
123454321	MAT123	S1996	2.0	
234567890	EE101	F1995	3.0	page 2
234567890	CS305	S1996	4.0	
425360777	CS305	S1996	3.0	
666666666	MGT123	F1994	4.0	page 3
666666666	MAT123	F1997	3.0	
666666666	EE101	S1991	3.0	
987654321	MGT123	F1994	3.0	page 4
987654321	CS305	F1995	2.0	

FIGURE 9.5 TRANSCRIPT table stored as a sorted file. At most four rows fit in a page.

recursively on the first half (or last half) of the data file. With this approach, the worst-case number of page transfers needed to locate the record with a particular student Id is approximately  $\log_2 F$ .

Unfortunately, the number of page transfers is not always the most accurate way of measuring the cost of searching a sorted file since it does not take into account the seek latency incurred. While doing a binary search, we might have to visit pages located on different disk cylinders, and this might involve considerable seek-latency costs. In fact, under certain circumstances the seek latency might be more important than the number of page transfers. Consider a sorted file that occupies  $N$  consecutive cylinders and in which successive pages are stored in adjacent blocks. A binary search might cause the disk head to move across  $N/2$ ,  $N/4$ , then  $N/8$  cylinders, and so on. Hence, the total number of cylinders traversed by the disk head will be about  $N$ , and the total cost of binary search will be

$$N \times \text{seek time} + \log_2 F \times \text{transfer time}$$

On the other hand, if we do a simple sequential search through the file, the average number of cylinders canvassed by the disk head will be  $N/2$  and the total cost

$$N/2 \times \text{seek time} + F/2 \times \text{transfer time}$$

Since seek time dominates transfer time, binary search is justified only if  $F$  is much larger than  $N$ .<sup>3</sup> In view of these results, neither sequential search nor binary search is considered a good option for equality searches in sorted files. A much more common approach is to augment sorted files with an index (see Section 9.4) and use binary search within the index. Since the index is designed to fit in main memory, a binary search over the index is very efficient and does not suffer from the disk-latency overhead described earlier.

Sorted files support range searches if the file is sorted on the same attribute as the requested range. Thus, the TRANSCRIPT file of Figure 9.5 supports a query requesting information about every student whose Id is between 100000000 and 199999999. An equality search for Id 100000000 locates the first tuple in the range (which might have Id 100000000 or, if such a tuple does not exist, will be the tuple with the smallest Id greater than this value). Subsequent tuples in the range occupy consecutive slots, and cache hits are likely to result as they are retrieved. If  $B$  is the number of rows stored in a page and  $R$  is the number of rows in a particular range, the number of I/O operations needed to retrieve all pages of the data file containing rows in the range (once the first row has been located) is roughly  $R/B$ .

Contrast these numbers with a heap file in which each row in the range can be on a different page and it is necessary to scan the entire file to ensure that all of them have been located. If the data file has  $F$  pages, the number of I/O operations is  $F$ . Similarly, a sorted file supports the retrieval of tuples that satisfy query (9.3), although in this case the table must be sorted on Grade instead. It cannot be sorted on both search keys at the same time.

**Maintaining sorted order.** In practice, it is difficult to maintain rows in sorted order if the table is dynamic. A heavy I/O price must be paid if, whenever a new row is inserted, all of the following rows have to be moved down one slot in the data file (that is, on average, half the pages have to be updated). One (partial) solution to this problem is to leave empty slots in each page when a data file is created to accommodate subsequent insertions between rows. The term **fillfactor** refers to the percentage of slots in a page that are initially filled. For example, the fillfactor for a heap file is 100%. In Figure 9.5 the fillfactor is 75% because three out of the four slots on a page are filled.

Empty slots do not provide a complete solution, however, since they can be exhausted as rows are inserted, and the problem then reappears for a subsequent insert. An **overflow page** might be used in this case, which Figure 9.6 illustrates using the TRANSCRIPT table. We have assumed that the initial state of the data file is as shown in Figure 9.5. Each page has a pointer field containing the page number of an overflow page (if it exists). In the figure, two new rows have been added to

<sup>3</sup> We have ignored the rotational delay, which strongly favors sequential over binary search.

				page 0
1111111111	MGT123	F1997	3.0	
1111111111	EE101	F1997	4.0	
1111111111	MAT123	F1997	3.0	
				page 1
123454321	CS305	S1996	4.0	
123454321	CS315	S1997	4.0	
123454321	MAT123	S1996	2.0	
Overflow: 5				page 2
234567890	EE101	F1995	3.0	
234567890	CS305	S1996	4.0	
234567890	LIT203	F1997	3.0	
425360777	CS305	S1996	3.0	
				page 3
666666666	MGT123	F1994	4.0	
666666666	EE101	S1991	3.0	
666666666	MAT123	F1997	3.0	
				page 4
987654321	MGT123	F1994	3.0	
987654321	CS305	F1995	2.0	
				page 5
313131313	CS306	F1997	4.0	

**FIGURE 9.6** The TRANSCRIPT table (augmented with overflow pointers) stored as a sorted file after the addition of several rows.

page 2 since the file was created. If an overflow page itself overflows, we can create an **overflow chain**: a linked list of overflow pages. Note that it is generally not the case that rows in the overflow chain are sorted. The problem of keeping the overflow chain sorted is the same as the problem of keeping the original file sorted.

While the use of overflow chains helps keep files **logically sorted**, we lose the advantage of the sorted files being stored in contiguous space on disk because an overflow page can be distant from the page that links to it. This causes additional latency during a sequential scan that reads records in order, and the number of I/O operations necessary to transfer records in a range is no longer  $R/B$ . If performance of sequential scan is an important issue (and it usually is), the data file must be reorganized periodically to ensure that all of its records are stored in contiguous disk space. The lesson here is that maintaining a data file in sorted order can be expensive if the file is dynamic.

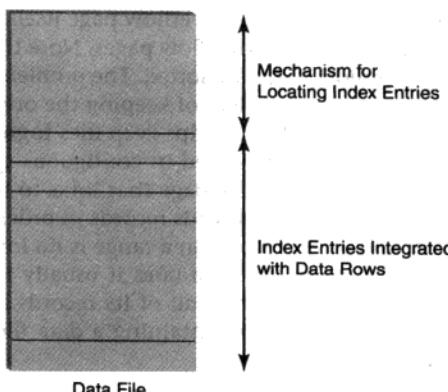
## 9.4 Indices

Suppose that you have set up a database for an application and given it to a set of users. However, instead of receiving a check in the mail for your work, you start getting angry phone calls complaining that the system is too slow. Some users claim that they have to wait a long time to get a response to their queries. Others claim that throughput is unacceptable. Queries are being submitted to the system at a rate greater than that at which the system is capable of processing them.

Indices can be used to improve this situation. An index over a table is analogous to a book index or a library card catalog. In the case of a book, certain terms that are of interest to readers are selected and made into index entries. In this case, each **index entry** contains a term (e.g., "gas turbines") and a pointer (e.g., "p. 348") to the location(s) in the book where the term appears. The entries are then sorted on the term to construct a table, called an **index**, for easy reference. Instead of having to scan the entire book for the locations at which a particular term is discussed, we can access the index efficiently and go directly to those locations.

The analogy is even closer in the case of a card catalogue. In this case, there might be several indices based on different properties of books: author, title, and subject. In each index, the entries are sorted on that property's value and each entry points to a book in the collection. Thus, an entry in an author index contains an author's name and a pointer (e.g., section, shelf) to a book that she has written.

Similarly, an index on a database table provides a convenient mechanism for locating a row (data record) without scanning the entire table and thus greatly reduces the time it takes to process a query. The property to be located is a column (or columns) of the indexed table called a **search key**. Do not confuse a search key of an index on a table with a candidate key of the table. Remember that a candidate key is a set of one or more columns having the property that no two rows in any instance of the table can have the same values in these columns. A search key does not have this restriction, so, for example, TRANSCRIPT has more than one row for the student with Id 11111111, which means that the index on StudId will have several index entries with the search-key value 11111111.



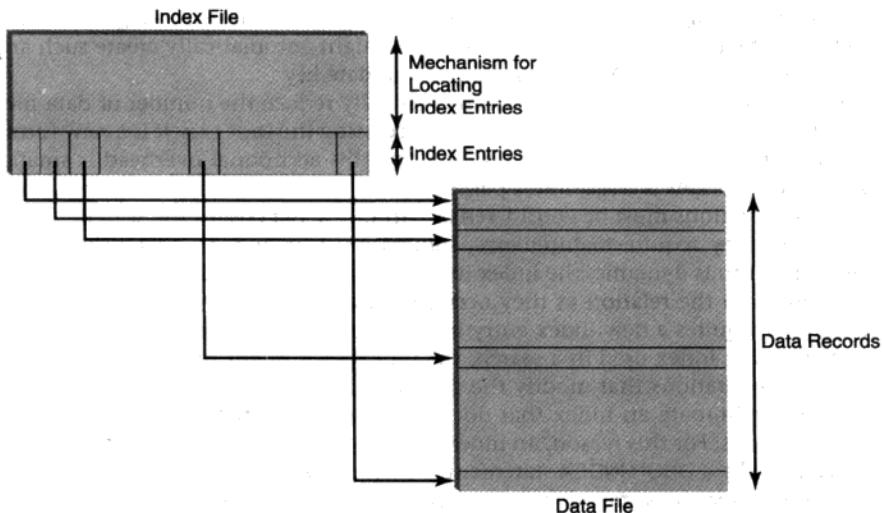
**FIGURE 9.7** A storage structure in which an index is integrated with the data records.

As with a candidate key, a search key can involve several columns. For example, the search key might include both the student Id and the semester. However, in contrast to a candidate key, the order of columns in a search key makes a difference, so you should think of a search key as a sequence (as contrasted with a set) of columns of the indexed table. You will see a context in which the ordering is important when we discuss partial-key searches.

An index consists of a set of index entries together with a mechanism for locating a particular entry efficiently based on a search-key value. With some indices, such as an ISAM index or a  $B^+$  tree, the location mechanism relies on the fact that the index entries are sorted on the search key. A hash index takes a different approach. In either case, the data structure that supports the location mechanism, together with the index entries, can be integrated into the data file containing the table itself, as shown in Figure 9.7. This integrated data file is regarded as a new storage structure and is called an **integrated index**. Later in this chapter we discuss ISAM,  $B^+$  trees, and hash storage structures as alternatives to heap and sorted storage structures. With an integrated storage structure, each index entry actually *contains* a row of the table (no pointer is needed).

Alternatively, the index might not be integrated, but instead stored in a separate file, called an **index file**, in which each index entry contains a search-key value and a rid.<sup>4</sup> This organization is shown in Figure 9.8. For example, the table TRANSCRIPT of

<sup>4</sup> With some indices, only the page Id field of the rid is stored. Since the major cost of accessing the data file is the cost of transferring a page, storing only the page Id in the index entry makes it possible to locate the correct page. Once the page has been retrieved, the record within it with the target search-key value can be located using a linear search. The added computational time for this search is generally small compared to the time to transfer a page and might be justified by the saving of space in each index entry. With smaller index entries, the index can fit in fewer pages, and hence less page I/O is needed to access it. In some index organizations, a single index entry can contain several pointers. Since this feature only complicates the discussion without adding any new concepts, we do not consider it further.



**FIGURE 9.8** A clustered index that references a separate data file.

Figure 9.3 on page 330 might have an index on `StudId`. Each index entry contains a value that appears in the `StudId` column in some row of the table and the rid of that row in the data file. Thus, the entry  $(425360777, (3,1))$  is present in the index.

Integration saves space since no pointer is needed and the search key does not have to be stored in both the index entry and the data record containing the corresponding row.

The SQL standard does not provide for the creation or deletion of indices. However, indices are an essential part of the database systems provided by most vendors. In some, for example, an index with a search key equal to the primary key is automatically created when a table is created. Such indices are often integrated storage structures that make it possible to efficiently guarantee that the primary key is unique when a row is added or modified (and to support the efficient execution of queries that involve the primary key). Heap storage structures result when no primary key is declared.

In addition to indices that are automatically created, database systems generally provide a statement (in their own dialect) that explicitly creates an index. For example,

---

```
CREATE INDEX TRANSGRD ON TRANSCRIPT (Grade)
```

---

creates an index named `TRANSGRD` on the table `TRANSCRIPT` with `Grade` as a search key. If `CREATE INDEX` does not provide an option to specify the type of index, a  $B^+$  tree is generally the result. In any case, the index created might be stored in an index file, as shown in Figure 9.8. The index might reference a heap or sorted file,

as shown in that figure, or the integrated storage structure shown in Figure 9.7 (in which case the table has two indices). A DBMS might automatically create such an index to enforce a **UNIQUE** constraint on a candidate key.

The use of an appropriate index can drastically reduce the number of data file pages that must be retrieved in a search, but accessing the index itself is a new form of overhead. If the index fits in main memory, this additional overhead is small. However, if the index is large, index pages must be retrieved from mass storage, and these I/O operations must be considered as part of the net cost of the search.

Because they require maintenance, indices must be added judiciously. If the indexed relation is dynamic, the index itself will have to be modified to accommodate changes in the relation as they occur. For example, a new row inserted into the relation requires a new index entry in the index. Thus, in addition to the cost of accessing the index used in a search, the index itself might have to be changed as a part of operations that modify the database. Because of this cost, it might be desirable to eliminate an index that does not support a sufficient number of the database queries. For this reason, an index is named in **CREATE INDEX** so that it can be referred to by a **DROP INDEX** statement, which causes it to be eliminated.

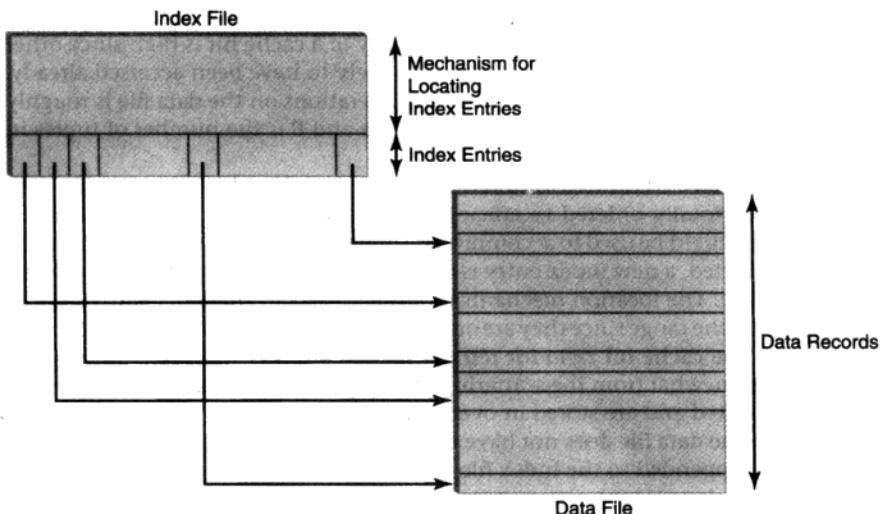
Before discussing particular index structures, we introduce several general properties for categorizing indices.

#### 9.4.1 Clustered versus Unclustered Indices

In a **clustered** index, the physical proximity of index entries in the index implies some degree of proximity among the corresponding data records in the data file. Such indices enable certain queries to be executed more efficiently than with unclustered indices. Query optimizers can use the fact that an index is clustered to improve the execution strategy. (Chapters 10 and 11 explain how clustering information is used by the query optimizer.) Clustering takes different forms depending on whether index entries are sorted (as in ISAM and  $B^+$  trees) or not (as in hash indices). A sorted index is **clustered** if the index entries and the data records are sorted on the same search key; otherwise, it is said to be **unclustered**. Clustered hash indices will be defined in Section 9.6.1. These definitions imply that, if the index is structured so that its entries are integrated with data records (the entries contain the records), it *must* be clustered. The index shown in Figure 9.8 is an example of a clustered index in which the index and the table are stored in separate files (and therefore the data records are not contained in the index entries). The regular pattern of the pointers from the index entries to the records in the data file is meant to reflect the fact that both index entries and data records are sorted on the same columns. The index shown in Figure 9.9 is unclustered.

A clustered index is often called a **main index**; an unclustered index is often called a **secondary index**.<sup>5</sup> There can be at most a single clustered index (since the

<sup>5</sup> Clustered indices are also sometimes called **primary indices**. We avoid this terminology because of a potential confusion regarding the connection between primary indices and the primary keys of relations. A primary index is *not* necessarily an index on the primary key of a relation. For instance, the PROFESSOR relation could be sorted on the **Department** attribute (which is not even a key) and



**FIGURE 9.9** An unclustered index over a data file.

data file can be sorted on at most one search key), but there can be several secondary indices. An index created by a `CREATE TABLE` statement is often clustered. With some database systems, a clustered index is always integrated into the data file as a storage structure. In this case, index entries contain data records, as shown in Figure 9.7. An index created by `CREATE INDEX` is generally a secondary, unclustered index stored in a separate index file (although some database systems allow the programmer to request the creation of a clustered index, which involves reorganizing the storage structure). The search key of the main index might be the primary key of the table, but this is not necessarily so.

A file is said to be **inverted** on a column if a secondary index exists with that column as a search key. It is **fully inverted** if a secondary index exists on all columns that are not contained in the primary key.

A clustered index with a search key,  $sk$ , is particularly effective for range searches involving  $sk$ . Its location mechanism efficiently locates the index entry whose search-key value is at one end of the range, and, since entries are ordered on  $sk$ , subsequent entries in the range are in the same index page (or in successive pages—we discuss this situation in Section 9.5). The data records can be retrieved using these entries.

The beauty of a clustered index is that the data records are themselves grouped together (instead of scattered throughout the data file). They are either contained in

a clustered index could be built, while the index on the  $Id$  attribute of that relation (which is also its primary key) will be unclustered because rows will not be sorted on that attribute.

the index entries or ordered in the same way (see Figure 9.8). Hence, in retrieving a particular data record in the range, the probability of a cache hit is high since other records in the page containing that record are likely to have been accessed already. As described in Section 9.3, the number of I/O operations on the data file is roughly  $R/B$ , where  $R$  is the number of tuples in the range and  $B$  is the number of tuples in a page.

With a clustered index that is stored separately from the data, the data file does not have to be totally ordered to efficiently process range searches. For example, overflow pages might be used to accommodate dynamically inserted records. As each new row is inserted, a new index entry is constructed and placed in the index file in the proper place. The location mechanism of the index can then efficiently find all index entries in the range since they are ordered in one (or perhaps several successive) index pages. The cache-hit ratio for retrieving data pages is still high, although it might suffer somewhat from the scattering of the rows that were appended after the data file was sorted and are stored in overflow pages.

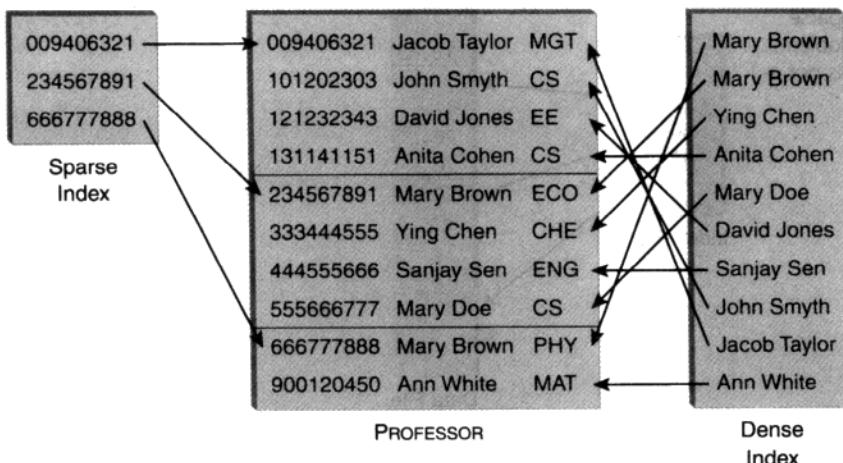
Although the data file does not have to be completely sorted, index entries cannot simply be appended to the index file—they must be integrated into the index's location mechanism. Hence, we seem to have replaced one difficult problem (keeping the data file sorted) with another (keeping the index file properly organized). However, the latter problem is not as difficult as it appears. For one thing, index entries are generally much smaller than data records, so the index file is much smaller than the data file and therefore index reorganization requires less I/O. Furthermore, as we will see, the algorithms associated with a  $B^+$  tree index are designed to accommodate the efficient addition and deletion of index entries.

The algorithm for locating the index entries containing search-key values in a range is the same for clustered or unclustered indices. The problem with an unclustered index is that, instead of finding the corresponding data records in the index entries or grouped together in the data file, the records might be scattered throughout the data file. Thus, if there are  $R$  entries in the range, as many as  $R$  separate I/O operations might be necessary to retrieve the data records (we will discuss a technique for optimizing this number in a later section).

For example, a data file might contain 10,000 pages, but there might be only 100 records in the range of a particular query. If an unclustered index on the attributes of the WHERE clause of that query is available, at most 100 I/O operations will be needed to retrieve the pages of the data file (as compared with 10,000 I/O operations if the data is stored in a heap file with no index). If the index is clustered and each data file page contains an average of 20 data records, approximately five data pages will have to be retrieved. We have ignored the I/O operations on the index file in this comparison. We deal with index I/O in the individual discussions of different index structures later in this chapter.

#### 9.4.2 Sparse versus Dense Indices

We have been assuming that indices are dense. A **dense index** is one whose entries are in a one-to-one correspondence with the records in the data file. A secondary,



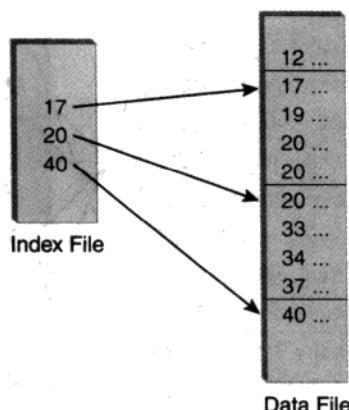
**FIGURE 9.10** The index entries of (left) a sparse index with search key *Id* and (right) a dense index with search key *Name*. Both refer to the table *PROFESSOR* stored in a file sorted on *Id*.

unclustered index must be dense, but a clustered index need not be. A **sparse index** over a sorted file is one in which there is a one-to-one correspondence between index entries and pages of the data file. The entry contains a value that is less than or equal to all values in the page it refers to. The difference between a sparse and a dense index is illustrated using the table *PROFESSOR* in Figure 9.10. To simplify the figure we do not show overflow pages in the data file and assume that all slots are filled. The data file is indexed by two separate index files. Only the index entries are shown in the index files, not the location mechanisms. Once again, we assume four slots per page. The *Id* attribute is the primary key of this table and is also the search key for the sparse index shown at the left of the figure. The *Name* attribute is the search key for the dense index at the right.

To retrieve the record for the faculty member with *Id* 333444555, we can use the sparse index to locate the index entry containing the largest value that is smaller than the target *Id*. In our case, the entry contains the value 234567891 and points to the second page of the data file. Once that page has been retrieved, the target record can be found by searching forward in the page. With a sparse index, it is essential that the data file be ordered on the same key as the index since it is the ordering of the data file that allows us to locate a record not referenced by an index entry. Hence, the sparse index must be clustered.

An important point concerning sparse indices is that the search key should be a candidate key of the table since, if several records exist in the data file with the same search key value and they are spread over several pages, the records in the first of those pages might be missed. The problem is illustrated in Figure 9.11. The search key of the sparse index is the first column of the table, which is not a candidate key.

**FIGURE 9.11** Sparse index with a problem: the search key is not a candidate key.



A search through the index for records having a search-key value of 20 follows the pointer in the index having a search-key value 20. Searching forward in the target page yields a single record with search-key value 20 and misses the records with that value in the previous page of the data file.

Several techniques can be used to correct this problem. The simplest is to start the search through the data file at the prior page when the target value is equal to the value in the sparse index. Another approach is to create an index entry for each *distinct search-key value* in the table (as opposed to one index entry per page). In that case, there might be several entries that point to the same page, but if there is considerable duplication of search-key values in the rows of the table, this index will still be considerably smaller than a dense index.

Unclustered indices are dense. The index on the right in Figure 9.10 is unclustered and, since an index entry exists for each record, the search key need not be a candidate key of the table. Therefore, several index entries can have the same search-key value (as shown in the figure).

### 9.4.3 Search Keys Containing Multiple Attributes

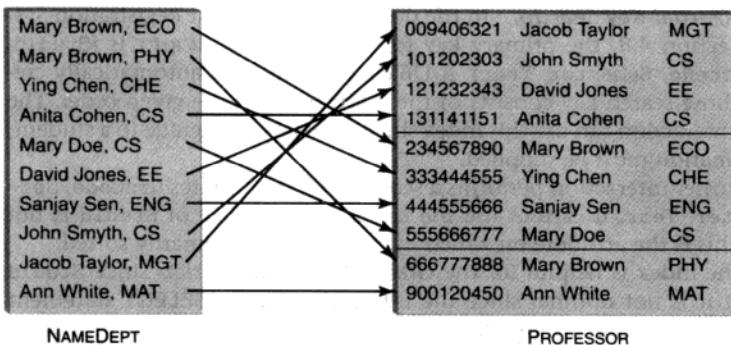
Search keys can contain multiple attributes. For example, we might construct an index on PROFESSOR by executing the statement

---

```
CREATE INDEX NAMEDEPT ON PROFESSOR (Name, DeptId)
```

---

Such an index is useful if clients of the supported application frequently request information about professors they identify by name and department Id. With such an index, the system can directly retrieve the required record. Not only is a scan of the entire table avoided, but records of professors with the same name in different departments are not retrieved (there might be two professors having the same name in a particular department, but this is unlikely).



**FIGURE 9.12** Dense index on PROFESSOR with search key **Name**, **DeptId**.

A dense, unclustered index that results from this statement is shown in Figure 9.12. The index entries are lexicographically ordered on **Name**, **DeptId** (notice that the order of the two entries for Mary Brown is now reversed from their position in the dense index of Figure 9.10).

One advantage of using multiple attributes in a search key is that the resulting index supports a finer granularity search. **NAMEDEPT** allows us to quickly retrieve the information about the professor named Mary Brown in the Economics Department; the dense index of Figure 9.10 requires that we examine two data records.

A second advantage of multiple attributes arises if the index entries are sorted (as with an ISAM or  $B^+$  tree index but not a hash index) since then a variety of range searches can be supported. For example, we can not only retrieve the records of all professors named Mary Brown in a particular department (an equality search), but in addition we can retrieve the records of all professors named Mary Brown in any department with a name alphabetized between economics and sociology, or all professors named Mary Brown in all departments, or any professor whose name is alphabetized between Mary Brown and David Jones in all departments. All of these are examples of range searches. In each case, the search is supported because index entries are sorted first on **Name** and second on **DeptId**. Hence, the target index entries are (logically) consecutive in the index file, and we are able to limit the range of entries to be scanned. The last two searches are examples of **partial-key searches**—that is, the values for some of the attributes in the search key are not specified.

Note that a range search that is not supported by **NAMEDEPT** is one in which a value for **Name** is not supplied (e.g., retrieve the records of all professors in the Computer Science Department) since in that case the desired index entries are not consecutive in the index file. This is precisely the reason that, in distinguishing a search key from a candidate key, we said that the ordering of attributes in a search key is important whereas it is not important for a candidate key. With partial-key searches, values for a *prefix* of the search key can be used in the index search, while searching on a proper suffix of the key is not supported by the index.

**Example 9.4.1 (Partial-Key Search).** In the design of the Student Registration System given in Section 4.8, the primary key for the table TRANSCRIPT is (StudId, CrsCode, SectionNo, Semester, Year). A DBMS will generally automatically create an index whose search key consists of these attributes in the given order. The Student Grade interaction can use this index since, in assigning a grade to a student, all of this information must be supplied.

The Class Roster interaction performs a search on the attributes CrsCode, SectionNo, Semester, Year. It cannot use the index since a value of StudId is not provided, and this is the attribute on which index entries are primarily sorted.

However, the index is helpful for the Grade History interaction since StudId is supplied, but it is not optimal since the interaction uses a SELECT statement in which the WHERE clause specifies StudId, Semester, and Year. Unfortunately, the index does not support the use of Semester and Year unless CrsCode and SectionNo are also supplied. Reversing the order of the attributes in the primary key specification allows it to support the Grade History interaction optimally and also the Student Grade interaction. An additional index is needed for the Class Roster interaction. ■

In general, a tree index on table R with search key K supports a search of the form

$$\sigma_{attr_1 op_1 val_1 \wedge \dots \wedge attr_n op_n val_n}(R)$$

9.4

if some prefix of K is a subset of  $\{attr_1, attr_2, \dots, attr_n\}$ . For example, if s attributes of (9.4),  $s \leq n$ , are a prefix of K (assume for simplicity that these are the first s attributes), the search locates the smallest index entry satisfying  $attr_1 = val_1 \wedge \dots \wedge attr_s = val_s$  and scans forward from that point to locate all entries satisfying (9.4). Thus, a particular index can be used in different ways to support a variety of searches and can therefore be used in a variety of access paths to R.

Finally, since an index can have a search key with multiple attributes, it can contain an arbitrary fraction of the information in the indexed table. In particular, a dense unclustered index has an index entry for each row, r, and that entry contains r's values of the search-key attributes. The implication is that, for some queries, it is possible to find all the requested information in the index itself, *without accessing the table*. Thus, for example, the result of executing the query

---

```

SELECT  P.Name
FROM    PROFESSOR P
WHERE   P.DeptId = 'EE'

```

---

can be obtained from the index NAMEDEPT, without accessing PROFESSOR, by scanning the index entries. Such a scan is less costly than a scan of PROFESSOR, since the index is stored in fewer pages than is the table. The use of an index in this way is referred to as an *index-only strategy* and is discussed in Section 12.2.1.

## 9.5 Multilevel Indexing

In previous sections, we discussed the index entries but not the location mechanism used to find them. In this section we discuss the location mechanism used in tree indices and then describe its use more specifically in the *index-sequential access method (ISAM)* and in  $B^+$  *trees*.

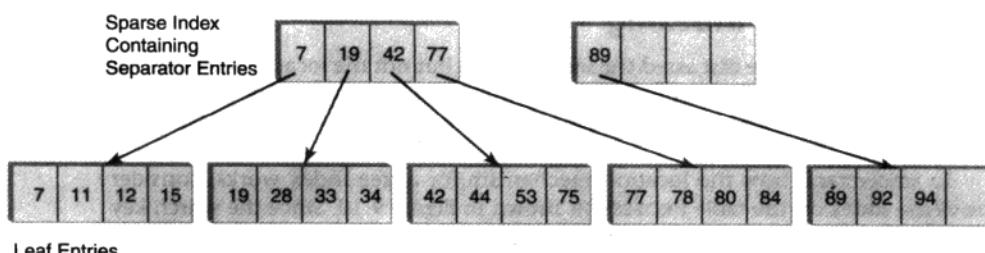
To understand how the location mechanism for a tree index works, consider the dense index on the table PROFESSOR shown in Figure 9.10. Since the search key is Name, the index entries are ordered on that field, and since the data records are ordered differently in the data file, the index is unclustered. We assume that the list of index entries is stored in a sequence of pages of the index file. A naive location mechanism for the index is a binary search over the entries. Thus, to locate the data record describing Sanjay Sen, we enter the list of index entries at its midpoint and compare the value Sanjay Sen with the search-key values found in that page. If the target value is found in the page, the index entry has been located; if it is less (or greater) than the values in the page, the process is repeated recursively on the first half (or, respectively, the last half) of the list. Once the index entry is located, the data page containing the record can be retrieved with one additional I/O operation.

It is important to note that a binary search on the list of index entries is a major improvement over a binary search on a sorted data file, as described in Section 9.3, since data records are generally much larger than index entries. Thus, if  $Q$  is the number of pages containing the index entries of a dense tree index and  $F$  is the number of pages in the data file,  $Q$  is much less than  $F$ . The number of I/O operations needed to locate a particular index entry using binary search can be no larger than approximately  $\log_2 Q$ .

We can further reduce the cost of locating the index entry by indexing the list of index entries itself. We construct a sparse index, using the same search key, on the index entries (a sparse index is possible because the index entries are sorted on that key) and do a binary search on that index. The entries in this second-level index serve as separators that guide the search toward the index entries of the first-level index. We thus use **leaf entry** to refer to the index entries (the entries that reside in the lowest level of the tree) and **separator entry** to refer to the entries that reside at higher levels.

The technique is illustrated in Figure 9.13, which shows a two-level index. To keep the figure concise, we assume a table with a candidate key having an integer domain, and we use that key as a search key for the index. We assume that a page of the index file can accommodate four index entries. Of course, in real systems one index page accommodates many more (e.g., 100), and the size of the index is relatively insignificant compared to the size of the data file.

In this and subsequent figures, we do not explicitly show the data records. The figures can be interpreted in two ways: (1) the leaf entries contain pointers to the data records in a separate data file; (2) the leaf entries contain the data records (the index is clustered in this case), and the figure represents a storage structure. In interpretation 1, the index file contains both leaf entries and the second-level index, and one additional I/O operation is required to access the data record in the



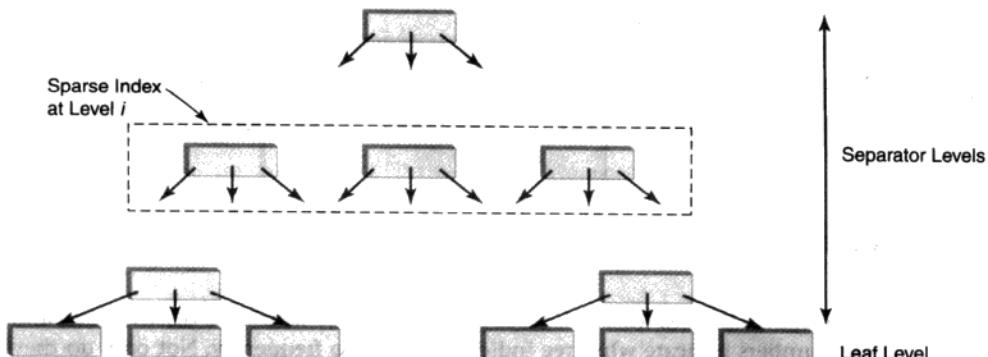
**FIGURE 9.13** A two-level index. At most four entries fit in a page.

data file. In interpretation 2, retrieving a leaf entry also retrieves the corresponding data record, and no additional I/O is required. Our discussion of multilevel indices applies to both interpretations.

Finally, although in the figure the separators in the second-level index look identical to leaf entries, this is not always the case. In interpretation 2, only leaves, not separator entries, contain data records, and thus they are considerably larger than the separator entries. In interpretation 1, separators and leaf entries look identical, except that separators in the second level point to the nodes of the first level whereas leaf entries point to the records in the actual data file. The formats of these pointers will be different.

With a two-level index, we replace the binary search of the leaves by a two-step process. If we are looking for a data record with search-key value  $k$ , then we first do a binary search of the top-level sparse index to find the appropriate separator entry. The  $i$ th separator entry, with search-key value  $k_i$ , points to a page containing index entries with search-key values greater than or equal to  $k_i$ . It will be the appropriate entry if  $k_i \leq k < k_{i+1}$ . In this case we next retrieve that page to find the index entry (if it exists). Thus, if we are looking for a search-key value of 33, we first do a binary search of the upper index to locate the rightmost separator with a value less than or equal to 33, which in this case is the separator containing 19. We then follow the pointer to the second page of index leaves and do a linear search in that page to find the desired index entry. If the target search-key value were 32, we would perform the same steps and conclude that no row in the indexed table had value 32 in the candidate-key field.

What have we achieved by introducing the second-level index? Since the upper-level index is sparse, it has fewer separator entries than there are leaf entries at the first level. If we assume separate data and index files, separator and leaf entries in the index are roughly the same size. Moreover, if we assume 100 entries in an index page, the second-level index occupies  $Q/100$  pages (where  $Q$  is the number of leaf pages) and the maximum cost of accessing a leaf entry using a binary search of the second-level index is roughly  $\log_2(Q/100)$  page I/Os plus 1 (for the page containing the index leaf). This compares to  $\log_2 Q$  for a binary search over the index's leaf entries and represents a major savings if  $Q$  is large.



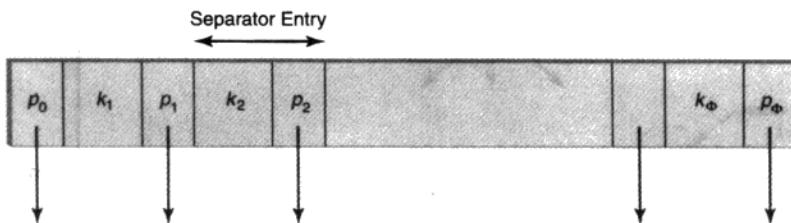
**FIGURE 9.14** Schematic view of a multilevel index.

These considerations lead us to a multilevel index. If a two-level index is good, why not a multilevel index? Each index level is indexed by a higher-level sparse index of a smaller size, until we get to the point where the index is contained in a single page. The I/O cost of searching that index is 1, and so the total cost of locating the target leaf entry equals the number of index levels.

A multilevel index is shown schematically in Figure 9.14. Each shaded rectangular box represents a page. The lowest level of the tree contains the leaf entries and is referred to as the **leaf level**. If we concatenate all of the pages at this level in the order shown, the leaf entries form an ordered list. The upper levels contain separators and are referred to as **separator levels**. This is the location mechanism of a tree index. If we concatenate all of the pages at a particular separator level in the order shown, we get a sparse index on the level below. The root of the tree (the top separator level) is a sparse index contained in a single page of the index file. If the leaf entries contain data records, the figure shows the storage structure of a tree-indexed file.

We use the term **index level** to refer to any level of an index tree, leaf or separator. We use the term **fan-out** to refer to the number of index separators in a page. The fan-out controls the number of levels in the tree: the smaller it is, the more levels the tree has. The number of levels equals the number of I/O operations needed to fetch a leaf entry. If the fan-out is denoted by  $\Phi$ , the number of I/O operations necessary to retrieve a leaf entry is  $\log_{\Phi}Q + 1$ .

If, for example, there are 10,000 pages at the leaf level and the fan-out is 100 ( $10^6$  rows in the data file assuming that leaf and separator entries are the same size), three page I/Os are necessary to retrieve a particular leaf. Thus, with a large fan-out, traversal of the index, even for a large data file, can be reduced to a few I/O operations. Since the root index occupies only a single page, it can often be kept in main memory, further reducing the cost. It might be practical to keep even the second-level index, which in this case occupies 100 pages, in main memory.



**FIGURE 9.15** Page at a separator level in an ISAM index.

Multilevel indices form the basis for tree indices, which we discuss next, and these numbers indicate why tree indices are used so frequently. Not only do they provide efficient access to the data file, but, as we will see, they also support range queries.

### 9.5.1 Index-Sequential Access

The **index-sequential access method (ISAM)**<sup>6</sup> is based on the multilevel index. An ISAM index is a main index, and hence it is a clustered index over records that are ordered on the search key of the index. Generally, the records are contained in the leaf level, so ISAM is a storage structure for the data file.

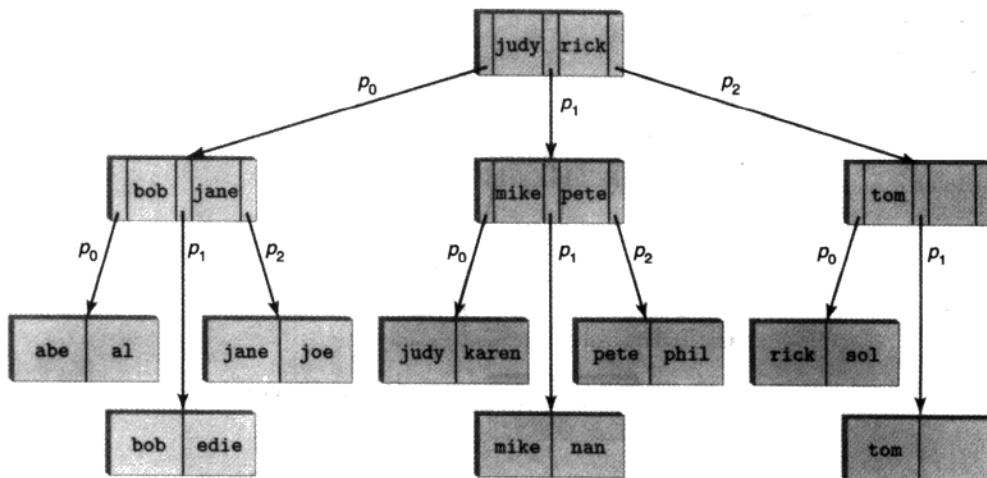
The format of a page at a separator level is shown in Figure 9.15. Each separator level is effectively a sparse index over the next level below. Each separator entry consists of a search-key value,  $k_i$ , and a pointer,  $p_i$ , to another page in the storage structure. This page might be in the next, lower separator level, or it might be a page at the leaf levels. The separators are sorted in the page, and we assume that a page contains a maximum of  $\Phi$  separators.

Each search-key value,  $k_i$ , separates the set of search-key values in the two subtrees pointed to by the adjacent pointers,  $p_{i-1}$  and  $p_i$ . If a search-key value,  $k$ , is found in the subtree referred to by  $p_{i-1}$ , it satisfies  $k < k_i$ ; if it is found in the subtree referred to by  $p_i$ , it satisfies  $k \geq k_i$  (hence the term “separator”). It appears as if an ISAM index page contains an extra pointer,  $p_0$ , if we compare it with a page of a sparse index in a multilevel index.<sup>7</sup> Actually, a better way to compare a page of an ISAM index to an index page at the separator level is that the latter contains an extra search-key value: the smallest search-key value in the page,  $k_0$ , is actually unnecessary.

**Example 9.5.1 (ISAM Index).** Figure 9.16 is an example of an ISAM index. In it, the tree has two separator levels and a leaf level. Search-key values are the names

<sup>6</sup> The term “access method” is often used interchangeably with the term “access path.” We prefer **access path** since it conforms more closely to the concept.

<sup>7</sup> Note that the fan-out is now  $\Phi + 1$ . Since  $\Phi$  is generally much greater than 1, we ignore the difference between  $\Phi$  and  $\Phi + 1$  in cost calculations.



**FIGURE 9.16** An example of an ISAM index.

of students, and the ordering is lexicographic. All search-key values in the leftmost subtree (the subtree referred to by  $p_0$  in the root page) are less than judy, and all search-key values in the middle subtree (the subtree referred to by  $p_1$  in the root page) are greater than or equal to judy.

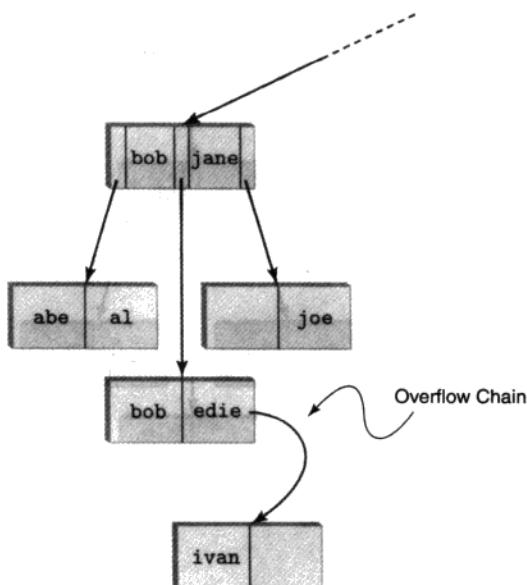
A search for the data record containing the search-key value karen starts at the root, determines that the target value is between judy and rick, and follows  $p_1$  to the middle page at the next index level. From that page, it determines that karen is less than mike and follows  $p_0$  to the leaf page that must contain the index entry for karen (if it exists). If the goal were to retrieve all data records with search-key values between karen and pete, we would locate the index entry with the largest search-key value less than or equal to karen and then scan the leaf level until the first entry with search-key value greater than pete is encountered. Because pages at the leaf level are stored sequentially in sorted order, the desired entries are consecutive in the file. ■

Example 9.5.1 illustrates the use of an ISAM index in supporting a range search. In addition the index supports keys with multiple attributes and partial-key searches.

An ISAM file is built by first allocating pages sequentially in the storage structure for the leaf pages (containing the data rows) and then constructing the separator levels from the bottom up: the root is the topmost index built. Therefore, the ISAM index initially has the property that all search-key values that appear at a separator level also appear at the leaf level.

The separator levels of an ISAM index never change once they have been constructed. It is for this reason that an ISAM index is referred to as a static index. Although the contents of leaf-level pages might change, the pages themselves are

**FIGURE 9.17** Portion of the ISAM index of Figure 9.16 after an insertion and a deletion.



not allocated or deallocated and hence their position in the file is fixed. If a row of the table is deleted, the corresponding leaf entry is deleted from the leaf-level page but no changes are made to the separator levels. Such a deletion can create a situation in which a search-key value in a separator entry has no corresponding value in a leaf entry. This would happen, for example, if *jane*'s row were deleted from Figure 9.16: the entry for *Jane* at the leaf level would be removed, but the entry at the separator level would remain. Such an index might seem strange, but it still functions correctly and does not represent a serious problem (other than a potential waste of space where the deallocated leaf entry resided).

Some systems take advantage of the static nature of an ISAM storage structure by placing pages on the disk so that the access time for a scan of the leaf level is minimized. The time to perform a range search can be minimized in this case.

A more serious problem arises when a new row is added since a new leaf entry must be created and the appropriate leaf page might be full. This can be avoided by using a fillfactor less than 1 (a fillfactor of .75 is reasonable for an ISAM file), but overflow pages might ultimately be necessary. For example, if a row for *ivan* were inserted (and the row for *jane* deleted), the leftmost subtree of the resulting index would be as shown in Figure 9.17. Note that the new page is an overflow of a leaf-level page, *not* a new level or even a new leaf-level page. If a table is dynamic, with frequent insertions, overflow chains can become long and, as a result, the index structure becomes less and less efficient because the overflow chains must be searched to satisfy queries. The entries on the chains might not be ordered, and the

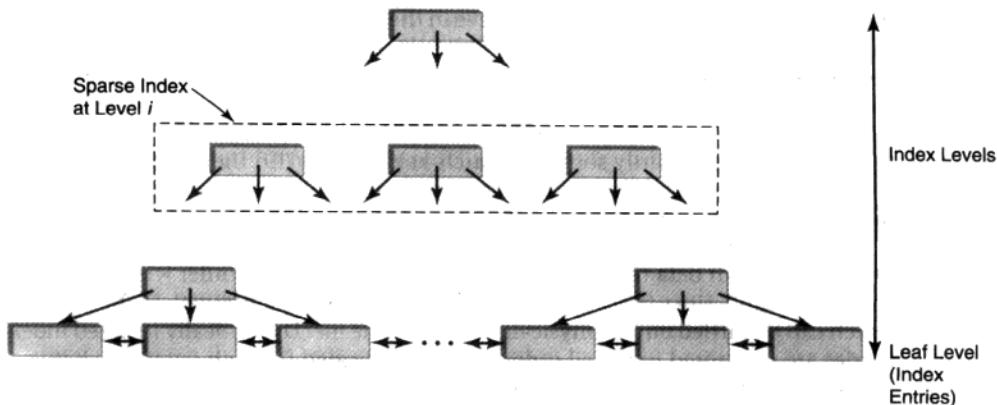


FIGURE 9.18 Schematic view of a B<sup>+</sup> tree.

overflow pages might not be physically close to one another on the mass storage device. The index can be reconstructed periodically to eliminate the chains, but this is expensive. For this reason, although an ISAM index can be effective for a relatively static table, it is generally not used when the table is dynamic.

### 9.5.2 B<sup>+</sup> Trees

A B<sup>+</sup> tree is the most commonly used index structure. Indeed, in some database systems it is the only one available. As with an ISAM index, the B<sup>+</sup> tree structure is based on the multilevel index and supports equality, range, and partial-key searches. Leaf pages might contain data records, in which case the B<sup>+</sup> tree acts not only as an index but as a storage structure that organizes the placement of records in the data file. Or the tree might be stored in an index file in which the leaf pages contain pointers to the data records. In the first case, the B<sup>+</sup> tree is a main index similar to an ISAM index since it is clustered. In the second case, it might be a main or a secondary index, and the data file need not be sorted on its search key.

**Searching the tree.** Figure 9.18 shows a schematic diagram of a B<sup>+</sup> tree in which each page contains a sorted set of entries. Each separator page has the form shown in Figure 9.14, and a search through the separator levels is conducted using the technique described for an ISAM index: if we are searching for a record with search-key value  $k$ , we choose the  $i$ th separator entry, where  $k_i \leq k < k_{i+1}$ .

The only difference between Figure 9.18 and Figure 9.14 is the addition of **sibling pointers**, which link pages at the leaf level in such a way that the linked list contains the search-key values of the data records of the table in sorted order. In contrast to an ISAM index, the B<sup>+</sup> tree itself changes dynamically. As records are added and deleted, leaf and separator pages have to be modified, added, and/or deleted, and

hence leaf pages might not be consecutive in the file. However, the linked list enables the B<sup>+</sup> tree to support range searches. Once the leaf entry at one end of a range has been located using an equality search, the other leaf entries that contain search-key values in the range can be located by scanning the list. Note that this scheme works when the B<sup>+</sup> tree is used as either a main or a secondary index (in which case data records are not necessarily sorted in search-key order). With the addition of sibling pointers, Figure 9.16 becomes a B<sup>+</sup> tree (keep in mind that leaf-level pages might or might not contain data records).

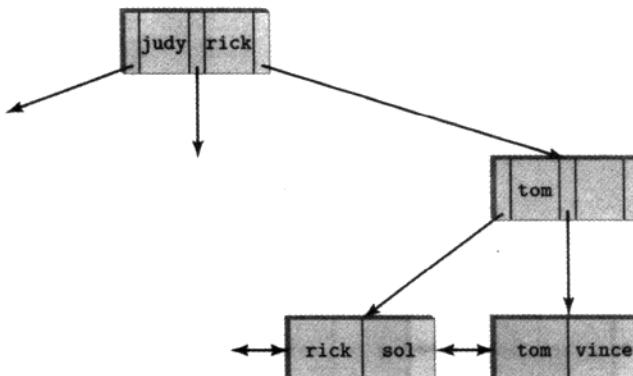
Sibling pointers are not needed in an ISAM index because leaf pages (which generally contain data records) are stored in the file in sorted order when the file is constructed and, since the index is static, that ordering is maintained. Hence, a range search can be carried out by physically scanning the file. Dynamically inserted index entries are not stored in sorted order but can be located through overflow chains.

B<sup>+</sup> trees are descendants of B trees. The main difference is that B trees can have pointers to the records in the data file *at any level*—not just at the leaf level. Thus, each index page can have a mixture of separator and leaf entries. This implies that a particular search-key value appears exactly once in the tree. A B<sup>+</sup> tree does not possess this property. Therefore, a B tree can be smaller than the corresponding B<sup>+</sup> tree and searching in a B tree can be a little faster. However, it is harder to organize pointers to the data records in a sorted fashion in such a tree since not all such pointers are stored in the leaves (note that adding sibling pointers to a B tree will be of little help here). Therefore, performing range searches in a B tree is trickier.

**Access cost.** The second difference between a B<sup>+</sup> tree and an ISAM index is that a B<sup>+</sup> tree is a **balanced tree**. This means that, despite the insertion of new records and the deletion of old records, any path from the root to a leaf page has the same length as any other. This is an important property. If the tree is unbalanced, then it is possible that the path from the root to a leaf page becomes very long, the I/O cost of accessing index entries in that page is large, and the index becomes the problem rather than the solution.

With a balanced tree the I/O cost of retrieving a particular leaf page is the same for all leaves. We have seen that, for multilevel indices with a reasonable fan-out, this cost can be surprisingly small.  $\Phi$  is the maximum number of separator entries that can be fit in an index page. If we assume that the algorithms for inserting and deleting entries (to be described shortly) ensure that the minimum number of separators stored in a page is  $\Phi/2$  (i.e.,  $\Phi/2$  is the minimum fan-out),<sup>8</sup> then the maximum cost of a search through a B<sup>+</sup> tree having  $Q$  leaf pages is  $\log_{\Phi/2} Q + 1$ . (In general, the root node can have fewer than  $\Phi/2$  separators, which affects this formula slightly.) Contrast this with an ISAM index with overflow chains. Because the length of a chain is unbounded, the cost of retrieving a leaf page at the end of a chain is also unbounded.

<sup>8</sup> In case  $\Phi$  is odd, the minimum number of separators should really be  $\lceil \Phi/2 \rceil$ —the smallest integer greater than  $\Phi/2$ .

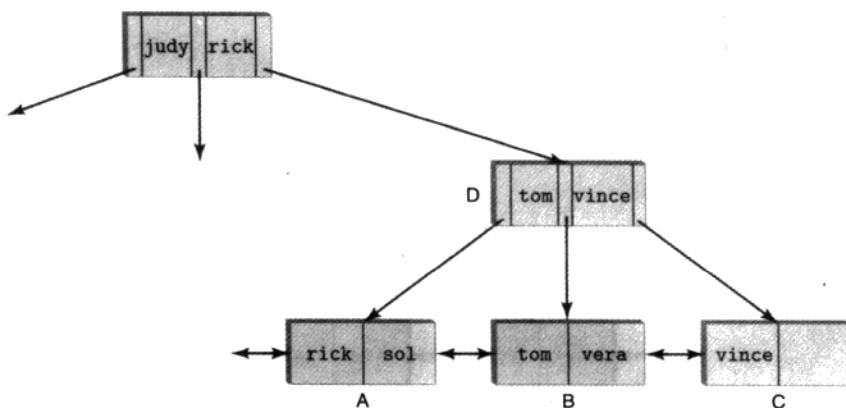


**FIGURE 9.19** Portion of the index of Figure 9.16 after insertion of an entry for `vince`.

**Inserting new entries.** A major advantage of the  $B^+$  tree over an ISAM index is its adaptation to dynamically changing tables. Instead of creating an overflow chain when a new record is added, we modify the tree structure so that it remains balanced. As the structure changes, entries are added to and deleted from pages and the number of separators in a page varies between  $\Phi/2$  and  $\Phi$ .

**Example 9.5.2 ( $B^+$  Tree).** Let us trace a sequence of insertions into the index of Figure 9.16, viewing that index as a  $B^+$  tree. In this case,  $\Phi = 2$ . Figure 9.19 shows the rightmost subtree after a record for `vince` has been added. Since `vince` follows `tom` in search-key order and there is room for an additional leaf entry in the rightmost leaf page, no modification to the  $B^+$  tree structure is required.

Suppose that the next insertion is `vera`, which follows `tom`. Since the rightmost leaf page is now full, a new page is needed, but instead of creating an overflow page (as in an ISAM index), we create a new leaf page. This requires modifying the structure of the index, which sets the  $B^+$  tree solution apart from the ISAM solution. Because the ordering of search-key values at the leaf level must be preserved, `vera` must be inserted between `tom` and `vince`. Hence, it is not sufficient simply to create a new rightmost leaf page containing `vera`. Instead, we must allocate a new leaf page and split the search-key values in sorted order between the existing leaf page and the new page so that roughly half are stored in each. The result is shown in Figure 9.20. The smallest entry in the new leaf page, labeled C in the figure, is `vince`, so `vince` becomes a new separator in index page D (all entries in leaf page B are less than `vince`), which fortunately has enough room for that entry. In general, when a (full) leaf page containing  $\Phi$  entries is split to accommodate an insertion, we create two leaf pages—one containing  $\Phi/2 + 1$  entries and the other containing  $\Phi/2$  entries—and we insert a separator at the next index level up. We refer to this as Rule 1. Note that we have both a separator and a leaf entry for `vince`.

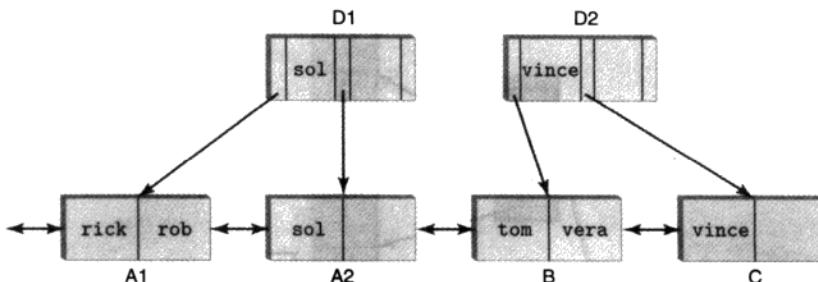


**FIGURE 9.20** Index subtree of Figure 9.19 after the insertion of *vera* has caused the split of a leaf page.

If the next insertion is *rob*, the problem is more severe. The new index entry must lie between *rick* and *sol*, requiring a split of page A, and four pointers are needed at the separator level (to refer to the two pages that follow from the split as well as B and C) in page D. Unfortunately, a separator page can accommodate only three pointers, and therefore we must split D as well. Furthermore, following Rule 1, the new separator value is *sol* (since it will be the smallest search-key value in the new leaf page). In the general case, an index page is split when it has to store  $\Phi + 1$  separators (in this case *sol*, *tom*, and *vince*) and thus  $\Phi + 2$  pointers to index pages at the next lower level. Each page that results from the split contains  $\Phi/2 + 1$  pointers and  $\Phi/2$  separators. It might seem that we have misplaced a separator, but we are not done yet.

The situation after the split of page A into A1 and A2 and page D into D1 and D2 is shown in Figure 9.21. Note that the total number of separator entries in pages D1 and D2—two—is the same as in page D, although the values are different: *sol* has replaced *tom*. This number seems strange, since the number of separators required to separate the pages at the leaf level is three (*sol*, *tom*, and *vince*). The explanation is that *tom*, the separator that separates the values contained in the two subtrees rooted at D1 and D2, becomes a separator at a higher level. In general, in splitting a page at the separator level to accommodate  $\Phi + 1$  separators, the middle separator in the separator sequence is not stored in either of the two separator pages resulting from the split, but instead is *pushed up* the tree. We refer to this pushing as Rule 2.

Hence, we are not finished. The separator has to be pushed and a reference has to be made to the new separator page (D2) at the next-higher index level. In other words, the process has to be repeated. In general, the process has to be repeated until an index level is reached that can accommodate a new separator without requiring a split. In our example, the next index level is the root page, and since it cannot



**FIGURE 9.21** Index subtree of Figure 9.20 after the insertion of `rob` has caused the split of a leaf page and of a separator page.

accommodate another separator, it will have to be split. In this case, the sequence of separators that we are dealing with is `judy`, `rick`, and `tom`. Using Rule 2, `rick` must be pushed up to be stored in a new root page.

This completes the process and yields the  $B^+$  tree shown in Figure 9.22. Note that the number of levels has increased by 1 but that the tree remains balanced. Four I/O operations are now required to access any leaf page. If the table is accessed frequently, the number of I/O operations might be reduced, keeping one or more of the upper levels of the tree in main memory. Also note that, in splitting A, the sibling pointer in B must be updated, which requires an additional I/O operation. ■

A node split incurs overhead and should be avoided, if possible. One way to do this is to use a fillfactor that is less than 1 when the  $B^+$  tree is created. A fillfactor of .75 is reasonable (however, if a table is read-only the fillfactor should be 1). Of course, this might increase the number of levels in the tree. One variation of the insertion algorithm that attempts to avoid a split involves redistributing index entries in the leaf-level pages. For example, if we insert a leaf entry for `tony`, in Figure 9.20 we might make room in page B by moving the entry for `vera` to page C (and replacing `vince` with `vera` as the separator in page D). Redistribution is generally done at the leaf level between neighboring pages that have the same immediate parent. Such pages are referred to as *siblings*.

The above discussion explains the main points of the process of inserting an entry into a  $B^+$  tree. Exercise 9.12 asks you to formalize this process in an actual algorithm.

**Deleting entries.** Deletion presents a different problem. When pages become sparsely occupied, the tree can become deeper than necessary, increasing the cost of each search down the tree and each scan across the leaf level. To avoid such situations, pages can be compacted. A minimum occupancy requirement of  $\Phi/2$  entries per page is set for this purpose. When a deletion is made from a page,  $p$ , with  $\Phi/2$

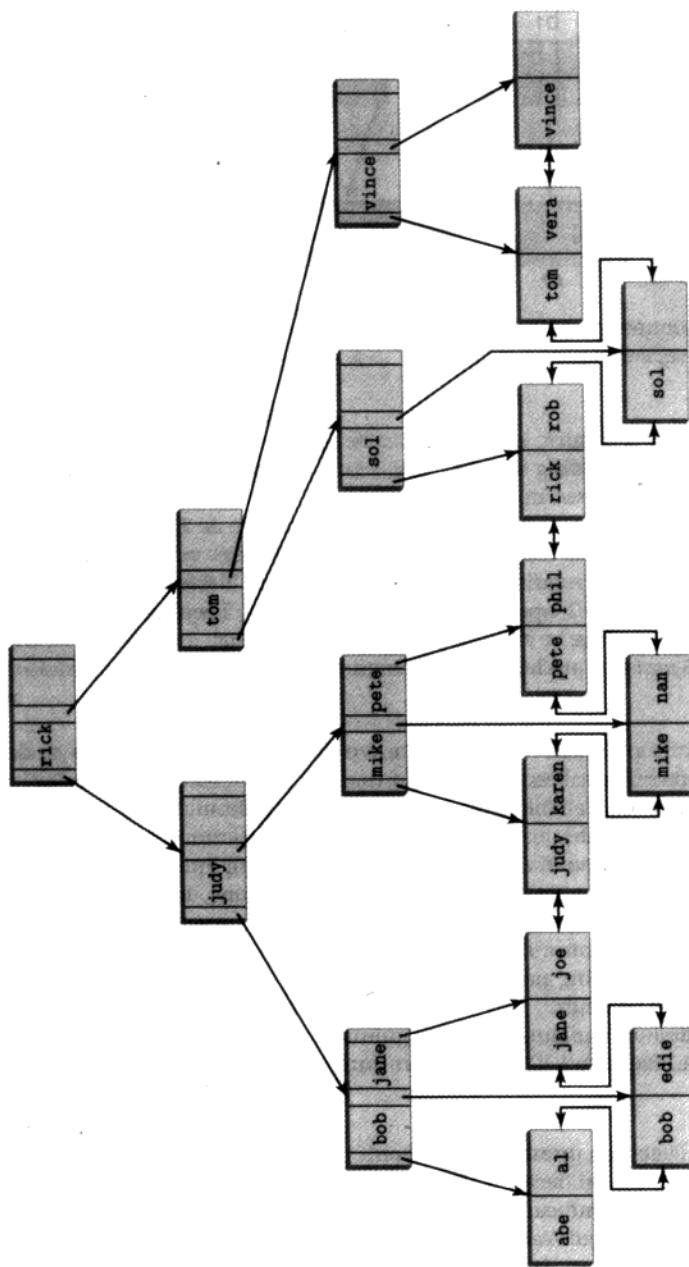
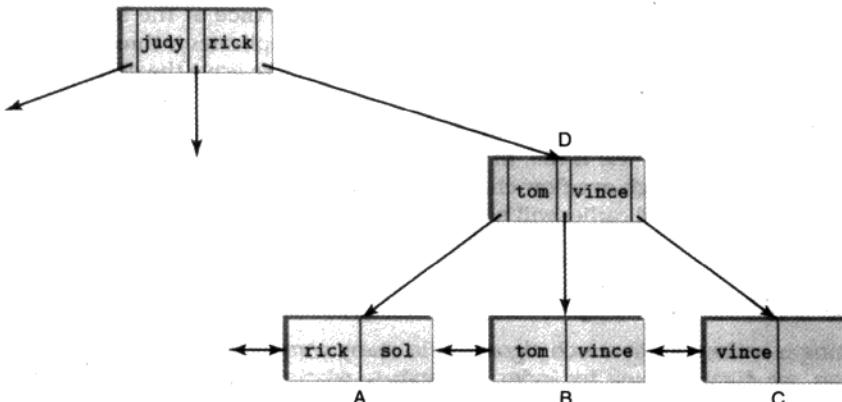


FIGURE 9.22  $B^+$  tree that results from the insertion of vince, vera, and rob into the index of Figure 9.16.



**FIGURE 9.23** Index subtree of Figure 9.19 after the insertion of a duplicate entry for *vince* has caused the split of a leaf page.

entries, an attempt is first made to redistribute entries from one of  $p$ 's siblings. This is not possible if both siblings also have  $\Phi/2$  entries. In this case  $p$  is merged with a sibling:  $p$  and its sibling are deleted and replaced with a new page containing the  $\Phi - 1$  entries previously stored in the deleted pages. Furthermore, a separator entry is deleted from the parent node in the next-higher level of the tree (recall that siblings have a common parent). Just as the effect of a split can propagate up the tree, so too can the effect of a merge. The deletion can cause the parent page to fall below the threshold level, requiring separators to be redistributed or pages to be merged. If the effect propagates up to the root and the last separator in the root is deleted, the depth of the tree is reduced by 1. In Exercise 9.13 you will formalize in an algorithm the process just described.

Since tables tend to grow over time, some database systems do not enforce the minimum occupancy requirement but simply delete pages when they become empty. If necessary, a tree can be completely reconstructed to eliminate pages that do not meet the minimum occupancy requirement.

**Multiple identical search-key values.** Although we ignored this possibility in our previous discussion, when the search key is not a candidate key of the table, multiple rows might have the same search-key value. Suppose, for example, that a record for another student named *vince* is added to the  $B^+$  tree of Figure 9.19. Using Rule 1, the rightmost leaf node must then be split to accommodate a second index entry with search-key value *vince*, and a new separator must be created, as shown in Figure 9.23. The first thing to note is that the search-key values in page B are no longer strictly less than the value *vince* in the separator entry in D. The second is that a search for *vince* terminates in leaf page C and therefore does not find the index entry for the other *vince* in B. Finally, if rows for additional students named

vince are inserted, there will be several separators for vince at the lowest (and perhaps a higher) separator level. One way of handling duplicates is thus to modify the search algorithm to accommodate these differences. We leave the details of the modification to an exercise.

Another approach to handling the insertion of a duplicate is simply to create an overflow page if the leaf is full. In this way, the search algorithm does not have to be modified, although overflow chains can grow large and the cost estimates for using the tree described earlier will no longer apply.

## 9.6 Hash Indexing

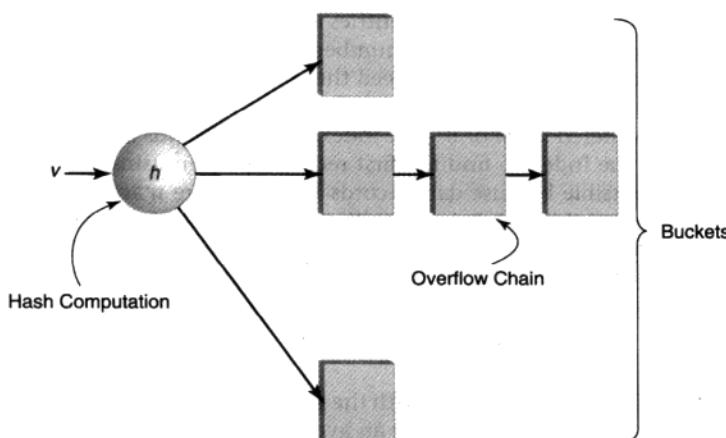
Hashing is an important search algorithm in many computer applications. In this section, we discuss its use for indexing database relations, looking at both **static hashing**, where the size of the hash table stays constant, and **dynamic hashing**, where the table may grow or shrink. The first technique is superior when the contents of a relation are more or less stable; the second is superior when indexing relations that are subject to frequent inserts and deletes.

### 9.6.1 Static Hashing

A hash index divides the index entries corresponding to the data records of a table into disjoint subsets, called **buckets**, in accordance with some **hash function**,  $h$ . The particular bucket into which a new index entry is inserted is determined by applying  $h$  to the search-key value,  $v$ , in the entry. Thus,  $h(v)$  is the address of the bucket. Since the number of distinct search-key values is generally much larger than the number of buckets, a particular bucket will contain entries with different search-key values. Each bucket is generally stored in a page (which might be extended with an overflow chain), identified by  $h(v)$ . The situation is shown in Figure 9.24.

As with a tree index, an index entry in a hash index might contain the data record or might store a pointer to the data record in the data file. If it contains the data record, the buckets serve as a storage structure for the data file itself: the data file is a sequence of buckets. If it contains a pointer to the record, the index entries are stored as a sequence of buckets in an index file and the records in the data file can be stored in arbitrary order. In this case, the hash index is secondary and unclustered. If the data records referred to by the index entries in a particular bucket are grouped together in the data file, the hash index is clustered. This implies that data records with the same value of the search key are physically close to each other on mass storage.

**Hash search.** An equality search for index entries with search key  $v$  is carried out by computing  $h(v)$ , retrieving the bucket stored in the referenced page, and then scanning its contents to locate the index entries (if any) containing  $v$ . Since no other bucket can possibly contain entries with that key value, if the entry is not found in the bucket, it is not in the file. Thus, without having to maintain an index



**FIGURE 9.24** Schematic depiction of a hash index.

structure analogous to a tree, the target index entry of an equality search can be retrieved with a single I/O operation (assuming no overflow chain).

A properly designed hash index can perform an equality search more efficiently than a tree index can, since, with a tree index, several index-level pages must be retrieved before the leaf level is reached. If a succession of equality searches must be performed, however, a tree index might be preferable. Suppose, for example, that it is necessary to retrieve the records with search-key values  $k_0, k_1, \dots, k_l$ , that the sequence is ordered on the search key (i.e.,  $k_i < k_{i+1}$ ), and that the sequence represents the order in which records are to be retrieved. Because the index leaves of a tree index are sorted on the key, the likelihood of cache hits when retrieving index entries is great. But with a hash index, each search-key value might hash to a different bucket, so the retrieval of successive index entries in the sequence might not generate cache hits. The example shows that, in evaluating which index might improve an application's performance, the entire application should be considered.

Despite the apparent advantage that hash indices have for equality searches, tree indices are generally preferable because they are more versatile: hash indices cannot support range or partial-key searches. A partial-key search is not supported because the hash function must be applied to the entire key. To understand why a range search cannot be supported efficiently, consider the student table discussed earlier. Suppose that we want to use a hash index to retrieve the records of all individuals in the file with names between paul and tom. Hashing can determine that there are no individuals with the name paul, and it can retrieve the index entry for tom, but it is of no help in locating index entries for individuals with names inside the range because we have no recourse but to apply the hash function to every *possible* value in the range—only a few of which are likely to appear in the database. Successive entries in the range are spread randomly through the buckets, so the cost of evaluating such

a range query is proportional to the number of entries in the range. In the worst case (when the number of such entries exceeds the number of pages in the file), the cost associated with the use of the index might exceed the cost of simply scanning the entire file!

In contrast, with a clustered ISAM or B<sup>+</sup> tree index the data records in the range can be located by using the index to find the first record and then using a simple scan. A simple scan is possible because data records in the file are maintained in search-key order. The cost of this search is proportional to the number of pages in the range plus the cost of searching the index. The important difference is that the I/O cost of a B<sup>+</sup> tree search depends on the number of *leaf pages* in the range, while the cost of a hash index search depends on the number of *records* in the range, which is much larger.

**Hash functions.** Hash functions are chosen with the goal of randomizing the index entries over the buckets in such a way that, for an average instance of the indexed table, the number of index entries in each bucket is roughly the same. For example,  $h$  might be defined as

$$h(v) = (a * v + b) \bmod M$$

where  $a$  and  $b$  are constants chosen to optimize the way the function randomizes over the search-key values,  $M$  is the number of buckets, and  $v$  is a search-key value treated as a binary number in the calculation of the hash value. Note that, with some applications, no matter how clever the hash function, it might be impossible to keep the population of a bucket close to the average: if the search key is not a candidate key a large fraction of the rows might have the same search-key value and hence will necessarily reside in the same bucket. For simplicity, we assume  $M$  to be a power of 2 in all of the algorithms that follow (but in practice it is usually a large prime number).

The indexing scheme we have just described is referred to as a static hash because  $M$  is fixed when the index is created. With static hashing, the location mechanism is the hash function—no data structures are involved in this case. The efficiency of static hashing depends on the assumption that all entries in each bucket fit in a single page. The number of entries in a bucket is inversely proportional to  $M$ : if fewer buckets are used, the average bucket occupancy is larger. The larger the average occupancy, the more unlikely it is that a bucket will fit in a page and so overflow pages will be needed. The choice of  $M$  is thus crucial to the index's efficient operation.

If  $\Phi$  is the maximum number of index entries that can fit in a page and  $L$  is the total number of index entries, choosing  $M$  to be  $L/\Phi$  leads to buckets that overflow a single page. For one thing,  $h$  does not generally divide entries exactly evenly over the buckets, so we can expect that more than  $\Phi$  entries will be assigned to some buckets. For another, bucket overflow results if the table grows over time. One way to deal with this growth is to use a fillfactor less than 1 and enlarge the number of buckets. The average bucket occupancy is chosen to be  $\Phi * \text{fillfactor}$  so that buckets with larger than average populations can be accommodated in a single page.  $M$  then becomes

$L/(\Phi * \text{fillfactor})$ . Fillfactors as low as .5 are not unreasonable. The disadvantage of enlarging  $M$  is that space requirements increase because some buckets have few entries.

This technique reduces, but does not solve, the overflow problem, particularly since the growth of some tables cannot be predicted in advance. Therefore, bucket overflow must be dealt with, and this can be done with overflow chains as shown in Figure 9.24. Unfortunately, as with an ISAM index, overflow chains can be inefficient. Because an entire bucket must be scanned with each equality search, a search through a bucket stored in  $n$  pages costs  $n$  I/O operations. This multiplies the cost of a search by  $n$  over the ideal case. Fortunately, studies have shown that  $n = 1.2$  for a good hash function.

### 9.6.2 Dynamic Hashing Algorithms

Just as  $B^+$  trees are adapted from tree indices to deal with dynamic tables, dynamic hashing schemes are adapted from static hashing to deal with the same problem. The goal of dynamic hashing is to change the number of buckets dynamically in order to reduce or eliminate overflow chains as rows are added and deleted. Two dynamic hashing algorithms that have received the most attention are *extendable hashing* and *linear hashing*. We briefly discuss them here.

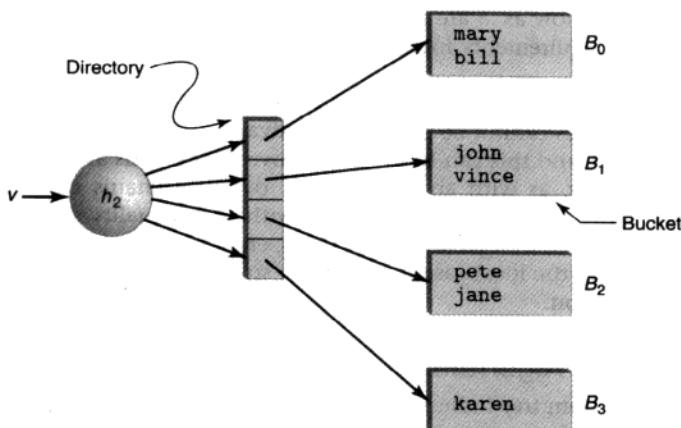
Static hashing uses a fixed hash function,  $h$ , to partition the set of all possible search-key values into subsets,  $S_i$ ,  $1 \leq i \leq M$ , and maps each subset to a bucket,  $B_i$ . Each element,  $v$ , of  $S_i$  has the property that  $h(v)$  identifies  $B_i$ . Dynamic hashing schemes allow  $S_i$  to be partitioned at run time into disjoint subsets,  $S'_i$  and  $S''_i$ , and  $B_i$  to be split into  $B'_i$  and  $B''_i$ , such that the elements of  $S'_i$  are mapped to  $B'_i$  and the elements of  $S''_i$  are mapped to  $B''_i$ . By reducing the number of values that map to a bucket, a split has the potential of replacing one overflowing bucket with two that are not full. A change in the mapping implies a change in the hash function that takes into account the split of a single bucket (or the merge of two buckets). Extendable and linear hashing do this mapping in different ways.

**Extendable hashing.** Extendable hashing uses a sequence of hash functions,  $h_0, h_1, \dots, h_b$ , based on a single hash function,  $h$ , which hashes a search-key value into a  $b$ -bit integer. For each  $k$ ,  $0 \leq k \leq b$ ,  $h_k(v)$  is the integer formed by the last  $k$  bits of  $h(v)$ . Stated mathematically,

$$h_k(v) = h(v) \bmod 2^k$$

Thus, the number of elements in the range of each function,  $h_k$ , is twice that of its predecessor,  $h_{k-1}$ . At any given time, a particular function in the sequence,  $h_k$ , directs all searches.

Unlike static hashing, dynamic hashing uses a second stage of mapping to determine the bucket associated with some search-key value,  $v$ . This mapping uses a level of indirection implemented through a directory, as shown in Figure 9.25. The value produced by  $h_k(v)$  serves as an index into the directory, and the pointer in the directory entry refers to the bucket associated with  $v$ . The key point is that distinct



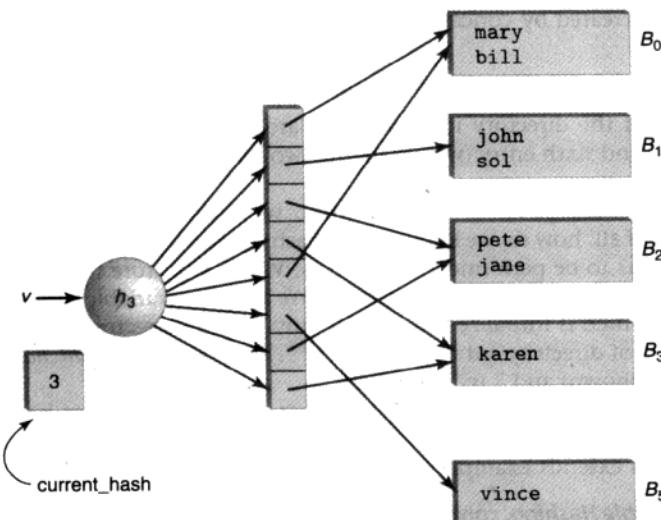
**FIGURE 9.25** With extendable hashing, the hash result is mapped to a bucket through a directory.

entries in the directory might refer to the same bucket, so  $v_1$  and  $v_2$  might be mapped to the same bucket even though  $h_k(v_1)$  and  $h_k(v_2)$  are distinct. Example 9.6.1 shows how this situation might arise.

**Example 9.6.1 (Extendable Hashing).** Consider an extendable hash index of student names as shown in Figure 9.25, and suppose that the range of  $h$  is the set of integers between 0 and  $2^{10} - 1$ . We assume that a bucket page can hold two index entries. The figure depicts the algorithm at a point at which  $h_2$  is used to hash search-key values (and, thus, the directory contains  $2^2 = 4$  entries). Since  $h_2$  uses only the last two bits of the values in the range of  $h$ , the figure could be produced by the following function  $h$ :

$v$	$h(v)$
pete	1001111010
mary	0100000000
jane	1100011110
bill	0100000000
john	0001101001
vince	1101110101
karen	0000110111

Thus,  $h$  hashes pete to 1001111010. Since  $h_2$  uses only the last two bits (10), it indicates that the third directory entry refers to the bucket,  $B_2$ , that contains pete.



**FIGURE 9.26** Bucket  $B_1$  of Figure 9.25 has been split using extendable hashing.

Suppose that we now insert a record for `sol` into the table and that  $h(\text{sol}) = 0001010001$ .  $h_2$  maps `john`, `vince`, and `sol` to  $B_1$ , causing an overflow. Rather than create an overflow chain, extendable hashing splits  $B_1$  so that a new bucket,  $B_5$ , is created, as shown in Figure 9.26. To accommodate five buckets, it is necessary to use a hash function whose range contains more than four values, so the index replaces  $h_2$  with  $h_3$ . Since the high-order bit of  $h_3(\text{john})$ , 0, and  $h_3(\text{vince})$ , 1, differ (whereas the two low-order bits, 01, are the same),  $h_3$  avoids overflow by mapping `john` and `vince` to different buckets ( $h_2$  does not do this). In Figure 9.26, note that if both  $v1$  and  $v2$  are elements of  $B_1$  (or  $B_5$ ),  $h(v1)$  and  $h(v2)$  agree in their last three bits.

The directory is needed to compensate for the fact that only  $B_1$  has been split. Indeed,  $h_3$  not only distinguishes between `john` and `vince`, but also produces different values for `pete` (010) and `jane` (110). Hence, without a directory it would be necessary to split  $B_2$  as well when  $h_2$  is replaced by  $h_3$ . By interposing a directory between the hash computation and the buckets, we can map both `pete` and `jane` to  $B_2$ , which we do by storing a pointer to  $B_2$  in both the third (010) and seventh (110) directory entries. As a result, in contrast to  $B_1$  and  $B_5$ , if  $v1$  and  $v2$  are elements of  $B_2$ , the values of the third bit of  $h(v1)$  and  $h(v2)$  might differ (but the values of the first two bits must be 10). ■

The algorithm used in moving from Figure 9.25 to Figure 9.26 is quite simple.

1. A new bucket,  $B'$ , is allocated and the contents of the overflowing bucket,  $B$ , are split between  $B$  and  $B'$  using the next hash function in the sequence.

2. A new directory is created by concatenating a copy of the old directory with itself.
3. The pointer to  $B$  in the copy is replaced by a pointer to  $B'$ .

Thus, the two halves of the directory in Figure 9.26 are identical except for the pointers in the second and sixth entry (which are the two halves of the bucket that has been split).

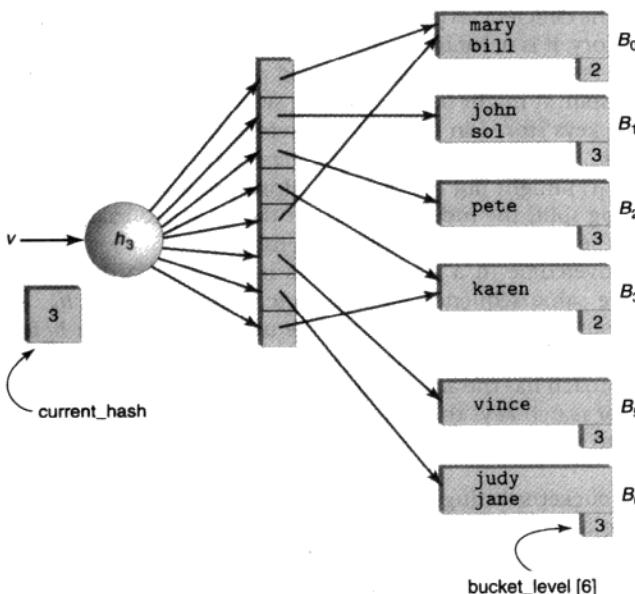
To give a complete description of the algorithm, we must deal with a few additional issues. First of all, how do we know which hash function in the sequence to use when a search has to be performed? That's easy. We simply store the index of the current hash function along with the directory. We assume a variable *current\_hash* for this purpose, which is initialized to 0 (with only a single directory entry). In general, the number of directory entries is  $2^{\text{current\_hash}}$ . The value of *current\_hash* is 2 in Figure 9.25 (not shown) and 3 in Figure 9.26.

A more subtle problem arises when an overflow occurs, but the current hash function can handle the new bucket that must be created. The situation is illustrated in Example 9.6.2, which extends Example 9.6.1.

**Example 9.6.2 (Extendable Hashing, continued).** Suppose a row for judy is inserted into the hash table shown in Figure 9.26 and  $h(\text{judy}) = 1110000110$ . judy is mapped to  $B_2$  in Figure 9.26, causing the bucket to overflow and requiring it to be split. However, this situation is different from the one that caused us to replace  $h_2$  with  $h_3$ , since  $h_3$  itself is capable of distinguishing among index entries that are mapped to  $B_2$  (recall that the hash values of the index entries in  $B_2$  agree in only their last two bits). In the example,  $h_3$  distinguishes judy and jane (110) from pete (010), so, instead of moving to a new hash function to deal with the overflow, we need only create a new bucket for judy and jane and update the appropriate pointer in the directory to refer to it. This is shown in Figure 9.27, in which the new bucket is labeled  $B_6$ . ■

There is a reason that the directory does not have to be enlarged this time. Each time it is enlarged, it becomes capable of storing pointers to accommodate the split of *every* bucket that exists at that time, whereas, in fact, only *one* bucket is actually split. Thus, when a directory is extended, all but one of the pointers in its new portion simply point back to an existing bucket. The directory in Figure 9.26 was created to accommodate the split of  $B_1$ , so only entry 101 in the new portion of that directory refers to a new bucket. Since entries 010 and 110 both point to  $B_2$ , the directory can accommodate a split of  $B_2$  without further enlargement.

We can detect this case by storing, along with each bucket, the number of times it has been split. We refer to this value as the *bucket level* and associate a variable, *bucket\_level[i]* to store it. Initially *bucket\_level[0]* is 0 and, when  $B_i$  is split, we use *bucket\_level[i] + 1* as the bucket level of both  $B_i$  and the newly created bucket. Because each time the directory is enlarged without splitting  $B_i$ , we double the number of pointers in the directory that refer to  $B_i$ ,  $2^{\text{current\_hash} - (\text{bucket\_level}[i])}$  is the number of pointers that point to  $B_i$  in the directory. Furthermore, when a bucket is split, the index entries are divided between the two resulting buckets so that, if  $v1$  and  $v2$  are both elements of  $B_i$ , then  $h(v1)$  and  $h(v2)$  agree in their last *bucket\_level[i]* bits.



**FIGURE 9.27** Bucket  $B_2$  of Figure 9.26 is split, without enlargement of the directory.

*bucket\_level[1]* is 3 in both Figure 9.26 (not shown) and Figure 9.27, whereas *bucket\_level[2]* is 2 in Figure 9.26 and is incremented to 3 in Figure 9.27.

The merge of two buckets can be handled using the inverse of the split algorithm. If the deletion of an index entry causes a bucket,  $B'$ , to become empty, and  $B'$  and  $B''$  were created when  $B$  was split,  $B'$  can be released by redirecting the pointer to  $B'$  in the directory so that it points to  $B''$  and decrementing the bucket level of  $B''$ . In addition, when merging creates a state in which the upper and lower halves of the directory are identical, one half can be released and *current\_hash* decremented. Merging is often not implemented since it is assumed that, although a table might temporarily shrink, it is likely to grow in the long term, and therefore a merge will ultimately be followed by a split.

Extendable hashing eliminates most of the overflow chains that develop with static hashing when the number of index entries grows. Unfortunately, though, it has several deficiencies. One is that additional space is required to store the directory. The other is that the indirection through the directory to locate the buffer requires additional time. If the directory is small, it can be kept in main memory, so neither of these deficiencies is major. In that case, directory access does not impose the cost of an additional I/O operation. Nevertheless, a directory can grow quite large and, if it cannot be kept in main memory, the I/O cost of extendable hashing is twice that of static hashing. Finally, splitting a bucket does not necessarily divide its contents and eliminate the overflow. For example, when the overflow is caused by multiple entries with the same search-key value, splitting cannot remove the overflow.

**Linear hashing.** Because the deficiencies of extendable hashing are associated with the introduction of a directory, it is natural to search for a dynamic hashing scheme for which a directory is not required. A directory is needed with extendable hashing because, when a bucket is split, it might be necessary to switch to a hash function with a larger range: search keys stored in different buckets must hash to different values. Thus, the range of  $h_{i+1}$  contains twice as many elements as the range of  $h_i$ , and therefore  $h_i(v)$  and  $h_{i+1}(v)$  might not be the same. However, if  $v$  is an element of a bucket that is not being split, the index must direct the search to that bucket before and after the split. The directory overcomes this problem.

Can this problem be overcome in a way that does not involve a directory? One possible way uses the same sequence of hash functions,  $h_0, h_1, \dots, h_b$ , as extendable hashing does, but differently.  $h_i$  and  $h_{i+1}$  are used concurrently:  $h_i$  for values belonging in buckets that have not been split and  $h_{i+1}$  for values belonging in buckets that have. This approach has the advantage of providing the same mapping before and after the split for search keys that belong in buckets not involved in the split. The trick is to be able to tell, when initiating a search through the index, which hash function to use.

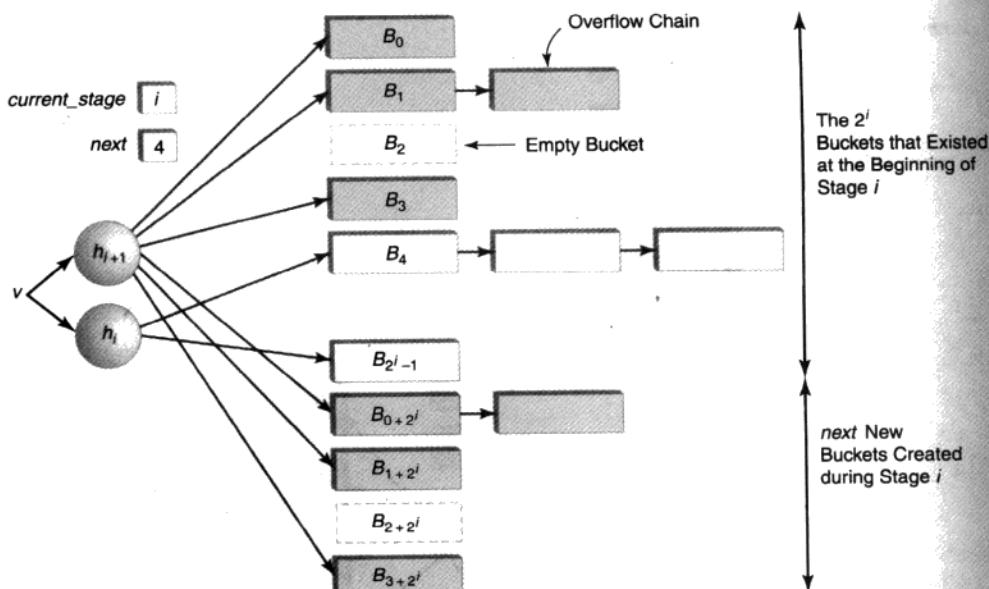
With **linear hashing**, bucket splitting is divided into stages. The buckets that exist at the start of a stage are split consecutively during the stage so that there are twice as many buckets at the end of the stage as there were in the beginning. The situation is shown in Figure 9.28. Stages are numbered, and the number of the current stage is stored in the variable *current\_stage*. Initially *current\_stage* is 0 and there is a single bucket. The variable *next* indicates the next bucket in the sequence to be split during the current stage.

In Figure 9.28 *current\_stage* has value  $i$ , indicating that it is in stage  $i$ . When stage  $i$  started there were  $2^i$  buckets and *next* was initialized to 0, indicating that no buckets have yet been split in this stage. Since no buckets have yet been split,  $h_i$ , with range  $\{0 \dots 2^i - 1\}$ , is used to hash search-key values for an index search. Once the algorithm starts splitting buckets during the stage, both  $h_i$  and  $h_{i+1}$  are used.

A decision is made periodically to split a bucket (we return to this point shortly), and the value of *next* is used to determine which bucket, among those that existed at the beginning of the stage, is to be split. *next* is incremented when the split occurs. Hence, buckets are split consecutively and *next* distinguishes those that have been split in the current stage from those that have not. Since *next* refers to the next bucket to be split, buckets with index less than *next* have been split in this stage. (Split buckets and their images are shaded in the figure.) Stage  $i$  is complete when the  $2^i$  buckets that existed when the stage began have been split.

The new bucket created when  $B_{next}$  is split has index  $next + 2^i$ . Elements of  $B_{next}$  are divided using  $h_{i+1}$ : if  $v$  was an element of  $B_{next}$ , it remains in that bucket if  $h_{i+1}(v) = h_i(v)$  and is moved to  $B_{next+2^i}$  otherwise. When the value of *next* reaches  $2^i$ , a new stage is started by resetting *next* to 0 and incrementing *current\_stage*.

When a search is to be performed on search key  $v$ ,  $h_i(v)$  is calculated. If  $next \leq h_i(v) < 2^i$ , the bucket indicated by  $h_i(v)$  is scanned for  $v$ . If  $0 \leq h_i(v) < next$ , the bucket indicated by  $h_i(v)$  has been split and  $h_i(v)$  does not provide enough information to

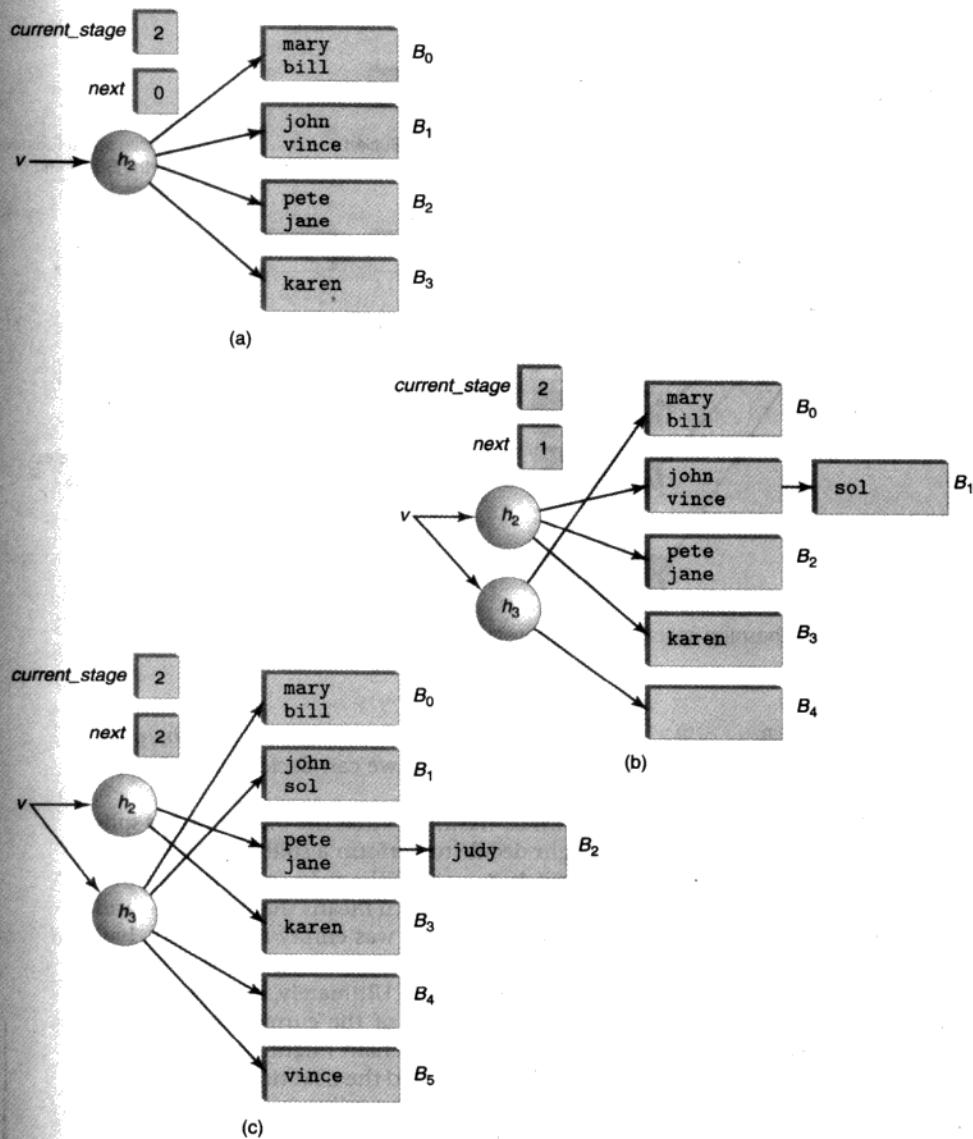


**FIGURE 9.28** Linear hashing splits buckets consecutively. The shaded buckets are accessed through  $h_{i+1}$ .

determine whether  $B_{h_i(v)}$  or  $B_{h_i(v)+2^i}$  should be scanned for  $v$ . However, since  $h_{i+1}$  was used to divide the elements when the split was made, we can decide which bucket to search using  $h_{i+1}(v)$ . Hence,  $v$  is rehashed using  $h_{i+1}$ .

The important point to note about linear hashing is that the bucket that is split has not necessarily overflowed. We might decide to perform a split when a bucket,  $B_j$ , overflows, but the bucket that is split is  $B_{next}$ , and the value of  $next$  might be different from  $j$ . Note that in Figure 9.28  $next = 4$ , which means that  $B_0, B_1, B_2$ , and  $B_3$  have been split. In particular,  $B_2$  was split when it was empty and, hence, had certainly not overflowed. Thus, linear hashing does not eliminate overflow chains (in this case, an overflow page must be created for  $B_j$ ). Ultimately, however,  $B_j$  will be split because every bucket that exists at the start of the current stage is split before the stage completes. So, although an overflow chain might be created for a bucket,  $B$ , it tends to be short and is normally eliminated the next time  $B$  is split. The average lifetime of an overflow page can be decreased by splitting more frequently, although the price for doing that is lower space utilization since buckets that have not overflowed are split during each stage.

**Example 9.6.3 (Linear Hashing).** Given the assumption that a split occurs each time an overflow page is created, Figure 9.29 shows the sequence of states that a linear index goes through when it starts in the same state as shown for extendable hashing



**FIGURE 9.29** (a) State of a linear hash index at the beginning of a stage; (b) state after insertion of *sol*; (c) state after insertion of *judy*.

in Figure 9.25 and experiences the same subsequent insertions. In Figure 9.29(a) we have assumed that the index has just entered stage 2. Figure 9.29(b) shows the result of inserting `sol` into  $B_1$ . Although  $B_1$  has overflowed,  $B_0$  is split since `next` has value 0. Note that, since both `mary` and `bill` are hashed to  $B_0$  by  $h_3$ ,  $B_4$  is empty. Figure 9.29(c) shows the result of inserting `judy` into  $B_2$ .  $B_1$  is now split, eliminating its overflow chain, but a new overflow chain is created for  $B_2$ . ■

It might seem that linear hashing splits the wrong bucket, but this is the price that is paid for avoiding the directory. Splitting buckets in sequence makes it easy to distinguish the buckets that have been split from those that have not. Although overflow chains exist with linear hashing, there is no additional cost for fetching the directory.

## 9.7 Special-Purpose Indices

The index structures introduced so far can be used in a variety of situations. However, there are a number of index structures that are applicable in only very special cases but can yield large savings in storage space and processing time. We consider two such techniques: bitmap and join indices.

### 9.7.1 Bitmap Indices

A **bitmap index** [O'Neil 1987] is implemented as one or more bit vectors. It is particularly appropriate for attributes that can take on only a small number of values—for example, the `Sex` attribute of the `PERSON` relation, which can take only two values: `Male` and `Female`. Suppose that `PERSON` has a total of 40,000 rows. A bitmap index on `PERSON` contains two bit vectors, one for each possible value of `Sex`, and each bit vector contains 40,000 bits, one for each row in `PERSON`. Thus, the  $i$ th bit in the `Male` bit vector is 1 if, in the  $i$ th row of `PERSON`, the `Sex` attribute has value `Male`. As a result, we can identify the row numbers of males by scanning the `Male` bit vector. Given the row number, we can calculate the offset from the beginning of the data file and find the desired row on disk by direct access.

Note that the space to store the index in our example is just 80,000 bits, or 10K bytes, which can fit handily in main memory. Because searching such an index is done entirely in main memory, it can be carried out quickly using sequential scan.

You may have noticed that bitmap indices seem to waste more space than they need to. Indeed, to encode the `Sex` attribute we use *two* bits, while only one bit suffices to encode all of its possible values. The reason for this is that bitmap indices are designed to trade space for efficiency, especially the efficiency of selecting on two or more attributes. To illustrate, suppose that our `PERSON` relation represents the result of a health survey and that there is an attribute `Smoker` and an attribute `HasHeartDisease`, both accepting just two values, `Yes` and `No`. One query might

request the number of males in a certain age group who smoke but do not have heart disease. As part of this query, we have a selection with the following condition:

---

```
Sex = 'Male' AND Smoker = 'Yes' AND HasHeartDisease = 'No'
```

---

If all three attributes have bit indices, we can easily find rids of all tuples that satisfy this condition: take the logical AND of the bit strings that correspond to the `Male` value of the `Sex` attribute, the `Yes` value of the `Smoker` attribute, and the `No` value of the `HasHeartDisease` attribute. The rids of the tuples that satisfy the condition correspond to the positions that have 1 in the resulting bit string.

More generally, bitmap indices can handle selection conditions that contain OR and NOT—all we have to do is compute an appropriate Boolean combination of the corresponding bit strings. For instance, suppose that the `Age` attribute has a bitmap index with 120 bit strings with a cost of 120 bits per `PERSON` record. This is quite acceptable, as a regular index will take at least 8 bytes (64 bits) per `PERSON` record anyway. Now we should be able to efficiently find all smoking males between the ages of 50 and 80 who never suffered from heart disease as follows: First compute the logical AND of the three bit strings as described above. Then compute the logical OR of the bit strings corresponding to ages 50 to 80 in the bitmap index for the `Age` attribute. Finally, compute the logical AND of the two results. Again, the 1s in the final bit string give us the IDs of the records we need to answer the query.

Bitmap indices play an important role in data mining and OLAP (online analytical processing) applications. The reason for their popularity is that such applications typically deal with queries that select on low-cardinality attributes, such as `sex`, `age`, company locations, financial periods, and the like. It also has to do with the fact that these applications operate on data that is fairly static, and bitmap indices are expensive to maintain if the underlying data changes frequently (think of what it takes to update a bitmap index if records are inserted or deleted in the data file).

In conclusion, we note that with a little more thought it is possible to devise a schema where only  $n - 1$  bit vectors would be needed to index an attribute that can take  $n$  different values. In particular, for Boolean attributes, such as `Sex`, only one bit vector is needed. We leave the development of such a schema to Exercise 9.25.

## 9.7.2 Join Indices

Suppose we want to speed up an equi-join of two relations, such as  $p \bowtie_{A=B} q$ . A **join index** [Valduriez 1987] is a collection that consists of all pairs of the form  $\langle p, q \rangle$ , where  $p$  is a rid of a tuple,  $t$ , in  $p$  and  $q$  is a rid of a tuple,  $s$ , in  $q$ , such that  $t.A = s.B$  (i.e., the two tuples join).

A join index is typically sorted lexicographically in ascending order of rids. Thus, the pair  $(3, 3)$  precedes the pair  $(4, 2)$ . Such an index can also be organized as a  $B^+$  tree or as a hash table. The values of the search key in the entries of a  $B^+$  tree are the rids of the rows in  $p$ . To find the rids of all rows of  $q$  that join with a row of  $p$  having rid  $p$ , one searches the tree using  $p$ . The pointers in the leaf entries (if they

exist) with search-key value  $p$  are the requested rids. Similarly, a hash index hashes on  $p$  to find the bucket containing the entries for that rid. Those entries contain the rids of the rows in  $q$  that join with the row in  $p$  at rid  $p$ .

A join index can be thought of as a precomputed join that is stored in compact form. Although its computation might require that a regular join be performed first, its advantage (apart from its compact size) is that it can be maintained incrementally: when new tuples are added to  $p$  or  $q$ , new index entries can be added to the join index without the entire join having to be recomputed from scratch (Exercise 9.24).

Given a join index,  $\delta$ , the system can compute  $p \bowtie_{A=B} q$  simply by scanning  $\delta$  and finding the rids of  $p$  and  $q$  that match. Note that, if  $\delta$  is sorted on its  $p$  field, the rids corresponding to the tuples in  $p$  appear in ascending order. Therefore, the join is computed in one scan of the index and of  $p$ . For each rid of a tuple in  $q$ , one access to  $q$  is also needed.

Worth mentioning is one other variation on the join index, called a **bitmapped join index** [O'Neil and Graefe 1995]. Recall that a join index is typically organized as a sorted file, a hash table, or a  $B^+$  tree in a way that makes it easy to find the rids of all rows of  $q$  that join with the row of  $p$  at rid  $p$ . We can combine these rids in  $q$  in the following (at first, unusual) way: for each rid  $p$  in  $p$ , replace all tuples of the form  $(p, q)$  in  $\delta$  with a single tuple of the form

$$(p, \text{bitmap for matching tuples in } q)$$

The bitmap here has 1 in the  $i$ th position if and only if the  $i$ th tuple in  $q$  joins with the tuple in  $p$  at the rid  $p$ . Attaching a bitmap might seem like a huge waste of space, because each bitmap has as many bits as there are tuples in  $q$ . However, bitmaps are easily compressed, so this is not a major issue.

## 9.8 Tuning Issues: Choosing Indices for an Application

Each type of index is capable of improving the performance of a particular group of queries. Hence, in choosing the indices to support a particular application it is important to know the queries that are likely to be executed and their approximate frequency. Creating an index to improve the performance of a rarely executed query is probably not wise since each index carries with it added overhead, particularly for operations that update the database.

For example, if query (9.2) on page 332, with the value of `StudId` specified as a parameter, is executed frequently, an index to ensure fast response might be called for. The search key for such an index is chosen from among the columns named in the `WHERE` clause (*not* the `SELECT` clause) because these are the columns that direct the search. In this case, an index on `TRANSCRIPT` with search key `StudId` is useful. Instead of scanning the entire table for rows in which the value of `StudId` is 111111111, the location mechanism finds the index entries containing the search-key value. Each entry contains either the record or the rid of the record. Thus, while a scan requires that the system retrieve (on average) half the pages in the data file,

access through the index requires only a single retrieval. On the other hand, an index on StudId is of no use in the execution of query (9.3) on page 333.

Suppose, however, that we need to support the query

---

```
SELECT T.Grade
FROM TRANSCRIPT T
WHERE T.StudId = '111111111' AND T.Semester = 'F1997'
```

---

9.5

We might choose to create a multiattribute index on StudId and Semester. If, however, we choose to limit ourselves to an index on a single attribute (perhaps to support other queries and avoid too many indices), which attribute should be its search key? Generally, the attribute that is most selective is chosen. If an index on StudId is created, the database system uses it to fetch all rows for the target student and then scans the result, retaining those rows for the target semester. This is more efficient than fetching all rows for the target semester since there are likely to be many more rows for a given semester than for a given student. In general, columns that have only a few values in their domain (for example, Sex) are not likely to be very selective.

The following points provide some guidance in the choice of a search key:

1. A column used in a join condition might be indexed.
2. A clustered B<sup>+</sup> tree index on a column that is used in an ORDER BY clause makes it possible to retrieve rows in the specified order.
3. An index on a column that is a candidate key makes it possible to enforce the unique constraint efficiently.
4. A clustered B<sup>+</sup> tree index on a column used in a range search allows elements in a particular range to be quickly retrieved.

A more complete discussion of tuning can be found in Chapter 12.

## BIBLIOGRAPHIC NOTES

B trees were introduced in [Bayer and McCreight 1972], and B<sup>+</sup> trees first appeared in [Knuth 1973]. The latest edition of that book, [Knuth 1998], contains much information on the material covered in this chapter. Hashing as a data structure was first discussed in [Peterson 1957]. Linear hashing was proposed in [Litwin 1980]; extendable hashing in [Fagin et al. 1979]. Index sequential files were analyzed in [Larson 1981].

Join indices were first proposed in [Valduriez 1987], and bitmap indices were first described in [O'Neil 1987]. Bitmapped join indices were introduced in [O'Neil and Graefe 1995].

## EXERCISES

- 9.1 State the storage capacity, sector size, page size, seek time, rotational latency, and transfer time of the disk
- a. On your local PC
  - b. On the server provided by your university
- 9.2 Explain the difference between an equality search and a range search.
- 9.3
  - a. Give an upper bound on the number of pages that must be fetched to perform a binary search for a particular name in the phone book for your city or town.
  - b. By how much is this number reduced if an index is prepared giving the name of the first entry on each page of the phone book? (Does that index fit on one page of the phone book?)
  - c. Conduct an experiment using your usual (informal) method to search for the name "John Lewis" in your local phone book, and compare the number of pages you look at with the number in Exercise 3(a).
- 9.4 Explain why a file can have only one clustered index.
- 9.5 Explain why a secondary, unclustered index must be dense.
- 9.6 Does the final structure of a B<sup>+</sup> tree depend on the order in which the items are added to it? Explain your answer and give an example.
- 9.7 Starting with an empty B<sup>+</sup> tree with up to two keys per node; show how the tree grows when the following keys are inserted one after another:

18, 10, 7, 14, 8, 9, 21

- 9.8 Consider the partially specified B<sup>+</sup> tree in Figure 9.30.
- a. Fill in the internal nodes without adding new keys.
  - b. Add the key bbb. Show how the tree changes.
  - c. Delete the key abc from the result of (b). Show how the tree changes.

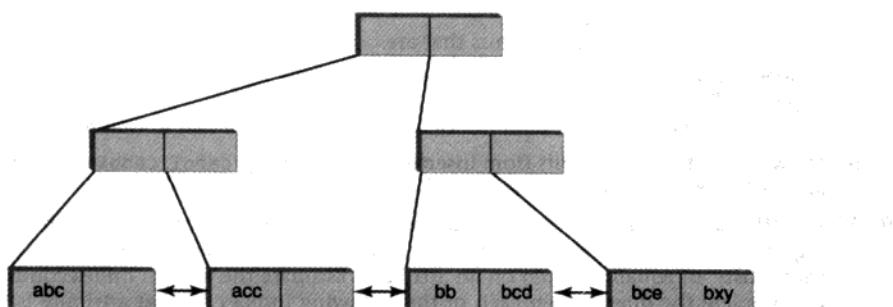
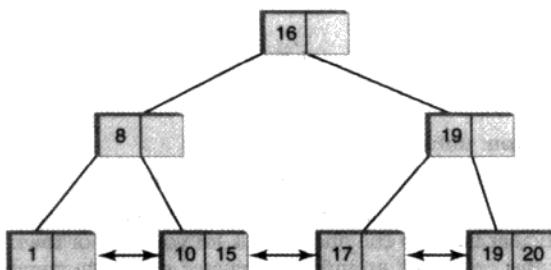


FIGURE 9.30 Partially specified B<sup>+</sup> tree.

FIGURE 9.31 B<sup>+</sup> tree.

- 9.9 Consider the B<sup>+</sup> tree in Figure 9.31. Suppose that it was obtained by inserting a key into a leaf node of some other tree, *causing a node split*. What was the original tree and the inserted key? Is the solution unique? Explain your answer.
- 9.10 Describe a search algorithm for a B<sup>+</sup> tree in which the search key is not a candidate key. Assume that overflow pages are not used to handle duplicates.
- 9.11 Consider a hash function,  $h$ , that takes as an argument a value of a composite search key that is a sequence of  $r$  attributes,  $a_1, a_2, \dots, a_r$ . If  $h$  has the form
- $$h(a_1 \circ a_2 \circ \dots \circ a_r) = h_1(a_1) \circ h_2(a_2) \circ \dots \circ h_r(a_r)$$
- where  $h_i$  is a hash of attribute  $a_i$  and  $\circ$  is the concatenation operator,  $h$  is referred to as a **partitioned hash function**. Describe the advantages of such a function with respect to equality, partial-key, and range searches.
- 9.12 Express the algorithm for insertion into a B<sup>+</sup> tree using pseudocode.
- 9.13 Express the algorithm for deletion from a B<sup>+</sup> tree using pseudocode.
- 9.14 Express the algorithm for insertion and deletion of index entries in the extendable hashing schema using pseudocode.
- 9.15 Give examples of select statements that are
- Speeded up due to the addition of the B<sup>+</sup> tree index shown in Figure 9.22 on page 358.
  - Slowed down due to the addition of the B<sup>+</sup> tree index shown in that figure.
- 9.16 Draw the B<sup>+</sup> tree that results from inserting `alice`, `betty`, `carol`, `debbie`, `edith`, and `zelda` into the index of Figure 9.22 on page 358.
- 9.17 A particular table in a relational database contains 100,000 rows, each of which requires 200 bytes of memory. A SELECT statement returns all rows in the table that satisfy an equality search on an attribute. Estimate the time in milliseconds to complete the query when each of the following indices on that attribute is used. Make realistic estimates for page size, disk access time, and so forth.
- No index (heap file)
  - A static hash index (with no overflow pages)
  - A clustered, unintegrated B<sup>+</sup> tree index

- 9.18 Estimate the time in milliseconds to insert a new row into the table of Exercise 9.17 when each of the following indices is used:
- No index (file sorted on the search key)
  - A static hash index (with no overflow pages)
  - A clustered, unintegrated B<sup>+</sup> tree index (with no node splitting required)
- 9.19 Estimate the time in milliseconds to update the search-key value of a row in the table of Exercise 9.17 when each of the following is used:
- No index (file sorted on the search key)
  - A static hash index (where the updated row goes in a different bucket than the original row's page, but no overflow pages are required)
  - A clustered, unintegrated B<sup>+</sup> tree index (where the updated row goes on a different page than the original row's pages, but no node splitting is required)
- 9.20 Estimate the amount of space required to store the B<sup>+</sup> tree of Exercise 9.17 and compare that with the space required to store the table.
- 9.21 Explain what index types are supported by your local DBMS. Give the commands used to create each type.
- 9.22 Design the indices for the tables in the Student Registration System. Give the rationale for all design decisions (including those not to use indices in certain cases where they might be expected).
- 9.23 Explain the rationale for using fillfactors less than 1 in
- Sorted files
  - ISAM indices
  - B<sup>+</sup> tree indices
  - Hash indices
- 9.24 Design an algorithm for maintaining join indices incrementally—that is, so that the addition of new tuples to the relations involved in the join do not require the entire join to be recomputed from scratch.
- \*9.25 Propose an improvement that permits bitmap indices to maintain only  $n - 1$  bit vectors in order to represent attributes that can take  $n$  different values. Discuss the change that is needed to perform selection, especially in case of a multiattribute selection that requires a Boolean operation on multiple bit vectors.



# 10

## The Basics of Query Processing

An application designer must understand the principles and methods of query processing in order to produce better systems. In this chapter, we examine the methods used to evaluate the basic relational operators and discuss their impact on physical database design. Chapter 11 will deal with a more advanced aspect of query processing: query optimization.

### 10.1 Overview of Query Processing

SQL queries submitted by the user are first parsed by the DBMS parser. The parser verifies the syntax of the query and, using the system catalogue, determines if the attribute references are correct. For instance, TRANSCRIPT.Student is not a correct reference because the relation TRANSCRIPT does not have Student as an attribute. Likewise, applying the AVG operator to the CrsCode attribute violates the type of that attribute.

Since an SQL query is declarative rather than procedural, it does not suggest any specific implementation. Thus, parsed queries must first be converted into relational algebra expressions. For instance, a query

---

```
SELECT S.Name  
FROM TRANSCRIPT T, STUDENT S  
WHERE T.Semester='F2004' AND S.Id=T.StudId AND S.Name='John Doe'
```

---

might be translated into the following expression:

---

```
 $\pi_{\text{Name}}(\sigma_{\text{Semester} = \text{'F2004'}} \text{ AND } \text{StudId}=\text{Id} \text{ AND } \text{Name} = \text{'John Doe'} (\text{TRANSCRIPT} \times \text{STUDENT}))$ 
```

---

A naive way to evaluate this expression would be to compute the results of the relational operators directly as specified. If we did so on a large university database, we would quickly realize that it takes awhile to get an answer. The problem is that the expression involves the Cartesian product, and the intermediate result of the computation can become very large only to be reduced to a few bytes in the end.

To improve performance, a DBMS will play around with the above expression before trying to evaluate it. First, it might convert it to an equivalent expression of the following (or similar) form:

---

$$\pi_{\text{Name}}(\sigma_{\text{Semester}=\text{'F2004'}}(\text{TRANSCRIPT} \bowtie_{\text{StudId}=\text{Id}} \sigma_{\text{Name} = \text{'John Doe'}}(\text{STUDENT})))$$

---

This transformation is based on a heuristic that believes that joins are better than Cartesian products and that joining smaller relations is better than joining larger ones. Next, the system would choose the appropriate algorithms to compute each operator mentioned in the query. It will make these decisions based on a set of heuristics for estimating the cost of the different algorithms.

This complex series of transformations and estimates, which result in a **query execution plan**, are performed by a DBMS module called a **query optimizer**. A simplified view of query processing in a typical DBMS is depicted in Figure 5.1 on page 129.

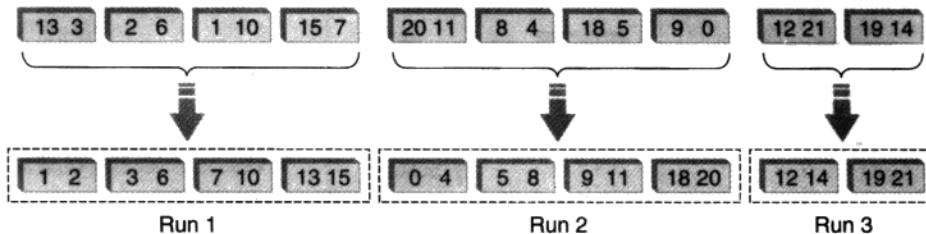
We will study the workings of a query optimizer in the next chapter. But before we can do this, we need to familiarize ourselves with the repertoire of algorithms that a query optimizer has at its disposal, and we need to learn to estimate the cost of the alternative ways of computing the relational operators. These algorithms are the subject of this chapter.

## 10.2 External Sorting

Sorting is an important part of many algorithms used in computer programming and is at the very core of the algorithms that support relational operations. For instance, sorting is one of the most efficient ways to get rid of duplicate tuples and is also the basis of some join algorithms. The sorting algorithms used to process queries in relational DBMSs are not the ones you might have studied in a basic course on algorithms. The latter are designed to perform sorting when all data is stored in main memory, which usually cannot be assumed in the database context. When files are large and are kept in external storage, such as a disk, we use what is called **external sorting**.

The main idea behind external sorting is to bring portions of the file into main memory, sort them using one of the known in-memory algorithms (e.g., Quicksort), and then dump the result back to disk. This creates sorted file segments, which must be merged in order to create a single sorted file. Because the time to execute an I/O operation is several orders of magnitude greater than the time to execute an instruction, it is assumed that the cost of I/O dominates the cost of in-memory sorting. Hence, the computational complexity of external sorting is often measured only in the number of disk reads and writes. More specifically, we will measure the complexity in terms of the number of pages that need to be transferred between the main memory and the disk. A typical external sorting algorithm consists of two stages: *partial sorting* and *merging*.

**Partial sorting.** The partial sorting stage is very simple. Suppose that we have a buffer in main memory that can accommodate  $M$  pages available for sorting and



**FIGURE 10.1** Partial sorting of a file,  $M = 4$ ,  $F = 10$ .

that the file has  $F$  pages.  $F$  is typically much larger than  $M$ . The first stage of the algorithm is as follows:

---

```

do {
    read  $M$  pages from disk into main memory
    sort them in memory with one of the known methods.
        (Assume that, apart from the  $M$ -page buffer, additional memory
         is available to enable the in-memory sorting algorithm to run)
    dump the sorted file segment into a new file
} until (end-of-file)

```

---

We use the term **run** to refer to a sorted file segment produced by one iteration of the above loop. The size of a run is the number of pages in the segment. Thus, the first stage produces  $\lceil F/M \rceil$  sorted runs at the cost of  $2F$  disk I/O operations (for simplicity, we assume that each I/O operation transfers exactly one page to or from main memory). (The symbol  $\lceil \cdot \rceil$  here denotes the operation of rounding up to the nearest integer that is greater than or equal to  $F/M$ .) The partial sorting stage is illustrated in Figure 10.1, in which we assume that each disk block contains two file records.

***k*-way merging.** The next stage of the algorithm takes the sorted runs and merges them into larger sorted runs. This process can be repeated until we end up with just one sorted run, which is our final goal: the sorted version of the original file. A  $k$ -way merge algorithm takes  $k$  sorted runs of size  $R$  pages and produces one run of size  $kR$ , as illustrated in Figure 10.2. The actual algorithm works as follows:

---

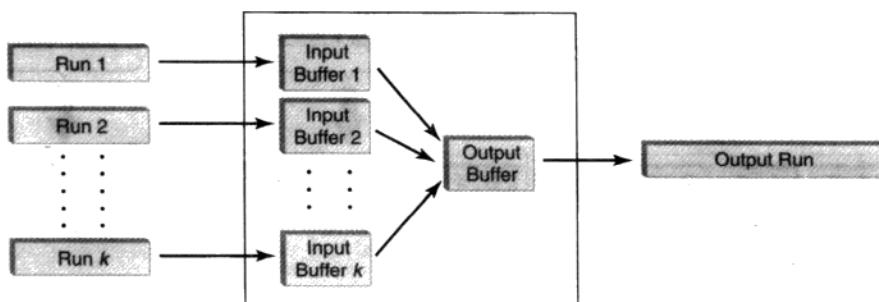
```

while (there are nonempty input runs) {
    choose a smallest tuple (with respect to the sort key) in each
        run, and output the smallest among these
    delete the chosen tuple from the respective input run
}

```

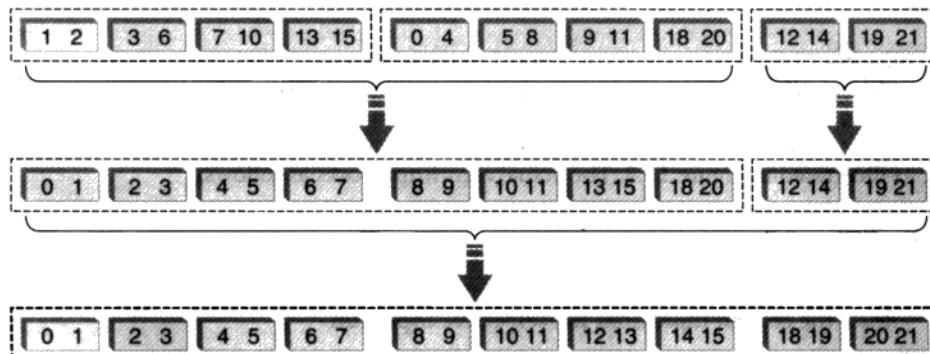
---

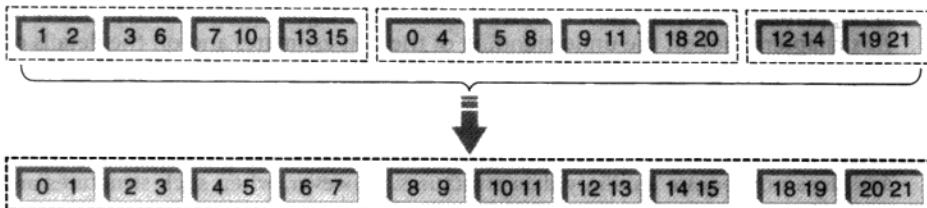
Because each run is sorted, the choice step is simple since the smallest remaining element in a run is always its current head element. Figure 10.3 illustrates repeated

FIGURE 10.2  $k$ -way merge.

application of the 2-way merge step; Figure 10.4 shows a 3-way merge. In each case we need to have a buffer with enough pages to perform the merge—three for the 2-way merge and four for the 3-way merge. The buffer size in the figures ( $M = 4$ ) can accommodate both 2-way and 3-way merges.

What is the cost of a  $k$ -way merge of  $k$  runs of size  $R$  pages? Clearly, each run must be scanned once and then the entire output must be written back on disk. The cost is thus  $2kR$ . Since we might start with more than  $k$  runs, we divide them into groups of  $k$  and apply the  $k$ -way merge separately to each group. If we refer to this process as a step and if we start with  $N$  runs, we have  $\lceil N/k \rceil$  groups, and the upper bound on the total cost of the merge step is  $2RN$ . Note that this value does not depend on  $k$ . Moreover, since at the next merge step we start with  $\lceil N/k \rceil$  runs, each with a maximum size  $kR$ , the cost of the merge again does not exceed  $2RN$ , and we are left with  $\lceil N/k^2 \rceil$  runs. In fact, it is easy to show by induction that this upper bound on the I/O cost holds for every step of the merge algorithm.

FIGURE 10.3 Merging sorted runs in a 2-way merge,  $M = 4, F = 10$ .



**FIGURE 10.4** Merging sorted runs in a 3-way merge,  $M = 4$ ,  $F = 10$ .

The next question is what the value of  $k$  should be at the merging stage in an external sort algorithm. If we start with  $N$  runs and perform a  $k$ -way merge at each step, the number of steps is  $\lceil \log_k N \rceil$ . Thus, the cost of the entire merging stage of the algorithm is bound by  $2RN * \log_k N$ , where  $R$  is the initial size of a sorted run. Since, in our case,  $R = M$  (i.e., we can use the entire buffer to produce the largest possible initial runs) and consequently  $N = \lceil F/M \rceil$ , we conclude that the cost is bounded by  $2F * \log_k \lceil F/M \rceil$ .

Thus, it appears that the larger  $k$  is, the smaller is the cost of external sorting. Why, then, should we not take  $k$  to be the maximum possible—that is, the number of initial sorted runs? The answer is that we are limited by the size  $M$  of the main memory buffer allocated for the external sort procedure. This reasoning suggests that we should utilize this memory in such a way as to make  $k$  as large as possible. Since we must allocate at least one page to collect the output (which is periodically flushed to disk) during the merge, the maximum value of  $k$  is  $M - 1$ . By substituting this value into the earlier cost estimate, we obtain the following estimate:

$$2F (\log_{(M-1)} F - \log_{(M-1)} M) \approx 2F(\log_{(M-1)} F - 1)$$

Finally, by combining this cost with the cost of the partial sorting stage, we obtain an estimate for the entire procedure of external sorting:

$$2F * \log_{(M-1)} F$$

**10.1**

In commercial DBMSs, the external sorting algorithms are highly optimized. They take into account not only the cost of in-memory sorting of the initial runs but also the fact that transferring multiple pages in one I/O operation might be more cost-effective than performing several I/O operations that transfer one page at a time. For instance, if we have a buffer that holds 12 pages, we can either perform an 11-way merge reading one page at a time or a 3-way merge reading three pages at a time. Since reading (and writing) three pages takes almost the same time as reading one, it is reasonable to consider such a 3-way merge as an alternative. Another consideration has to do with delays when the output buffer is being flushed to disk during the merge operation. Techniques such as double or triple buffering might be used to reduce such delays. Despite all of these simplifications, however, the algorithm and the cost estimate developed in this section provide a good approximation for what

is happening in real systems. Our discussion of the algorithms in the remainder of this chapter relies on the understanding of the cost estimates and on the details of the sorting algorithm developed here.

**Sorting and B<sup>+</sup> trees.** The merge-based algorithm described above is the most commonly used sorting method in query processing because it works in all cases and does not require auxiliary data structures. However, when such structures are available, sorting can be performed at a lower cost.

For instance, suppose that a secondary B<sup>+</sup> tree index on the sort key is available. Traversal of the leaf entries of the tree produces a sorted list of the record IDs (rids) for the actual data file. In principle, then, we can simply follow the pointers and retrieve the records in the data file in the order of the search key. Surprisingly, this might not always beat the merge-based algorithm!

The main consideration in deciding whether a B<sup>+</sup> tree index is worth considering for sorting a file is whether the index is clustered or unclustered. The short answer is, if the index is clustered, using a B<sup>+</sup> tree index is a good idea; otherwise, it might be a bad idea. If the index is clustered, the data file must already be almost sorted (by definition), so there is nothing we need to do. However, if the index is unclustered, traversing the leaves of a B<sup>+</sup> tree and following the data record pointers retrieves pages of the main file in random order. In the worst case, this might mean that we must transfer one page for each *record* in the index leaf (recall that our previous analysis was based on the number of *pages* in the file—typically a much smaller number than the number of records). Exercise 10.1 deals with estimating the cost of using unclustered B<sup>+</sup> trees for external sorting.

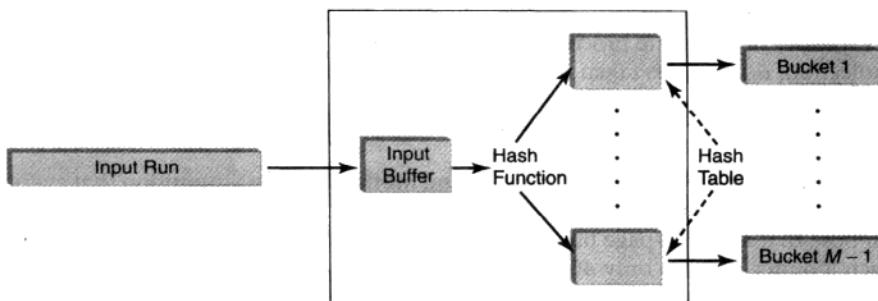
## 10.3 Computing Projection, Union, and Set Difference

At first glance, computing the projection, union, and set difference operators is easy. With projection, for instance, we can just scan the relation and delete the unwanted columns. However, the situation is more complicated if the user query has the DISTINCT directive. The problem here is that duplicate tuples, which might arise as a result of the projection operation, must be eliminated. For instance, if we project out the attributes StudId and Grade of the relation TRANSCRIPT in Figure 3.5 on page 39, the tuple  $\langle \text{MGT123}, \text{F1994} \rangle$  will appear twice in the result. Thus, we must find efficient ways of eliminating duplicate tuples.

The same problem with duplicates can arise in the computation of the union of two relations. In the case of the difference of two relations,  $r - s$ , duplicates cannot arise unless the relation  $r$  had them all along. Nevertheless, the problem we are facing is similar: to identify the tuples in  $r$  that are equal to tuples in  $s$ .

There are two techniques for finding identical tuples: sorting (which we discussed in Section 10.2) and hashing. We first apply these techniques to the projection operator and then discuss the modifications needed for union and set difference operators.

**Sort-based projection.** This technique scans the original relation, removes the tuple components that are to be projected out, and writes the result back on disk.



**FIGURE 10.5** Hashing input relation into buckets.

(We assume that there is not enough memory to store the result.) The cost of this operation is of the order  $2F$ , where  $F$  is the number of pages in the relation. Then the result is sorted at the cost of  $2F * \log_{(M-1)} F$ , where  $M$  is the number of main memory pages available for sorting. Finally, we scan the result again (at the cost of  $2F$ ), and, since identical tuples are right next to each other (because the relation is sorted), we can easily delete the duplicates.

In fact, we can do better than that if we combine sorting and scanning. First, we delete the unwanted components from the tuples during the partial sorting stage of the sorting algorithm. At that stage, we have to scan the original relation anyway, so removal of the tuple components comes at no additional cost in terms of disk I/O. Second, we eliminate the final scan needed to remove the duplicates by combining duplicate elimination with the steps in which sorted runs are output to the disk. Since each such step writes out blocks of sorted tuples, duplicate elimination can be done in main memory, at no additional I/O cost.

Thus, the cost of the sort-based projection is  $2F * \log_{(M-1)} F$ . Furthermore, if we take into account that the first scan is likely to produce a smaller relation (of size  $\alpha F$ , where  $\alpha < 1$  is the reduction factor), the cost of projection is even lower (see Exercise 10.2).

**Hash-based projection.** Another way to quickly identify the duplicates is to use a hash function. Suppose that a hash function yields integers in the range of 1 to  $M - 1$  and that there are  $M$  buffer pages in main memory, which includes an  $(M - 1)$ -page hash table and an input buffer. The algorithm works as follows. In the first phase, the original relation is scanned. During the scan, we chop off the tuple components that are to be projected out, and the rest of the tuple is hashed on the remaining attributes. Whenever a page of the hash table becomes full, it is flushed to the corresponding bucket on disk. This step is illustrated in Figure 10.5.

Clearly, duplicate tuples are always hashed into the same bucket, so we can eliminate duplicates in each bucket separately. This elimination is done in the second phase of the algorithm. Assuming that each bucket fits into the main memory, the second phase can be carried out by simply reading each bucket in its entirety, sorting

it in main memory to eliminate the duplicates, and then flushing it to disk. The I/O complexity of the entire process is  $4F$  (or  $F + 3\alpha F$  if the size reduction factor due to projection,  $\alpha$  ( $\alpha < 1$ ), is taken into account). If the individual buckets do not fit in main memory, they must be sorted using external sorting, which incurs additional I/O overhead. Exercise 10.3 deals with the cost estimate in this case.

**Comparison of sort-based and hash-based methods.** The assumption that every bucket can fit into main memory is realistic even for very large files. For instance, suppose that a 10,000-page buffer is available to the program to do the projection. Such a buffer requires only 40M of memory, which is well within the range of an inexpensive desktop computer. We can first use this buffer to store the hash table and then use it to read in the buckets. Let us assume that each hash bucket fits into our buffer and suppose we have a  $10,000 \times 10,000 = 10^8$ -page file (400G) to process. According to the above discussion, projection of such a file can be computed at the cost of just under  $4 \times 10^8$  page transfers. Does the sort-based projection algorithm fare better? In this case, the sort-based algorithm costs a bit more:  $2 \times 10^8 \log_{10^4-1} 10^8$  page transfers.

The possibility that we might have to externally sort an average-size hash bucket is fairly remote. A much bigger risk is that the hash function we use will not distribute tuples to the buckets evenly. In this case, although the average bucket might fit in main memory, other buckets will not. In the worst case (which is unlikely), all tuples might fall into the same buffer, which will require external sorting. The cost will then be  $2F$  to scan the original relation and copy it into the single bucket plus  $2F \log_{(M-1)} F$  to sort the bucket and eliminate the duplicates—a waste of  $2F$  page transfers compared to sort-based projection.

**Computing union and set difference.** Computing the union and set difference of two relations is similar to computing projection, except that we do not need to chop off unwanted attributes. For example, to compute a set difference,  $r - s$ , we sort both  $r$  and  $s$  and then scan them in parallel, similar to the merging process. However, instead of merging tuples, whenever we discover that a tuple,  $t$ , of  $r$  is also in  $s$ , we do not add it to the result. Furthermore, we can combine this step with the final merges of the sorted runs of  $r$  and of  $s$ , which are part of the sorting algorithm. Details of this combination are left to Exercise 10.9. Since the final scan of  $r$  and  $s$  comes for free, the cost of union and difference operations is the cost of sorting of these two relations.

In hash-based set difference computation, we can hash  $r$  and  $s$  into buckets as described earlier. However, in each bucket we must keep the distinction between tuples that came from  $r$  and those that came from  $s$ . In the second stage, the set difference operation must be applied to each bucket separately.

## 10.4 Computing Selection

Computing the selection operator can be much more complex than computing projection and computing the set operations, and a wider variety of techniques can be used. The choice of a technique for a particular selection operator can depend on

the type of the selection condition and on the physical organization of the relation in question. Typically, the DBMS decides automatically on the technique it will use based on the heuristics that we describe in the following subsections. However, understanding these heuristics gives the programmer an opportunity to request the physical organization that is most favorable to the selection types that occur most frequently in a particular application.

We first consider simple selection conditions of the form  $attr \text{ op } value$  (where  $\text{op}$  is one of the comparisons  $=$ ,  $>$ ,  $<$ , and the like) and then generalize our techniques to complex conditions that involve Boolean operators.

Database queries usually lead to two distinct kinds of selection: those based on equality ( $attr = val$ ) and those based on inequality (such as  $attr < val$ ). The latter are called **range queries** because they usually come in pairs that specify a range of values, for example,  $\sigma_{c_1 < attr \leq c_2}(\mathbf{r})$ .

Our discussion of the techniques for implementing the selection operator focuses on the cost of retrieving the requisite tuples and ignores the cost of outputting the result. The reason is that the cost of the output is the same in all cases and thus is irrelevant for comparing the different techniques. Section 11.4 presents some heuristics for estimating this cost.

### 10.4.1 Selections with Simple Conditions

One obvious way of evaluating a selection,  $\sigma_{attr \text{ op } value}(\mathbf{r})$ , is to scan the relation  $\mathbf{r}$  and check the selection condition for each tuple, outputting those that satisfy it. However, if only a small number of tuples satisfies the condition, the price of scanning an entire relation seems too high. In situations where more information is available about the structure of  $\mathbf{r}$ , a complete scan is not needed. We consider three cases: (1) when no index is available on  $attr$ ; (2) when there is a  $B^+$  tree index on  $attr$ ; and (3) when there is a hash index on  $attr$ . In case 3, only the equality selections (where  $\text{op}$  is  $=$ ) can be handled efficiently. In case 2, both equality selections and range queries can be handled efficiently, although case 3 is generally better for equality selection. In case 1, complete scan of  $\mathbf{r}$  is the only option unless the relation  $\mathbf{r}$  is already sorted on  $attr$ . In this case, both equality and range conditions can be handled, although not as efficiently as when a  $B^+$  tree index is available.

**No index.** In general, we might have no choice but to scan the entire relation  $\mathbf{r}$  at the cost of  $F$  page transfers (the number of disk blocks in  $\mathbf{r}$ ). However, if  $\mathbf{r}$  is sorted on  $attr$ , we can use binary search to find the pages of  $\mathbf{r}$  that house the first tuple where  $attr = value$  holds. We can then scan the file in the appropriate direction to retrieve all of the tuples that satisfy  $attr \text{ op } value$ .

The cost of such a binary search is proportional to  $\log_2 F$ . To this, we must add the cost of scanning the blocks that contain the qualifying tuples. For instance, if  $\mathbf{r}$  has 500 pages, the cost of the search is  $\lceil \log_2 500 \rceil$ , that is, 9 page transfers (plus the number of disk blocks that contain the qualifying entries).

**$B^+$  tree index.** With a  $B^+$  tree index on  $attr$ , the algorithm is similar to that for a sorted file. However, instead of the binary search, we use the index to find the first

tuple of  $r$  where  $attr = value$ . More precisely, we find the leaf node of the  $B^+$  tree that contains or points to the first row satisfying the condition. From there, we scan the leaves of the  $B^+$  tree index to find all of the index entries that point to the pages that hold the tuples that satisfy  $attr \text{ op } value$ .

The cost of finding the first qualifying leaf node of the index equals the depth of the  $B^+$  tree. As before, we also have to add the cost of scanning the leaves of the index to identify all of the qualifying entries. Of course, this cost depends on the number of qualifying entries, which depends on the selection condition as well as on the actual data in the relation.

This is not the whole story, however. So far we have described only the process of getting the index entries. The cost of getting the actual tuples depends on whether or not the index is clustered. If the index is clustered, all of the tuples of interest are stored in one page or in several adjacent pages (this is true whether or not the index is integrated into the storage structure). For instance, if there are 1000 qualifying tuples and each disk block stores 100 tuples, getting all these tuples (assuming that we already had found the appropriate index nodes) requires 10 page transfers. On the other hand, if the index is unclustered, each qualifying tuple might be in a separate block, so retrieving all qualifying tuples might take 1000 page transfers!

This raises the unhappy prospect of having to perform as many page transfers as the number of qualifying tuples in the selection, which can handily beat the number of pages in the entire relation  $r$ . Fortunately, with a little thought, we can do better than that. Let us first sort the record IDs of the qualifying tuples that we obtained from the index. Then we can retrieve the data pages from the relation in the ascending order of qualifying record IDs, which guarantees that every data page will be retrieved at most once. Thus, even with an unclustered index, the cost is proportional to the number of pages that contain qualifying tuples (plus the cost of searching the index and sorting the record IDs). In the worst case, this cost can be as high as the number of pages in the original relation (but not as large as the number of tuples there!). This is because the qualifying tuples are not packed into the retrieved pages as they would be with a clustered index: a retrieved page might only contain a single qualifying tuple. Hence, a clustered index is still greatly preferred.

**Hash index.** In this case, we can use the hash function to find the bucket that has the tuples where  $attr = value$  holds. Since two tuples that differ only slightly in  $value$  can hash to different buckets, this method cannot be efficiently used with range conditions, such as  $attr < value$ .

Generally, the cost of finding the right bucket is constant (close to 1.2 for a good hash function). However, the actual cost of tuple retrieval depends on the number of qualifying tuples. If this number is larger than one, then, as in the case of  $B^+$  tree indices, the actual cost depends on whether or not the index is clustered. In the clustered case, all qualifying tuples are packed into a few adjacent pages and the cost of retrieval is just the cost of scanning these pages. In the unclustered case, tuples are scattered through the data file and we face the same problem as with unclustered  $B^+$  trees—sorting the record IDs results in a cost proportional to the number of pages that contain qualifying tuples.

### 10.4.2 Access Paths

The above algorithms for implementing the relational operators all assume that certain auxiliary data structures (indices) are available (or unavailable) for the relations being processed. These data structures, along with the algorithms that use them, are called **access paths**. So far, we have seen several examples of access paths that can be used to process a particular query: a *file scan* can always be used; a *binary search* can be used on files that are sorted on attributes specified in the query; a *hash index* or a  $B^+$  tree can be used if the index has a search key that involves those attributes.

**Example 10.4.1 (Access Paths).** Consider the TRANSCRIPT relation of Figure 3.5, page 39, and suppose that we have a hash index on the search key  $\langle \text{StudId}, \text{Semester} \rangle$ . The index is useful in computing  $\pi_{\text{StudId}, \text{Semester}}(\text{TRANSCRIPT})$ , since we can be certain that any duplicates created as a result of the projection originate from tuples that were stored in the same bucket. This greatly simplifies duplicate elimination since the search for duplicates can be done one bucket at a time. If we compute  $\pi_{\text{StudId}, \text{CrsCode}}(\text{TRANSCRIPT})$ , on the other hand, the index is of no help in eliminating duplicates. Although the projections of tuples  $t_1$  and  $t_2$  on  $\langle \text{StudId}, \text{CrsCode} \rangle$  might be identical,  $t_1$  and  $t_2$  might be hashed to different buckets since the values of their **Semester** attribute can differ. In order to use hashing for duplicate elimination, the entire search key of the hash index must be contained in the set of attributes that survive the projection (see Exercise 10.4).

At the same time, the above hash index can be very useful for computing the query  $\sigma_{\text{StudId}=666666666 \wedge \text{Grade}='A' \wedge \text{Semester}='F1994'}(\text{TRANSCRIPT})$ . We can use the hash index to retrieve the tuples that satisfy the partial condition  $\text{StudId}=666666666 \wedge \text{Semester}='F1994'$  and then scan the result (which presumably will be small) to find the tuples that additionally satisfy  $\text{Grade}='A'$ . On the other hand, the same index is of no help in evaluating the expression  $\sigma_{\text{StudId}=666666666}(\text{TRANSCRIPT})$  since we cannot hash on a partial-search key and tuples satisfying this condition can be scattered over different buckets.

Finally, a hash index on  $\langle \text{Grade}, \text{StudId} \rangle$  is not helpful for computing the query  $\sigma_{\text{Grade}>'C'}(\text{TRANSCRIPT})$ , but a  $B^+$  tree index on the search key  $\langle \text{Grade}, \text{StudId} \rangle$  can help (although a  $B^+$  tree with search key  $\langle \text{StudId}, \text{Grade} \rangle$  cannot). To use the hash function, we have to supply all possible values for **StudId** and all values for **Grade** above 'C', which is impractical. In contrast, since **Grade** is a *prefix* of the  $B^+$  tree search key, we can use this tree to efficiently find all of the index entries with the search key  $\langle g, id \rangle$ , where  $g$  is higher than 'C'. ■

Example 10.4.1 leads to the notion of when an access path *covers* the use of a particular relational operator. We define this notion precisely only for a selection operator whose selection condition is a conjunction of terms of the form  $\text{attr op value}$ . Projection and set difference operators are left to Exercise 10.6.

Covering relates access paths to relational expressions that can be evaluated using those paths. Consider a relational expression of the form

$$\sigma_{\text{attr}_1 \text{op}_1 \text{val}_1 \wedge \dots \wedge \text{attr}_n \text{op}_n \text{val}_n}(\mathbf{R})$$

where  $R$  is a relation schema. This expression is **covered** by an access path if and only if one of the following conditions holds:

- The access path is a file scan. (A file scan can obviously be used to compute any expression.)
- The access path is a hash index whose search key is a subset of the attributes  $attr_1, \dots, attr_n$  and all  $op_i$  in this subset are equality operators. (We can use the hash index to identify the tuples that satisfy some of the conjuncts in the selection condition and then scan the result to verify the conjuncts that remain.)
- The access path is a  $B^+$  tree index with the search key  $sk_1, \dots, sk_m$  such that some prefix  $sk_1, \dots, sk_i$  of that search key is a subset of  $attr_1, \dots, attr_n$ . (Section 9.4.3 explained how to use  $B^+$  trees for partial-key searches. This can help us find the tuples that satisfy some of the conjuncts in the selection. The rest of the conjuncts can be verified by a sequential scan of the result.)
- The access path is a binary search, and the relation instance corresponding to  $R$  is sorted on the attributes  $sk_1, \dots, sk_m$ . The definition of covering in this case is the same as for  $B^+$  tree indices.

Note that access paths based on hashing can be used only if all comparisons that correspond to the attributes in the search key are  $=$ . The other access paths can be used even if these comparisons involve inequalities, such as  $\leq$ ,  $<$ ,  $>$ , and  $\geq$ . If the comparison operator is  $\neq$ , the only applicable access path is a file scan since no index is effective in enumerating all the qualifying tuples in this case.

**Example 10.4.2 (Covering Complex Selection).** Consider the expression  $\sigma_{a_1 \geq 5 \wedge a_2 = 3.0 \wedge a_3 = 'a'}(R)$ , and assume that there is a  $B^+$  tree with search key  $a_2, a_1, a_4$  on  $R$ . This access path covers the expression, which can be computed as follows. Use the index to find the leaf entry,  $e$ , in which  $a_2$  has value 3.0 and  $a_1$  has value 5 or, if such an entry is not present, the first entry in the index that would follow  $e$ . Then scan the leaf entries from that point on to find all those tuples in which, in addition,  $a_3$  has value  $a$ . ■

One more notion before we proceed: the **selectivity** of an access path is the number of pages that will be retrieved if we use the evaluation method corresponding to that path. The smaller the selectivity, the better the access path. Selectivity is closely related to the cost of evaluating a query, although the query cost might involve other factors. For example, multiple access paths might be used (see Section 10.4.3) or it might be necessary to sort the result before output (if an `ORDER BY` clause is used). Clearly, for any given relational expression, access path selectivity depends on the size of the result of that expression and is always greater than or equal to the number of pages that hold the tuples in that result. However, some access paths have selectivity that is closer to the theoretical minimum, while others are closer to the cost of the entire file scan. The notion of when an access path covers an expression helps in identifying access paths whose selectivity seems “reasonable” for the given type of expression.

### 10.4.3 Selections with Complex Conditions

We are now ready to discuss the methods used to evaluate arbitrary selection.

**Selections with conjunctive conditions.** These are the expressions of the form (10.2) considered on page 389. We have two choices:

1. *Use the most selective access path to retrieve the corresponding tuples.* Such an access path tries to form a prefix of the search key by using as many of the attributes mentioned in the selection condition as possible. In this way, it retrieves the smallest possible superset of the required tuples, and we can scan the result to find the tuples that satisfy the entire selection condition. For instance, suppose that we need to evaluate

$$\sigma_{\text{Grade} > 'C' \wedge \text{Semester} = 'F1994'} (\text{TRANSCRIPT})$$

and there is a  $B^+$  tree index with the search key (`Grade`, `StudId`). Since this access path covers the selection condition `Grade > 'C'`, we can use it to compute  $\sigma_{\text{Grade} > 'C'} (\text{TRANSCRIPT})$ . Then we can scan the result to identify the transcript records that correspond to the fall 1994 semester. If we had a  $B^+$  tree index with the search key (`Semester`, `Grade`), we could use it as an access path, because this path covers both selection conditions.

2. *Use several access paths that cover the expression.* For instance, we might have two secondary indices whose selectivity is less than that of the plain file scan. We can then use both access paths to find the rids of the tuples that might belong to the query result and then compute the intersection of those sets of rids. Finally, we can retrieve the selected tuples and test them for the remaining selection conditions. For instance, consider the expression

$$\sigma_{\text{StudId} = 666666666 \wedge \text{Grade} = 'A' \wedge \text{Semester} = 'F1994'} (\text{TRANSCRIPT})$$

and suppose that there are two hash indices: on `Semester` and on `Grade`. Using the first access path, we can find the rids of the transcript records for the fall 1994 semester. Then we can use the second access path to find the rids for the records with grade 'A'. Finally, we can find the rids that belong to both sets and retrieve the corresponding pages. As we scan the tuples, we can further select those that correspond to the student Id 666666666.

**Selections with disjunctive conditions.** When selection conditions contain disjunctions, we must first convert them into *disjunctive normal form*. A condition is in **disjunctive normal form** if it has the form  $C_1 \vee \dots \vee C_m$ , where each  $C_i$  is a conjunction of comparison terms (as in expression (10.2)).

It is known from elementary predicate calculus that every condition has an equivalent disjunctive normal form. For instance, for the condition

---


$$(\text{Grade} = 'A' \vee \text{Grade} = 'B') \wedge (\text{Semester} = 'F1994' \vee \text{Semester} = 'F1995')$$


---

the corresponding disjunctive normal form is

---


$$\begin{aligned} & (\text{Grade} = 'A' \wedge \text{Semester} = 'F1994') \vee (\text{Grade} = 'A' \wedge \text{Semester} = 'F1995') \\ & \vee (\text{Grade} = 'B' \wedge \text{Semester} = 'F1994') \vee (\text{Grade} = 'B' \wedge \text{Semester} = 'F1995') \end{aligned}$$


---

For conditions in disjunctive normal form, the query processor must examine the available access paths for the individual disjuncts and choose the appropriate strategy. Here are some possibilities:

- *One of the disjuncts,  $C_i$ , must be evaluated using a file scan.* In this situation, we might as well evaluate the entire selection expression during that scan.
- *Each  $C_i$  has an access path that is better than the plain file scan.* We have two subcases here:
  1. The sum of the selectivities of all of these paths is close to the selectivity of the file scan. In this case, we should prefer the file scan because the overhead of the index search and other factors are likely to outweigh the small potential gain due to the use of more sophisticated access paths.
  2. The combined selectivity of the access paths for all disjuncts is much smaller than the selectivity of the file scan. In this case, we should compute  $\sigma_{C_i}(R)$  separately, using the appropriate access paths, and then take the union of the results.

## 10.5 Computing Joins

The methods for computing projections, selections, and the like, are nothing but a prelude to a more difficult problem: evaluation of relational joins. With all of the attention given to the comparison of different access paths, the worst that can happen during the computation of a projection or selection is that we might have to scan or sort the entire relation. The result of such an expression is also well behaved: it cannot be larger than the original relation.

Compare this to relational joins, where both the number of pages to be scanned and the size of the result can be *quadratic* in the size of the input. While “quadratic” might not seem too bad in applications in which some algorithms have exponential complexity, it is prohibitive in database query evaluation because of the large amounts of data and the relative slowness of disk I/O. For example, joining two files that span a mere 1000 pages can require  $10^6$  I/O operations, which might be unacceptable even for batch jobs. And joining just three such relations would fetch  $10^9$  I/Os. For these reasons, joins are given special attention in database query processing.

Consider a join expression  $r \bowtie_{A=B} s$ , where  $A$  is an attribute of  $r$  and  $B$  is an attribute of  $s$ . There are three main methods for computing joins: nested loops (with and without the help of indices), sort-merge, and hash-based joins. We consider them in this order.

### 10.5.1 Computing Joins Using Simple Nested Loops

One obvious way to evaluate the join  $r \bowtie_{A=B} s$  is to use the following loop:

---

```
foreach t ∈ r do
    foreach t' ∈ s do
        if t.A = t'.B then output (t, t')
```

---

The cost of this procedure can be estimated as follows. Let  $F_r$  and  $F_s$  be the number of pages in  $r$  and  $s$ , and let  $\tau_r$  and  $\tau_s$  be the number of tuples in  $r$  and  $s$ , respectively. It is easy to see that the relation  $s$  must be scanned from start to end for each tuple in  $r$ , resulting in  $\tau_r F_s$  page transfers. In addition,  $r$  must be scanned once in the outer loop. All in all, there are  $F_r + \tau_r F_s$  page transfers. (In all cost estimates for the join operation, we ignore the cost of writing the final result to disk because this step is the same for all methods and also because the estimate at this stage depends on the actual size of the result. Trying to estimate this size takes us away from the main topic and is a distraction at this stage.)

The above cost estimate teaches us two lessons:

1. *It involves a lot of page transfers!* Let  $F_r = 1000$ ,  $F_s = 100$ , and  $\tau_r = 10,000$ . Our cost estimate says that the computation might require  $1000 + 10,000 \times 100 = 1,001,000$  page transfers—too much to join such relatively small tables (about 166 minutes if one page I/O takes 10 ms).
2. *The order of the loops matters.* Suppose that instead of scanning  $r$  in the outer loop, we scan  $s$  in the outer loop and  $r$  in the inner loop. Suppose that  $\tau_s = 1000$ . Switching  $r$  and  $s$  in the cost estimate yields:  $100 + 1000 \times 1000 = 1,000,100$ . Although, in this example, the reduction in operations is minimal (a whopping nine seconds!), it is clear that the order of scans matters.

Simple nested loops are actually never used for computing joins, because it is a hugely wasteful method. We will now consider a related, but much better technique.

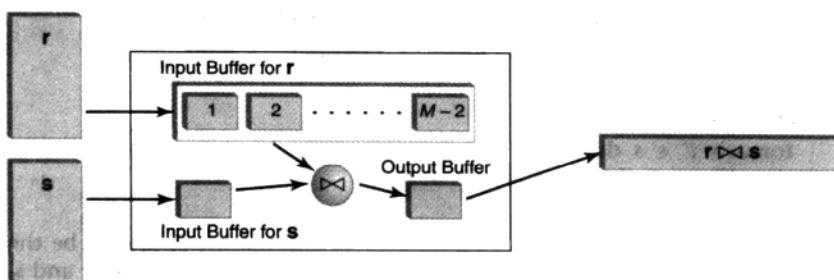
**Block-nested loops join.** The complexity of the nested loops join can be reduced considerably if, instead of scanning  $s$  once per tuple of  $r$ , we scan it once per page of  $r$ . This will reduce the cost estimate to  $F_r + F_r F_s$ —a reduction of an order of magnitude from the above example. The way to achieve this feat is to output the result of the join for all tuples in the page of  $r$  that is currently in memory.

---

```
foreach page  $p_r$  of r do
    foreach page  $p_s$  of s do
        output  $p_r \bowtie_{A=B} p_s$ 
```

---

If we can reduce the number of scans of  $s$  to one per page of  $r$ , can we further reduce this number to one per group of pages? The answer is yes—if we can use a little more memory. Suppose that the query processor has an  $M$ -page main memory buffer to do the join. We can allocate  $M - 2$  of these pages for the outer loop relation



**FIGURE 10.6** Block-nested loops join.

$r$  and one page for the inner loop relation  $s$ , leaving the last page reserved for the output buffer. This process is depicted in Figure 10.6.

The cost of a block-nested loops join can be estimated similarly to our previous examples: the outer relation  $r$  is scanned once at the cost of  $F_r$  page transfers; the relation  $s$  is scanned once per group of  $M - 2$  pages of  $r$ , that is,  $\lceil \frac{F_r}{M-2} \rceil$ . Thus, the cost (excluding the cost of writing the output to disk) is

$$\underline{F_r + F_s \lceil \frac{F_r}{M-2} \rceil} \quad \textbf{10.3}$$

**Example 10.5.1 (An Estimate of a Block-Nested Loops Join).** In our example, if  $M = 102$ , the cost will go down to  $1000 + 100 \times 10 = 2000$ . And, if the smaller relation,  $s$ , is scanned in the outer loop, the cost will be even lower:  $100 + 100 \times 10 = 1100$  (or 11 seconds, assuming 10 ms per page I/O)—quite a reduction from  $10^6$ , the cost of the original, naive implementation. ■

This example shows again that the order of the loops matters. Looking at the general formula (10.3) we can observe that  $F_s \lceil \frac{F_r}{M-2} \rceil$  and  $F_r \lceil \frac{F_s}{M-2} \rceil$  are approximately the same. Therefore, if  $F_r > F_s$  then  $F_r + F_s \lceil \frac{F_r}{M-2} \rceil > F_s + F_r \lceil \frac{F_s}{M-2} \rceil$ .

**It is always cheaper to scan the smaller relation in the outer loop.**

**Index-nested loops join.** The next idea is to use indices. This technique achieves particularly good results if the number of tuples in  $r$  and  $s$  that match on the attributes  $A$  and  $B$  is small compared to the size of the file.

Suppose that relation  $s$  has an index on attribute  $B$ . Then, instead of scanning  $s$  in the inner loop, we can use the index to find the matching tuples:

---

```

foreach  $t \in r$  do {
    Use the index on  $B$  to find all tuples  $t' \in s$  such that  $t.A = t'.B$ 
    Output  $\langle t, t' \rangle$  for each such  $t'$ 
}

```

---

To estimate the cost of this method, we have to take into account the type of index and whether it is clustered or not. The number of matching tuples in  $s$  also matters.

If the index is a  $B^+$  tree, the cost of finding the first leaf node for the matching tuple in  $s$  is 2 to 4 I/O operations, depending on the size of the relation. In a hash-based index, it is about 1.2, if the hash function is well chosen. The next question is how many I/O operations are required to fetch the matching tuples in  $s$ , and the answer depends on the number of matching tuples and whether the index is clustered or unclustered.

If the index is unclustered, the number of I/Os needed to retrieve all matching tuples can be as high as the number of pages in  $s$  (this cost is bounded both by the number of pages in  $s$  and the number of matching tuples). So unclustered indices are not very useful for index-nested loops joins, *unless* the number of matching tuples is small (for instance, if  $B$  is a candidate key of  $s$ ). For clustered indices, all matching tuples are likely to be in the same or adjacent disk blocks, so the number of I/Os needed to retrieve them is typically 1 or 2. Thus, in case of the clustered index the cost estimate is

$$F_r + (\rho + 1) \times \tau_r$$

where  $\rho$  is the number of I/Os needed to retrieve the leaf node of a  $B^+$  tree index or to find the correct bucket of a hash index (we assume that the index is not integrated with the data file and that all matching tuples fit in one page, which is where the 1 comes from). In case of an unclustered index, the cost is

$$F_r + (\rho + \mu) \times \tau_r$$

where  $\mu$  is the average number of matching tuples in  $s$  per tuple in  $r$ .<sup>1</sup>

**Example 10.5.2 (An Estimate of an Index-Nested Loops Join).** Let us return to our example and compare this cost with the cost of block-nested loops joins. Assuming that  $\rho$  is 2 (our relations are fairly small), we obtain  $1000 + 3 \times 10,000 = 31,000$  in the case of a clustered index—much higher than in the case of block-nested loops. However, if we switch  $r$  and  $s$  in the nested loop, the costs of index- and block-nested loops are much closer:  $100 + 3 \times 1000 = 3100$  versus 1100. ■

Still, in this example indices seem to be losing to block-nested loops by a large margin. Why consider indices at all? It turns out that indexed loops have one remarkable property: the cost is not significantly affected by the size of the inner relation, as can be seen from the above formulas. So, for example, if we use  $s$  in the inner loop and its size grows to 10,000 pages (100,000 tuples), the cost of block-nested loops joins grows to  $1000 + 10,000 \times 10 = 101,000$  page transfers. In contrast, the cost of index-nested loops joins increases much more conservatively:  $1000 + 3 \times 10,000 = 31,000$ . Thus, indexed joins tend to work better when relations in the join are fairly large and one is much larger than the other.

<sup>1</sup> Assuming that the number of matching tuples in  $s$  per tuple in  $r$  is always less than the number of pages in  $s$ , which is typically the case.

FIGURE 10.7 The merge step of the sort-merge join algorithm.

```

Input: relation  $r$  sorted on attribute  $A$ ;  

       relation  $s$  sorted on attribute  $B$   

Output:  $r \bowtie_{A=B} s$ 

Result := {}                                // initialize Result
 $t_r$  := getFirst( $r$ )                         // get first tuple
 $t_s$  := getFirst( $s$ )
while !eof( $r$ ) and !eof( $s$ ) do {
    while !eof( $r$ ) &&  $t_r.A < t_s.B$  do
         $t_r$  := getNext( $r$ )                  // get next tuple
    while !eof( $s$ ) and  $t_r.A > t_s.B$  do
         $t_s$  := getNext( $s$ )
    if  $t_r.A = t_s.B = c$  then {                // for some constant  $c$ 
        Result :=  $(\sigma_{A=c}(r) \times \sigma_{B=c}(s)) \cup Result$ ;
         $t_r$  := the next tuple  $t \in r$  where  $t.A > c$ ;
    }
}
return Result;

```

---

### 10.5.2 Sort-Merge Join

The idea behind sort-merge is first to sort each relation on the join attributes and then to find matching tuples using a variation of the merge procedure, that is, scanning both relations simultaneously and comparing the join attributes. When a match is found, the joined tuple is added to the result.

The algorithm for this merge step is shown in Figure 10.7. The algorithm scans the relations  $r$  and  $s$  until a match on the attributes  $A$  and  $B$  is found. When this happens, all possible combinations (the Cartesian product) of the matching tuples are added to the result and the scan resumes.

Let us now estimate the cost of the sort-merge join in terms of the number of page transfers. Obviously, we must pay the usual price to sort the relations  $r$  and  $s$ :  $2F_r[\log_{M-1} F_r] + 2F_s[\log_{M-1} F_s]$  (assuming that  $M$  buffers are available).

The cost of the merging step consists of the cost of scanning  $r$  and  $s$ , which is  $F_r + F_s$  I/Os, plus the cost of computing  $\sigma_{A=c}(r) \times \sigma_{B=c}(s)$  for each match between  $r$  and  $s$ . At first, it seems that  $\sigma_{A=c}(r) \times \sigma_{B=c}(s)$  can be computed during the scan of  $r$  and  $s$  in Figure 10.7. However, if  $\sigma_{A=c}(r)$  does not fit in main memory, computing the Cartesian product might require additional scans of  $\sigma_{B=c}(s)$ . The best way to compute this product would then be the block-nested loops join algorithm. The actual number of page transfers here depends on the sizes of  $\sigma_{A=c}(r)$  and  $\sigma_{B=c}(s)$ , and on the amount of available memory. Typically, however, these subrelations are small and can fit in the available buffer, so the additional scans of  $\sigma_{B=c}(s)$  can be avoided.

In this lucky case, the I/O cost of  $\sigma_{A=c}(r) \times \sigma_{B=c}(s)$  is zero and the entire sort-merge join takes  $2F_r \lceil \log_{M-1} F_r \rceil + 2F_s \lceil \log_{M-1} F_s \rceil + F_r + F_s + \text{cost of outputting the result}$ .

**Brain Teaser:** What is the maximum size of  $\sigma_{A=c}(r) \times \sigma_{B=c}(s)$  in terms of the sizes of  $r$  and  $s$ ?

**An optimization.** A more careful analysis of the sort-merge algorithm shows that one can save the cost of one scan of  $r$  and  $s$ . The idea is to combine the scan needed for the merging step with the final stage of sorting  $r$  and  $s$ —analogously to the optimization for the union and difference operators discussed earlier. This can be accomplished as follows.

First,  $r$  and  $s$  are sorted in parallel, each using  $\frac{M}{2}$  buffer pages. (We could split the buffer into unequal chunks depending on the relative sizes of the files, but we will ignore this possible enhancement.) The final stage of sorting  $r$  consists of merging all the remaining sorted runs of  $r$  into one final sorted relation. Similarly, the final stage of sorting  $s$  merges the remaining sorted runs of  $s$ . Instead of performing these final steps, we can modify the algorithm in Figure 10.7 to perform a generalized merge of the set of final sorted runs of  $r$  with the set of final sorted runs of  $s$ . Since the number of runs in each file is  $\leq (\frac{M}{2} - 1)$ , we can use one buffer page for scanning each run of  $r$  and  $s$ . One more page will be used for the output of the merge.

The new merge algorithm looks like the old one except that  $r$  is now understood as being a set of final runs of the first relation to be joined and  $s$  is viewed as a set of final runs of the second relation. The operation `getNext` is changed so that it will return the tuple with the lowest value of the join attributes in  $r$  and  $s$ , respectively. Details are left to Exercise 10.10.

The overall cost of the optimized algorithm is:  $2F_r \lceil \log_{\frac{M}{2}-1} F_r \rceil + 2F_s \lceil \log_{\frac{M}{2}-1} F_s \rceil$  (to sort and merge) plus the cost of outputting the final result (which is the same as before). Note that for large  $M$  and  $F$ ,  $\log_{\frac{M}{2}-1} F = (\log_{M-1} F) \times \log_{\frac{M}{2}-1} (M-1) \approx \log_{M-1} F$ , which means that we have eliminated the cost of one scan of  $r$  and  $s$ .

**Example 10.5.3 (An Estimate of a Sort-Merge).** For our running example, we have the following relation sizes in blocks— $F_r = 1000$ ,  $F_s = 100$ —and the buffer dedicated to our join operation has 102 pages. This means  $s$  can be sorted in 200 page transfers and  $r$  in  $2 \times 1000 \times \lceil \log_{101} 1000 \rceil = 4000$  page transfers. Assuming that during the merge step of the join the matching tuples all fit in  $M - 1$  pages, this step can be done without the need for an additional scan of  $r$  and  $s$  (by combining this step with the final merge performed while sorting  $r$  and  $s$ , as explained earlier). Thus, the whole join will take 4200 page transfers. ■

It may seem that the block-nested loops algorithm is better than sort-merge in this particular case. However, just as with the index-nested loops method, the asymptotic behavior of sort-merge is better than that of block-nested loops. When the sizes of  $r$  and  $s$  grow, the cost of block-nested loops grows quadratically— $O(F_r F_s)$ —while the cost of sort-merge join increases much more slowly (assuming that  $\sigma_{A=c}(r)$  and  $\sigma_{B=c}(s)$  are small, as discussed above)— $O(F_r \log F_r + F_s \log F_s)$ .

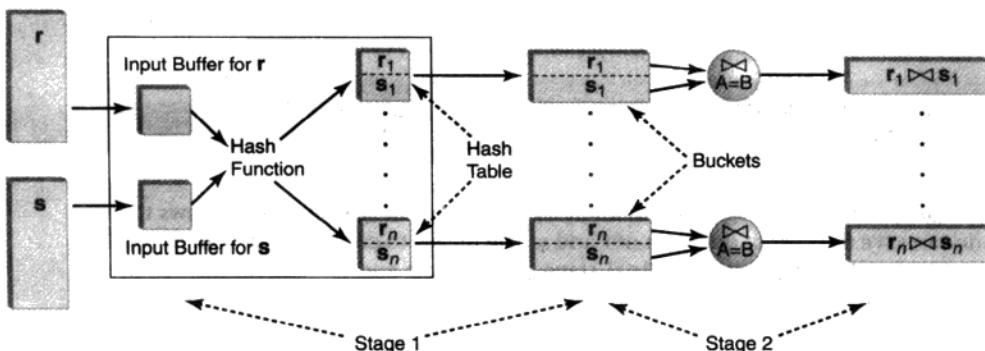


FIGURE 10.8 Hash join.

### 10.5.3 Hash Join

One way to compute a join,  $r \bowtie_{A=B} s$ , is to preprocess the relations  $r$  and  $s$  so that the tuples that possibly match will be placed on the same or adjacent pages. Such preprocessing eliminates the need for repeated scans of the inner-loop relation and is the basic idea behind the sort-merge technique described above. However, sorting is just one of the possible preprocessing techniques. Alternatively, we can use hashing to make sure that matching tuples are placed close to each other. We used this technique earlier, when we needed to place duplicates tuples (that arise due to projection or set-theoretic operations) close to each other. Clearly, the problem of identifying the duplicates is a special case of the problem we now face: identifying the tuples that have the same value for one or more attributes (e.g.,  $A$  and  $B$  above). The idea is illustrated in Figure 10.8.

The hash-join method first hashes each input relation onto the hash table, where  $r$  is hashed on attribute  $A$  and  $s$  is hashed on attribute  $B$ . This has the effect that the tuples of  $r$  and  $s$  that can *possibly* match are put in the same bucket.

In the second stage, the  $r$  half and the  $s$  half of each bucket are joined to produce the final result. If both halves fit in main memory, all of these joins can be done at the cost of a single scan of  $r$  and  $s$ . If the buckets are too large for main memory, other join techniques can be tried. Typically in this case, the  $r$  portion of each bucket is further partitioned by hashing on the attribute  $A$  using a different hash function. Then the  $s$  portion of the corresponding bucket is scanned. In the process, each tuple of  $s$  is hashed on attribute  $B$  using the new hash function, and matching tuples in  $r$  are identified.

Assuming that each bucket fits in memory, we can join  $r$  and  $s$  at the cost of three I/Os per page of each relation:  $3(F_r + F_s)$ . First,  $r$  and  $s$  must be input and the resulting buckets output (2 I/Os per page). Then each bucket must be input to join the two parts of the bucket. This requires one additional scan for each relation.

(recall that we do not include the cost of dumping the final result of the join on disk). In our running example, the cost is 3300 page transfers, which is higher than the cost of block-nested loops but the asymptotic behavior of hash join is better. In fact, if each hash bucket produced at the first stage of the algorithm fits in main memory, the cost is linear in the size of  $r$  and  $s$ . This makes hash join the best among all of the methods considered so far. However, it is important to realize that hash-join heavily depends on the choice of hash function and can be easily subverted by an unfortunate data skew (what if all tuples are hashed into the same bucket?). In addition, hash joins can be used only for equi-joins and are inappropriate for more general join conditions, such as inequalities.

## 10.6 Multirelational Joins \*

In Section 9.7, we discussed join indices and their use in computing joins. The algorithm for computing a join of the form  $p \bowtie_{A=B} q$  works by scanning the join index and fetching the tuples whose rids are found in the index entries.

The actual computation is essentially similar to the indexed loop join, with  $p$  scanned in the outer loop, except that we use the join index to locate the tuples of  $q$  instead of a general-purpose index on attribute  $B$  of  $q$ . The advantage of the join index over other index types in this case is that it does not need to be searched: since all matching tuples are already associated with each other, the index can simply be scanned, the pairs or rids of the matching tuples fetched, and the tuples joined.

The idea underlying join indices can be extended to multirelational joins, where an index can be created to relate rids of more than two tuples. For instance, in a 3-way join,  $p \bowtie q \bowtie r$ , a join index consists of triples of the form  $\langle p, q, r \rangle$ , where  $p$  is a rid of a tuple in relation  $p$ ,  $q$  is a rid of a matching tuple in  $q$ , and  $r$  is a rid of a matching tuple in  $r$ . The triples are sorted in ascending order of rids beginning with the first column of the index, then the second, and then the third (the index can also be a  $B^+$  tree).

With such an index, the join can be computed with a simple loop that scans the join index. For each triple  $\langle p, q, r \rangle$  in the index, the tuples corresponding to the rids  $p$ ,  $q$ , and  $r$  are fetched. Since the index is sorted on column 1 first, the join is performed in a single scan of the index and of the relation  $p$ . However, the relations  $q$  and  $r$  might have to be accessed many times. Indeed, if  $N$  is the average number of matching tuples in  $q$  per tuple in  $p$  and  $M$  is the average number of matching tuples in  $r$  per tuple in  $p$ , then, to compute the join,  $|p| \times N$  pages of  $q$  and  $|p| \times M$  pages of  $r$  might have to be retrieved.

Multiway join indices are especially popular for speeding up star joins—a common type of join used in online analytical processing.

A **star join** is a multiway join of the form  $r \bowtie_{cond_1} r_1 \bowtie_{cond_2} r_2 \bowtie_{cond_3} \dots$ , where each  $cond_i$  is a join condition that involves the attributes of  $r$  and  $r_i$  only. In other words, there are no conditions that relate the tuples of  $r_i$  and  $r_j$  directly, and all matching is done through the tuples of  $r$ . An example of a star join is shown in

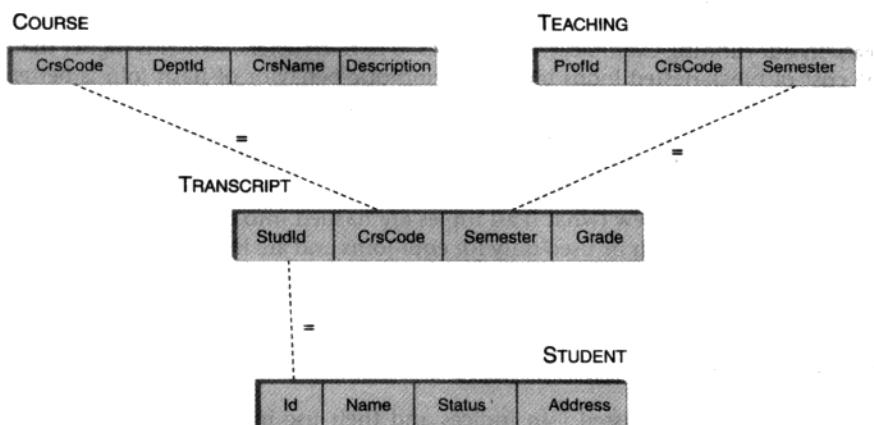


FIGURE 10.9 Star join.

OPTIONAL

Figure 10.9, where the “satellite” relations COURSE, TEACHING, and STUDENT are joined with the “star” relation TRANSCRIPT using equi-join conditions that match the attributes of the satellite relations only to the attributes of the star relation.

One reason that multiway join indices are good for computing star joins is that the join index of a star join is easier to maintain than the join index of a general multiway join (Exercise 10.14). Furthermore, computing a general multiway join using a join index can be expensive. Consider a join,  $r \bowtie r_1 \bowtie r_2 \bowtie \dots \bowtie r_n$ , and suppose that  $N$  is the average number of matching tuples in  $r_i$  per tuple in  $r$ . Then, using analysis similar to that for 3-way joins, a join index computation might need to access  $|r| \times N \times n$  pages.

Fortunately, star joins have more promising methods. One, described in [O’Neil and Graefe 1995], takes advantage of bitmapped join indices, introduced in Section 9.7.2. Instead of one join index that involves  $n$  relations, we can use one bitmapped join index,  $\beta_i$ , for each partial join  $r_i \bowtie r$ . Each  $\beta_i$  is a collection of pairs  $(v, \text{bitmap})$ , where  $v$  is a rid of a tuple in  $r_i$  and  $\text{bitmap}$  has 1 in the  $k$ th position if and only if the  $k$ th tuple in  $r$  joins with the  $r_i$ ’s tuple represented by  $v$ . We can then scan  $\beta_i$  and logically OR all of the bitmaps. This will give us the rids of all tuples in  $r$  that can join with *some* tuple in  $r_i$ . After obtaining such an ORed bitmap for each satellite relation  $r_i$ , where  $i = 1, \dots, n$ , we can logically AND these bitmaps to obtain the rids of all tuples in  $r$  that join with *some* tuple in each  $r_i$ . In other words, this procedure prunes away all tuples in  $r$  that *do not* participate in the star join. The rationale is that there will be only a small number of tuples left, so the join can be computed inexpensively by a brute-force technique like nested loops.

Join indices and star join optimization are supported by the recent versions of commercial DBMSs from the major vendors, such as IBM’s DB/2, Oracle, and Microsoft’s SQL Server.

## 10.7 Computing Aggregate Functions

Generally, computing aggregate functions (such as AVG or COUNT) in a query involves a complete scan of the query output. The only issue here is the computation of aggregates in the presence of the GROUP BY *attrs* statement. Once again, the problem reduces to finding efficient techniques for partitioning the tuples according to the values of certain attributes. We have identified three such techniques so far:

1. Sorting
2. Hashing
3. Indexing

All three techniques provide efficient ways to access the groups of tuples specified by the GROUP BY clause. All that remains is to apply the aggregate functions to the member tuples of these groups.

### BIBLIOGRAPHIC NOTES

Sort-based evaluation techniques for relational operators are discussed in [Blasgen and Eswaran 1977]; hash-based techniques are covered in [DeWitt et al. 1984; Kit-suregawa et al. 1983]. Good surveys of techniques for evaluating relational operators and additional references can be found in [Graefe 1993; Chaudhuri 1998]. The use of join indices for computing multirelational joins is studied in [Valduriez 1987], and techniques for computing various relational operators with the help of bitmap indices are discussed in [O'Neil and Graefe 1995; O'Neil and Quass 1997].

### EXERCISES

- 10.1 Consider the use of unclustered B<sup>+</sup> trees for external sorting. Let  $R$  denote the number of data records per disk block, and let  $F$  be the number of blocks in the data file. Estimate the cost of such a sorting procedure as a function of  $R$  and  $F$ . Compare this cost to merge-based external sorting. Consider the cases of  $R = 1, 10$ , and  $100$ .
- 10.2 Estimate the cost of the sort-based projection assuming that, during the initial scan (where tuple components are deleted), the size of the original relation shrinks by the factor  $\alpha < 1$ .
- 10.3 Consider hash-based evaluation of the projection operator. Assume that all buckets are about the same size but do not fit in main memory. Let  $N$  be the size of the hash table measured in memory pages,  $F$  be the size of the original relation measured in pages, and  $\alpha < 1$  be the reduction factor due to projection. Estimate the number of page transfers to and from the disk needed to compute the projection.
- 10.4 Give an example of an instance of the TRANSCRIPT relation (Figure 3.5) and a hash function on the attribute sequence (StudId, Grade) that sends two identical tuples in  $\pi_{\text{StudId}, \text{Semester}}(\text{TRANSCRIPT})$  into *different* hash buckets. (This shows that such a hash-based access path cannot be used to compute the projection.)

- 10.5 Clearly, the theoretical minimum for the selectivity of an access path is the number of pages that hold the output of the relational operator involved. What is the best theoretical upper bound on the selectivity of an access path when selection or projection operators are involved?
- 10.6 Based on the discussion in Section 10.4.2, give a precise definition of when an access path covers the use of projection, union, and set-difference operators.
- 10.7 Consider the expression

$$\sigma_{\text{StudId}=66666666 \wedge \text{Semester}='F1995' \wedge \text{Grade}='A'}(\text{TRANSCRIPT})$$

Suppose the following access paths are available:

- An unclustered hash index on StudId
- An unclustered hash index on Semester
- An unclustered hash index on Grade

Which of these access paths has the best selectivity, and which has the worst? Compare the selectivity of the worst access path (among the above three) to the selectivity of the file scan.

- 10.8 Compute the cost of  $r \bowtie_{A=B} s$  using the following methods:
- Nested loops
  - Block-nested loops
  - Index-nested loops with a hash index on B in s (consider both clustered and unclustered index)
- where r occupies 2000 pages, 20 tuples per page; s occupies 5000 pages, 5 tuples per page; and the amount of main memory available for a block-nested loops join is 402 pages. Assume that at most 5 tuples of s match each tuple in r.
- 10.9 In sort-based union and difference algorithms, the final scan—where the actual union or difference is computed—can be performed at no cost in I/O because this step can be combined with the last merge step during sorting of the relations involved. Work out the details of this algorithm.
- \*10.10 In the sort-merge join of  $r \bowtie s$ , the scan in the algorithm of Figure 10.7 can be performed at no cost in I/O because it can be combined with the final merging step of sorting r and s. Work out the details of such an algorithm.
- 10.11 Estimate the number of page transfers needed to compute  $r \bowtie_{A=B} s$  using a sort-merge join, assuming the following:
- The size of r is 1000 pages, 10 tuples per page; the size of s is 500 pages, 20 tuples per page.
  - The size of the main memory buffer for this join computation is 10 pages.
  - The Cartesian product of matching tuples in r and s (see Figure 10.7) is computed using a block-nested loops join.
  - $r.A$  has 100 distinct values and  $s.B$  has 50 distinct values. These values are spread around the files more or less evenly, so the size of  $\sigma_{A=c}(r)$ , where  $c \in r.A$ , does not vary much with c.
- 10.12 The methods for computing joins discussed in Section 10.5 all deal with equijoins. Discuss their applicability to the problem of computing inequality joins, such as  $r \bowtie_{A < B} s$ .

**10.13** Consider a relation schema,  $R(A, B)$ , with the following characteristics:

- Total number of tuples: 1,000,000
- 10 tuples per page
- Attribute A is a candidate key; range is 1 to 1,000,000
- Clustered  $B^+$  tree index of depth 4 on A
- Attribute B has 100,000 distinct values
- Hash index on B

Estimate the number of page transfers needed to evaluate each of the following queries for each of the proposed methods:

- $\sigma_{A < 3000}$ : sequential scan; index on A
- $\sigma_{A > 3000 \wedge A < 3200 \wedge B = 5}$ : index on A; index on B
- $\sigma_{A \neq 22 \wedge B \neq 66}$ : sequential scan; index on A; index on B

**10.14** Design an algorithm for incremental maintenance of a join index for a multiway star join.

**10.15** Design a join algorithm that uses a join index. Define the notion of a *clustered* join index (there are three possibilities in the case of a binary join!) and consider the effect of clustering on the join algorithm.



# 11

## An Overview of Query Optimization

This chapter is an overview of relational query optimization techniques typically used in database management systems. Our goal here is not to prepare you for a career as a DBMS implementor but rather to make you a better application designer or database administrator. Just as the knowledge of the evaluation techniques used in relational algebra can help you make better physical design, an understanding of the principles of query optimization can help you formulate SQL queries that stand a better chance of being efficiently implemented by the query processor.

Relational query optimization is a fascinating example of tackling a problem of immense computational complexity with relatively simple heuristic search algorithms. A more extensive treatment of the subject can be found in [Garcia-Molina et al. 2000].

### 11.1 Query Processing Architecture

When the user submits a query, it is first parsed by the DBMS, which verifies the syntax and type correctness of the query. Being a declarative language, SQL does not suggest concrete ways to evaluate its queries. Therefore, a parsed query has to be converted into a relational algebra expression, which can be evaluated directly using the algorithms presented in Chapter 10. A typical SQL query such as

---

```
SELECT DISTINCT TargetList
  FROM REL1 V1, ..., RELn Vn
 WHERE Condition
```

---

11.1

is normally translated into the following relational algebraic expression:

---

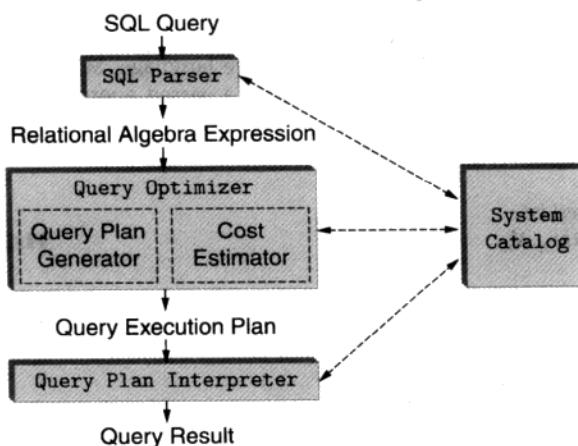
$$\pi_{TargetList}(\sigma_{Condition'}(REL_1 \times \dots \times REL_n))$$

---

where *Condition'* is *Condition* converted from SQL syntax to relational algebra form. Section 10.1 has an example of such a transformation from SQL to the relational algebra.

While the above algebraic expressions are straightforward and easy to produce, it might take ages to evaluate them. For one thing, they contain Cartesian products, so a join of four 100-block relations produces a  $10^8$ -block intermediate relation, which, with a disk speed of 10 ms/page, takes about 50 hours just to write out. Even if we manage to convert the Cartesian product into equi-joins (as explained in Section 11.2), we might still have to grapple with the long turnaround time (dozens of minutes) for the above query. It is the job of the **query optimizer** to bring this time down to seconds (or, for very complex queries, a few minutes).

A typical **rule-based query optimizer** uses a set of rules (e.g., an access path based on an index is better than a table scan) to construct a **query execution plan**. A **cost-based query optimizer** estimates the cost of query plans based on statistics maintained by the DBMS and uses this information, in addition to the rules, to choose a plan. The two main components of a cost-based query optimizer are the **query execution plan generator** and the **plan cost estimator**. A query execution plan can be thought of as a relational expression with concrete evaluation methods (or *access paths*, as we called them in Chapter 10) attached to each occurrence of a relational operator in the expression. Thus, the main job of the optimizer is to propose a single plan that can evaluate the given relational expression at a “reasonably cheap” cost according to the cost estimator. This plan is then passed to the **query plan interpreter**, a software component directly responsible for query evaluation according to the given plan. The overall architecture of query processing is depicted in Figure 11.1.



**FIGURE 11.1** Typical architecture for DBMS query processing.

## 11.2 Heuristic Optimization Based on Algebraic Equivalences

The heuristics used in relational query evaluation are (for the most part) based on simple observations, such as that joining smaller relations is better than joining large ones, that performing an equi-join is better than computing a Cartesian product, and that computing several operations in just one relation scan is better than doing so in several scans. Most of these heuristics can be expressed in the form of relational algebra transformations, which take one expression and produce a different but equivalent expression. Not all transformations are optimizations by themselves. Sometimes they yield less efficient expressions. However, relational transformations are designed to work with other transformations to produce expressions that are better overall.

We now present a number of heuristic transformations used by the query optimizers.

### Selection and projection-based transformations.

- $\sigma_{cond_1 \wedge cond_2}(R) \equiv \sigma_{cond_1}(\sigma_{cond_2}(R))$ . This transformation is known as **cascading of selections**. It is not an optimization per se, but it is useful in conjunction with other transformations (see the discussion on page 408 of pushing selections and projections through joins).
- $\sigma_{cond_1}(\sigma_{cond_2}(R)) \equiv \sigma_{cond_2}(\sigma_{cond_1}(R))$ . This transformation is called **commutativity of selection**. Like cascading, it is useful in conjunction with other transformations.
- $\pi_{attr}(R) \equiv \pi_{attr}(\pi_{attr'}(R))$ , if  $attr \subseteq attr'$  and  $attr'$  is a subset of the attributes of  $R$ . This equivalence is known as **cascading of projections** and is used primarily with other transformations.
- $\pi_{attr}(\sigma_{cond}(R)) \equiv \sigma_{cond}(\pi_{attr}(R))$ , if  $attr$  includes all attributes used in  $cond$ . This equivalence is known as the **commutativity of selection and projection**. It is usually used as a preparation step for pushing a selection or a projection through the join operator.

**Cross product and join transformations.** The transformations used for cross products and joins are the usual commutativity and associativity rules for these operators.

- $R \bowtie S \equiv S \bowtie R$
- $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$
- $R \times S \equiv S \times R$
- $R \times (S \times T) \equiv (R \times S) \times T$

These rules can be useful in conjunction with the various nested loops evaluation strategies. As we saw in Chapter 10, it is generally better to scan the smaller relation in the outer loop, and the above rules can help maneuver the relations into the right

positions. For instance,  $\text{BIGGER} \bowtie \text{SMALLER}$  can be rewritten as  $\text{SMALLER} \bowtie \text{BIGGER}$ , which intuitively corresponds to the query optimizer deciding to use  $\text{SMALLER}$  in the outer loop.

The commutativity and associativity rules (at least in the case of the join) can reduce the size of the intermediate relation in the computation of a multirelational join. For instance,  $S \bowtie T$  can be much smaller than  $R \bowtie S$ , in which case the computation of  $(S \bowtie T) \bowtie R$  might take fewer I/O operations than the computation of  $(R \bowtie S) \bowtie T$ . The associativity and commutativity rules can be used to transform the latter expression into the former.

In fact, the commutativity and associativity rules are largely responsible for the many alternative evaluation plans that might exist for the same query. A query that involves the join of  $N$  relations can have  $T(N) \times N!$  query plans just to handle the join, where  $T(N)$  is the number of different binary trees with  $N$  leaf nodes. ( $N!$  is the number of permutations of  $N$  relations, and  $T(N)$  is the number of ways a particular permutation can be parenthesized.) This number grows very rapidly and is huge even for very small  $N$ .<sup>1</sup> A similar result holds for other commutative and associative operations (e.g., union), but our main focus is on join because it is the most expensive operation to compute.

The job of the query optimizer is to estimate the cost of these plans (which can vary widely) and to choose one “good” plan. Because the number of plans is large, it can take longer to find a good plan than to evaluate the query by brute force. (It is faster to perform  $10^6$  I/Os than  $15!$  in-memory operations.) To make query optimization practical, an optimizer typically looks at only a small subset of all possible plans, and its cost estimates are approximate at best. Therefore, query optimizers are very likely to miss the optimal plan and are actually designed only to find one that is “reasonable.” In other words, the “optimization” in “query optimizer” should always be taken with a grain of salt since it does not adequately describe what is being done by that component of the DBMS architecture.

### **Pushing selections and projections through joins and Cartesian products.**

- $\sigma_{cond}(R \times S) \equiv R \bowtie_{cond} S$ . This rule is used when  $cond$  relates the attributes of both  $R$  and  $S$ . The basis for this heuristic is the belief that Cartesian products should never be materialized. Instead, selections must always be combined with Cartesian products and the techniques for computing joins should be used. By applying the selection condition as soon as a row of  $R \times S$  is created, we can save one scan and avoid storing a large intermediate relation.
- $\sigma_{cond}(R \times S) \equiv \sigma_{cond}(R) \times S$ , if the attributes used in  $cond$  all belong to  $R$ . This heuristic is based on the idea that if we absolutely must compute a Cartesian product, we should make the relations involved as small as possible. By pushing the selection down to  $R$ , we hope to reduce the size of  $R$  *before* it is used in the cross product.

<sup>1</sup> When  $N = 4$ ,  $T(4)$  is 5, and the number of all plans is 120. When  $N = 5$ ,  $T(5) = 14$ , and the number of all plans is 1680.

- $\sigma_{cond}(R \bowtie_{cond'} S) \equiv \sigma_{cond}(R) \bowtie_{cond'} S$ , if the attributes in *cond* all belong to R. The rationale here is the same as for Cartesian products. Computing a join can be very expensive, and we must try to reduce the size of the relations involved. Note that if *cond* is a conjunction of comparison conditions, we can push each conjunct separately to either R or S as long as the attributes named in the conjunct belong to only one relation.
- $\pi_{attr}(R \times S) \equiv \pi_{attr}(\pi_{attr'}(R) \times S)$ , if  $attributes(R) \supseteq attr' \supseteq (attr \cap attributes(R))$ , where *attributes*(R) denotes the set of all the attributes of R. The rationale for this rule is that, by pushing the projection inside the Cartesian product, we reduce the size of one of its operands. In Chapter 10, we saw that the I/O complexity of the join operation (of which  $\times$  is a special case) is proportional to the number of pages in the relations involved. Thus, by applying the projection early we might reduce the number of page transfers needed to evaluate the cross product.
- $\pi_{attr}(R \bowtie_{cond} S) \equiv \pi_{attr}(\pi_{attr'}(R) \bowtie_{cond} S)$ , if  $attr' \subseteq attributes(R)$  is such that it contains all the attributes that R has in common with either attr or cond. The potential benefit here is the same as for the cross product. The important additional requirement is that attr' must include those attributes of R that are mentioned in cond. If some of these attributes are projected out, the expression  $\pi_{attr'}(R) \bowtie_{cond} S$  will not be syntactically correct. This requirement is unnecessary in the case of the Cartesian product since no join condition is involved.

The rules for pushing selections and projections through joins and cross products are especially useful when combined with the rules for cascading  $\sigma$  and  $\pi$ . For instance, consider the expression  $\sigma_{c_1 \wedge c_2 \wedge c_3}(R \times S)$ , where  $c_1$  involves the attributes of both R and S,  $c_2$  involves only the attributes of R, and  $c_3$  involves only the attributes of S. We can transform this expression into one that can be evaluated more efficiently by first cascading the selections, then pushing them down and finally eliminating the Cartesian product:

$$\sigma_{c_1 \wedge c_2 \wedge c_3}(R \times S) \equiv \sigma_{c_1}(\sigma_{c_2}(\sigma_{c_3}(R \times S))) \equiv \sigma_{c_1}(\sigma_{c_2}(R) \times \sigma_{c_3}(S)) \equiv \sigma_{c_2}(R) \bowtie_{c_1} \sigma_{c_3}(S)$$

We can optimize the expressions that involve projections in a similar way. Consider, for instance,  $\pi_{attr}(R \bowtie_{cond} S)$ . Suppose that  $attr_1$  is a subset of the attributes in R such that  $attr_1 \supseteq attr \cap attributes(R)$  and such that  $attr_1$  contains all the attributes in cond. Let  $attr_2$  be a similar set for S. Then

$$\begin{aligned} \pi_{attr}(R \bowtie_{cond} S) &\equiv \pi_{attr}(\pi_{attr_1}(R \bowtie_{cond} S)) \equiv \pi_{attr}(\pi_{attr_1}(R) \bowtie_{cond} S) \\ &\equiv \pi_{attr}(\pi_{attr_2}(\pi_{attr_1}(R) \bowtie_{cond} S)) \equiv \pi_{attr}(\pi_{attr_1}(R) \bowtie_{cond} \pi_{attr_2}(S)) \end{aligned}$$

The resulting expression can be more efficient because it joins smaller relations.

**Using the algebraic equivalence rules.** Typically, the above algebraic rules are used to transform queries expressed in relational algebra into expressions that are believed to be better than the original. The word “better” here should not be understood literally because the criteria used to guide the transformation are heuristic. In fact, in the next section we will see that following through with all the suggested

transformations might not yield the best result. Thus, the outcome of the algebraic transformation step should yield a set of candidate queries, which must then be further examined using cost-estimation techniques discussed in Section 11.3. Here is a typical heuristic algorithm for applying algebraic equivalences:

1. Use the cascading rule for selection to break up the conjunctions in selection conditions. The result is a single selection transformed into a sequence of selection operators, each of which can be applied separately.
2. The previous step leads to greater freedom in pushing selections through joins and Cartesian products. We can now use the rules for commutativity of selection and for pushing selections through joins to propagate the selections as far inside the query as possible.
3. Combine the Cartesian product operations with selections to form joins. As we saw in Chapter 10, there are efficient techniques for computing joins, but little can be done to improve the computation of a Cartesian product. Thus, converting these products into joins is a potential time and space saver.
4. Use the associativity rules for joins and Cartesian products to rearrange the order of join operations. The purpose here is to come up with the order that produces the smallest intermediate relations. (Note that the size of the intermediate relations directly contributes to overhead, so reducing these sizes speeds up query processing.) Techniques for the estimation of the size of intermediate relations are discussed in Section 11.3.
5. Use the rules for cascading projections and for pushing them into queries to propagate projections as far into the query as possible. This can potentially speed up the computation of joins by reducing the size of the operands.
6. Identify the operations that can be processed in the same pass to save time writing the intermediate results to disk. This technique is called *pipelining* and is illustrated in Section 11.3.

### 11.3 Estimating the Cost of a Query Execution Plan

As defined earlier, a query execution plan is more or less a relational expression with concrete evaluation methods (access paths) attached to each operation. In this section, we take a closer look at this concept and discuss ways to evaluate the cost of a plan to compute query results.

For discussion purposes, it is convenient to represent queries as trees. In a **query tree** each inner node is labeled with a relational operator and each leaf is labeled with a relation name. Unary relational operators have only one child; binary operators have two. Figure 11.2 presents four query trees corresponding to the following equivalent relational expressions, respectively:

$$\pi_{\text{Name}}(\sigma_{\text{DeptId}='CS' \wedge \text{Semester}='F1994'}(\text{PROFESSOR} \bowtie_{\text{Id}=\text{ProfId}} \text{TEACHING})) \quad 11.2$$

$$\pi_{\text{Name}}(\sigma_{\text{DeptId}='CS'}(\text{PROFESSOR}) \bowtie_{\text{Id}=\text{ProfId}} \sigma_{\text{Semester}='F1994'}(\text{TEACHING})) \quad 11.3$$

$$\pi_{\text{Name}}(\sigma_{\text{Semester}='F1994'}(\sigma_{\text{DeptId}='CS'}(\text{PROFESSOR}) \bowtie_{\text{Id}=\text{ProfId}} \text{TEACHING})) \quad 11.4$$

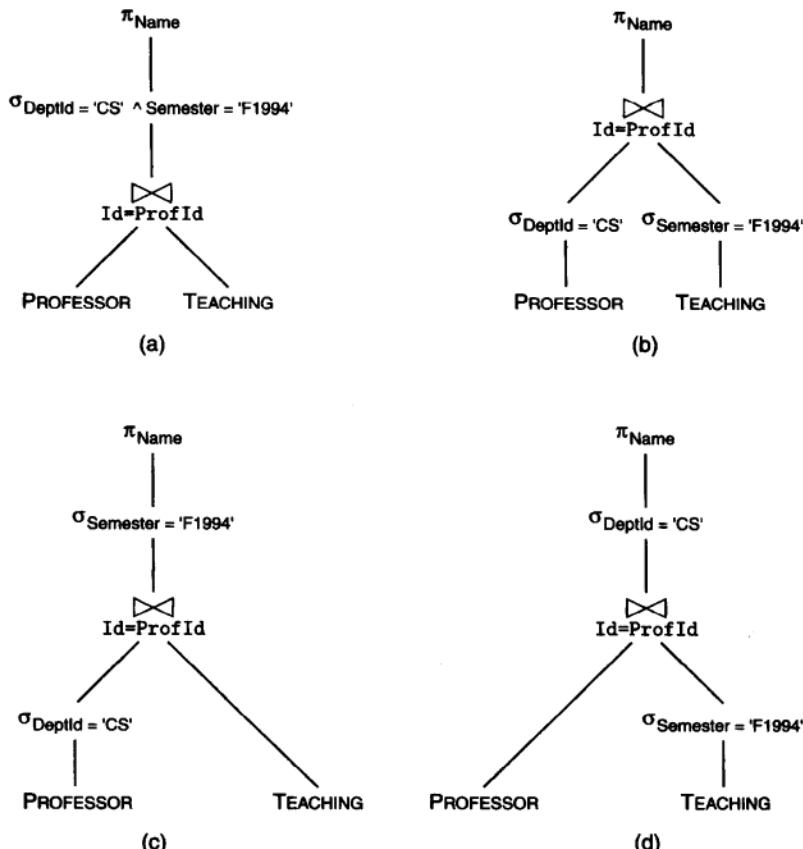


FIGURE 11.2 Query trees for relational expressions (11.2) through (11.5).

$\pi_{\text{Name}}(\sigma_{\text{DeptId} = 'CS'}(\text{PROFESSOR} \bowtie_{\text{Id}=\text{ProfId}} \sigma_{\text{Semester} = 'F1994'}(\text{TEACHING})))$  **11.5**

The relations PROFESSOR and TEACHING were described in Figure 3.5, page 39.

Expression (11.2), corresponding to Figure 11.2(a), is what a query processor might initially generate from the SQL query (after combining the selection  $\text{Id} = \text{ProfId}$  with the cross product)

---

```

SELECT P.Name
FROM PROFESSOR P, TEACHING T
WHERE P.Id = T.ProfId AND T.Semester = 'F1994'
      AND P.DeptId = 'CS'
  
```

---

**11.6**

The second expression, (11.3), corresponding to Figure 11.2(b), is obtained from the first by fully pushing the selections through the join, as suggested by the heuristic rules in the previous section. The third and fourth expressions, corresponding to Figure 11.2(c) and (d), are obtained from (11.2) by pushing only part of the selection condition down to the actual relations.

We are now going to augment these query trees with specific methods for computing joins, selections, and so on, and thus produce query execution plans. We will then estimate the cost of each plan and choose the best one.

Suppose that the following information is available on these relations in the system catalog:

#### PROFESSOR

*Size:* 200 pages, 1000 records on professors in 50 departments (5 tuples/page).

*Indices:* clustered 2-level B<sup>+</sup> tree on DeptId, hash index on Id.

#### TEACHING

*Size:* 1000 pages, 10,000 teaching records for the period of 4 semesters (10 tuples/page).

*Indices:* clustered 2-level B<sup>+</sup> tree index on Semester, hash index on ProfId

We need one additional piece of information before we can proceed: the weight of the attribute Id in the relation PROFESSOR and the weight of ProfId in the relation TEACHING. In general, the **weight** of an attribute,  $A$ , in a relation,  $r$ , is the average number of tuples that match the different values of attribute  $A$ . In other words, weight is the average number of tuples in  $\sigma_{A=value}(r)$ , where the average is taken over all values of  $A$  in  $r$ .

The weights for various attributes are typically derived from the statistical information stored in the system catalog and maintained by the DBMS. Recent query optimizers go as far as maintaining *histograms* for the distribution of values in a particular attribute. Histograms give more precise information about how many tuples are likely to be selected for a given value of the attribute. Attribute weights are needed to estimate the cost of computing the join using index-based techniques, as well as to estimate the size of the result of all of the operations in our examples. Since intermediate results of the various operations might later be used as input to other operators, knowing the sizes is important for estimating the cost of each concrete plan. Section 12.6 discusses statistics and size estimation in more detail.

Returning to our example, we first need to find realistic weights for the attributes Id and ProfId. For the Id attribute of PROFESSOR, the weight must be 1, since Id is a key. For the weight of ProfId in TEACHING, let us assume that each professor is likely to have been teaching the same number of courses. Since there are 1000 professors and 10,000 teaching records, the weight of ProfId must be about 10. Let us now consider the four cases in Figure 11.2. In all of them, we assume that a 52-page buffer is available for evaluating the join and that there is a small amount of additional memory to hold some index blocks and other auxiliary information (the exact amount will be specified when necessary).

**Case a: selection not pushed.** One possibility to evaluate the join is the index-nested loops method. For instance, we can use the smaller relation, PROFESSOR, in the outer loop. Since the indices on Id and ProfId are not clustered *and* because each tuple in PROFESSOR is likely to match some tuple in TEACHING (generally, every professor teaches something), the cost can be estimated as follows.

- *To scan the PROFESSOR relation:* 200 page transfers.
- *To find matching tuples in TEACHING:* We can use 50 pages of the buffer to hold the pages of the PROFESSOR relation. Since there are 5 PROFESSOR tuples in each such page, and since each tuple matches 10 TEACHING tuples, the 50-page chunk of the PROFESSOR relation can, on average, match  $50 \times 5 \times 10 = 2500$  tuples of TEACHING. The index on the ProfId attribute of TEACHING is not clustered, so record IDs retrieved from it will not be sorted. As a result, the cost of fetching the matching rows of the data file (leaving aside for the moment the cost of fetching the IDs from the index), can be as much as 2500 page transfers. By sorting the record IDs of these matching tuples first, however (a technique described in Section 10.4.1), we can guarantee that the tuples will be fetched in no more than 1000 page transfers (the size of the TEACHING relation).<sup>2</sup> Since this trick must be performed four times (for each 50-page chunk of PROFESSOR), the total number of page transfers to fetch the matching tuples of TEACHING is 4000.
- *To search the index:* Since TEACHING has a hash index on ProfId, we can assume 1.2 I/Os per index search. For each ProfId, the search finds the bucket that contains the record IDs of all matching tuples (10 on average). These IDs can be retrieved in one I/O operation. Thus, the 10,000 matching record IDs of tuples in TEACHING can be retrieved 10 tuples per I/O—1000 I/Os in total. The total cost of the index search for all tuples is therefore 1200.
- *Combined cost:*  $200 + 4000 + 1200 = 5400$  page transfers.

Alternatively, we can use a block-nested loops join or a sort-merge join. For a block-nested loops join that utilizes a 52-page buffer of main memory, the inner relation, TEACHING, must be scanned 4 times. This leads to a smaller number of page transfers:  $200 + 4 \times 1000 = 4200$ . Note, however, that if the weight of ProfId in TEACHING is lower, the comparison between the index-nested and block-nested techniques can be very different (Exercise 11.4) since the index may become more effective in reducing the number of I/Os.

The result of the join is going to have 10,000 tuples (because Id is a key for PROFESSOR and every PROFESSOR tuple matches roughly 10 TEACHING tuples). Since every PROFESSOR tuple is twice the size of a TEACHING tuple, the resulting file will be three times the size of TEACHING—3000 pages.

Next we need to apply the selection and the projection operators. As the result of the join does not have any indices, we choose the file scan access path. Moreover, we can apply selection and projection during the same scan. Examining each tuple

<sup>2</sup> Note that we need extra space for sorting the record IDs. Since we have rids for 2500 tuples and each rid is typically 8 bytes long, we need about five 4K pages to hold all these rids in main memory.

in turn, we discard it if it does not satisfy the selection condition; if it does, we discard the attributes not named in the SELECT clause and output the result.

We could treat the join phase and the select/project phase separately, outputting the result of the join to an intermediate file and then inputting the file to do the select/project, but there is a better way. By interleaving the two phases, we can eliminate the I/O operations associated with creating and accessing the intermediate file. With this technique, called **pipelining**, join and select/project operate as coroutines. The join phase is executed until the available buffers in memory are filled, and then select/project takes over, emptying the buffers and outputting the result. The join phase is then resumed, filling the buffers, and the process continues until select/project outputs the last tuple. In pipelining, the output of one relational operator is “piped” to the input of the next relational operator—without saving the intermediate result on disk.

The resulting query execution plan is depicted in Figure 11.3(a). All in all, using the block-nested loops strategy, evaluating this plan takes  $4200 + \alpha \times 3000$  page I/Os, where 3000 is the size of the join (computed earlier) and  $\alpha$ , a number between 0 and 1, is the reduction factor due to selection and projection. We study the techniques for estimating this reduction factor in Section 11.4. The last component,  $\alpha \times 3000$ , represents the cost of writing the query result out on disk. Since this cost is the same for all plans, (a) through (d), we will ignore it in our further analysis.

**Case b: selection fully pushed.** The query tree in Figure 11.2(b) suggests a number of alternative query execution plans. First, if we push selections down to the leaf nodes of the tree (the relations PROFESSOR and TEACHING), then we can compute the relations  $\sigma_{\text{DeptId}=\text{'CS'}}(\text{PROFESSOR})$  and  $\sigma_{\text{Semester}=\text{'F1994'}}(\text{TEACHING})$  using the existing B<sup>+</sup> tree indices on DeptId and Semester. However, the resulting relations will not have any indices (unless the DBMS decides that it is worth building them, which incurs additional cost). In particular, we cannot make use of the hash indices on PROFESSOR.Id and TEACHING.ProfId. Thus, we must use block-nested loops or sort-merge to compute the join. The projection is then applied to the result of the join on the fly, while it is being written out to disk. In other words, we again use pipelining to minimize the overhead of applying the projection operator.

We estimate the cost of the plan, depicted in Figure 11.3(b), where the join is performed using block-nested loops. Since there are 1000 professors in 50 departments, the weight of DeptId in the PROFESSOR relation is 20; hence, the size of  $\sigma_{\text{DeptId}=\text{'CS'}}(\text{PROFESSOR})$  is about 20 tuples, or 4 pages. The weight of Semester in TEACHING is  $10,000/4 = 2500$  tuples, or 250 pages. Because the indices on DeptId and Semester are clustered, computing the selection will require the following I/Os: 4 (to access the two indices) + 4 (to access the qualifying tuples in PROFESSOR) + 250 (to access the qualifying tuples in TEACHING).

The results of the two selections do not need to be written out on disk. Instead, we can pipe  $\sigma_{\text{DeptId}=\text{'CS'}}(\text{PROFESSOR})$  and  $\sigma_{\text{Semester}=\text{'F1994'}}(\text{TEACHING})$  into the join operation, which is computed using block-nested loops. Since the first relation is only 4 pages long, we will keep it entirely in main memory. As we compute the second relation, we join the results with the 4-page  $\sigma_{\text{DeptId}=\text{'CS'}}(\text{PROFESSOR})$  relation and pipe the result further into the operation  $\pi_{\text{Name}}$ . After the entire selection of

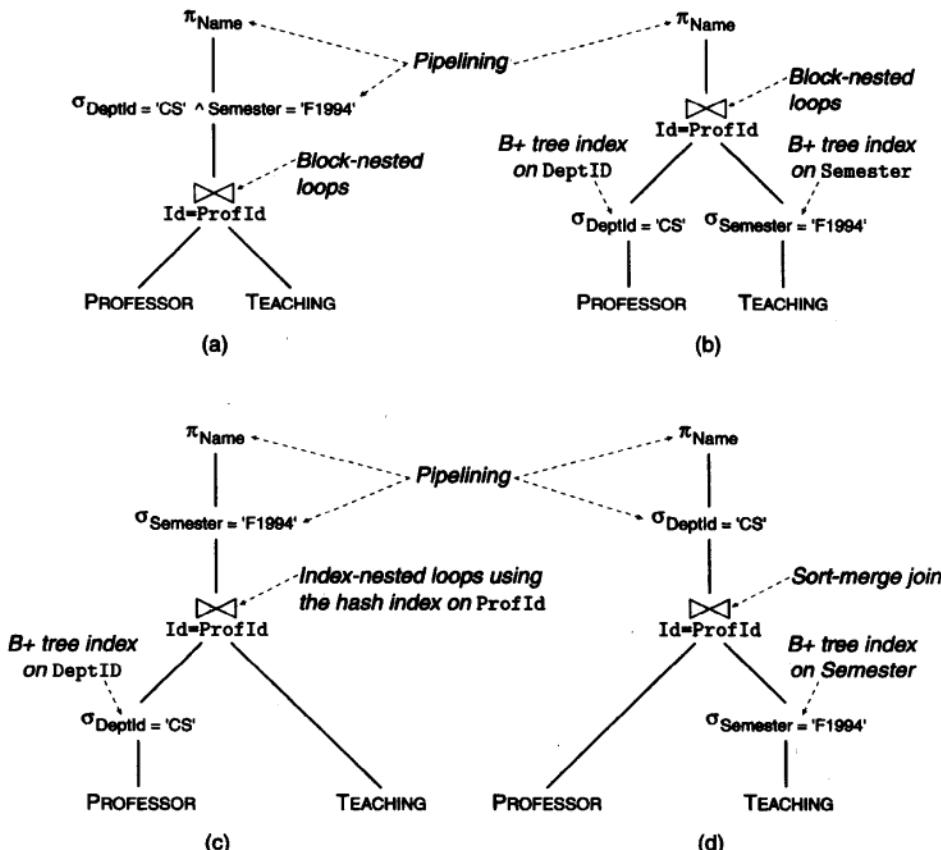


FIGURE 11.3 Query execution plans for relational expressions (11.2) through (11.5).

**TEACHING** is computed, the join will also be finished with no extra I/O. Thus, the total cost is  $4 + 4 + 250 = 258$ .

Note that if  $\sigma_{DeptId = 'CS'}(\text{PROFESSOR})$  were too big to fit in the buffer, then it would not be feasible to compute the join without writing  $\sigma_{Semester = 'F1994'}(\text{TEACHING})$  on disk. Indeed, scanning of  $\sigma_{DeptId = 'CS'}(\text{PROFESSOR})$  and the initial scan of  $\sigma_{Semester = 'F1994'}(\text{TEACHING})$  could still be done through pipelining but now  $\sigma_{Semester = 'F1994'}(\text{TEACHING})$  would have to be scanned multiple times, once for each chunk of  $\sigma_{DeptId = 'CS'}(\text{PROFESSOR})$ . To enable this,  $\sigma_{Semester = 'F1994'}(\text{TEACHING})$  would have to be written out on disk after the first scan.

**Case c: selection pushed to the PROFESSOR relation.** For the query tree in Figure 11.2(c), a query execution plan can be constructed as follows. First, compute

$\sigma_{\text{DeptId}=\text{'CS'}}$ (PROFESSOR) using the B<sup>+</sup> tree index on PROFESSOR.DeptId. As in case b, this prevents us from further using the hash index on PROFESSOR.Id in the subsequent join computation. Unlike case b, however, the relation TEACHING remains untouched, so we can still use index-nested loops (utilizing the index on TEACHING.ProfId) to compute the join. Other possibilities are block-nested loops and sort-merge join. Finally, we can pipe the output of the join to the selection operator  $\sigma_{\text{DeptId}=\text{'F1994'}}$  and apply the projection during the same scan.

The above query execution plan is depicted in Figure 11.3(c). We now estimate the cost of this plan.

- $\sigma_{\text{DeptId}=\text{'CS'}}$ (PROFESSOR). There are 50 departments and 1000 professors. Thus, the result of this selection will contain about 20 tuples, or 4 pages. Since the index on PROFESSOR.DeptId is clustered, retrieval of these tuples should take about 4 I/Os. Index search will take an additional 2 I/Os for a 2-level B<sup>+</sup> tree index. Because we intend to pipe the result of the selection into the join step that follows, there is no output cost.
- *Indexed-nested loops join.* We use the result of the previous selection and pipe it directly as input to the join. An important consideration here is that, because we chose index-nested loops utilizing the hash index on TEACHING.ProfId, the result of the selection does not need to be saved on disk even if this result is large. Once selection on PROFESSOR produces enough tuples to fill the buffers, we can immediately join these tuples with the matching TEACHING tuples, using the hash index, and output the joined rows. Then we can resume the selection and fill the buffers again.

As before, each PROFESSOR tuple matches about 10 TEACHING tuples, which are going to be stored in one bucket. So, to find the matches for 20 tuples, we have to search the index 20 times at the cost of 1.2 I/Os per search. Another 200 I/Os are needed to actually fetch the matching tuples from disk since the index is unclustered. All in all, this should take  $1.2 * 20 + 200 = 224$  I/Os.

- *Combined cost.* Since the result of the join is piped into the subsequent selection and projection, these latter operations do not cost anything in terms of I/O. Thus, the total cost is:  $4 + 2 + 224 = 230$  I/Os.

**Case d: selection pushed to the TEACHING relation.** This case is similar to case c, except that selection is now applied to TEACHING rather than to PROFESSOR. Since the indices on TEACHING are lost after applying the selection, we cannot use this relation in the inner loop of the index-nested loops join. However, we can use it in the outer loop of the index-nested loops join that utilizes the hash index on PROFESSOR.Id in the inner loop. This join can also be computed using block-nested loops and sort-merge. For this example, we select sort-merge. The subsequent application of selection and projection to the result can be done using pipelining, as in earlier examples. The resulting query plan is depicted in Figure 11.3(d).

- *Join: the sorting stage.* The first step is to sort PROFESSOR on Id and  $\sigma_{\text{Semester}=\text{'F1994'}}$ (TEACHING) on ProfId.

- To sort PROFESSOR, we must first scan it and create sorted runs. Since PROFESSOR fits in 200 blocks, there will be  $\lceil 200/50 \rceil = 4$  sorted runs. Thus, creation of the 4 sorted runs and storing them back on disk takes  $2 \times 200 = 400$  I/Os. These runs can then be merged in just one more pass, but we postpone this merge and combine it with the merging stage of the sort-merge join. (See below.)
- To sort  $\sigma_{\text{Semester}=\text{'F1994'}}(\text{TEACHING})$ , we must first compute this relation. Since TEACHING holds information for about 4 semesters, the size of the selection is about  $10,000/4 = 2500$  tuples. The index is clustered, so the tuples are stored consecutively in the file in 250 blocks. The cost of the selection is therefore about 252 I/O operations, which includes 2 I/O operations for searching the index.

The result of the selection is not written to disk. Instead, each time the 50-page buffer in main memory is filled, it is immediately sorted to create a run and then written to disk. In this way we create  $\lceil 250/50 \rceil = 5$  sorted runs. This takes 250 I/Os.

The 5 sorted runs of  $\sigma_{\text{Semester}=\text{'F1994'}}(\text{TEACHING})$  can be merged in one pass. However, instead of doing this separately, we combine this step with the merging step of the join (and the merging step of sorting PROFESSOR, which was postponed earlier).

- *Join: the merging stage.* Rather than merging the 4 sorted runs of PROFESSOR and the sorted runs of  $\sigma_{\text{Semester}=\text{'F1994'}}(\text{TEACHING})$  into two sorted relations, the runs are piped directly into the merge stage of the sort-merge join without writing the intermediate sorted results on disk. In this way, we combine the final merge steps in sorting these relations with the merge step of the join.

The combined merge uses 4 input buffers for each of the sorted runs of PROFESSOR, 5 input buffers for each sorted run of  $\sigma_{\text{Semester}=\text{'F1994'}}(\text{TEACHING})$ , and one output buffer for the result of the join. The tuple  $p$  with the lowest value of  $p.\text{Id}$  among the heads of the 4 PROFESSOR's runs is selected and matched against the tuple  $t$  with the lowest value of  $t.\text{ProfId}$  among the tuples in the head of the 5 runs corresponding to  $\sigma_{\text{Semester}=\text{'F1994'}}(\text{TEACHING})$ . If  $p.\text{Id}=t.\text{ProfId}$ ,  $t$  is removed from the corresponding run and the joined tuple is placed in the output buffer (we remove  $t$  and not  $p$  because the same PROFESSOR tuple can match several TEACHING tuples). If  $p.\text{Id}< t.\text{ProfId}$ ,  $p$  is discarded; otherwise,  $t$  is discarded. The process then repeats itself until all the input runs are exhausted.

The combined merge can be done at a cost of reading the sorted runs of the two relations: 200 I/Os for the runs of PROFESSOR and 250 I/Os for  $\sigma_{\text{Semester}=\text{'F1994'}}(\text{TEACHING})$ , respectively.

- *The rest.* The result of the join is then piped directly to the subsequent selection (on DeptId) and projection (on Name) operators. Since no intermediate results are written to disk, the I/O cost of these stages is zero.
- *Combined cost.* Summing up the costs of the individual operations, we get:  $400 + 252 + 250 + 200 + 250 = 1352$ .

**And the winner is . . .** Tallying up the results, we can see that the best plan (among those considered—only a small portion of all possible plans) is plan (c) from Figure 11.3. The interesting observation here is that this plan is better than plan (b), even though plan (b) joins smaller relations (because the selections are fully pushed). The reason for this apparent paradox is the loss of an index when selection is pushed down to the TEACHING relation. This illustrates once again that the heuristic rules of Section 11.2 are just that—heuristics. While they are likely to lead to better query execution plans, they must be evaluated within a more general cost model.

## 11.4 Estimating the Size of the Output

The examples in Section 11.3 illustrate the importance of accurate estimates of the output size of various relational expressions. The result of one expression serves as input to the next, and the input size has a direct effect on the cost of the computation. To give a better idea of how such estimates can be done, we present a simple technique based on the assumption that all values have an equal chance of occurring in a relation.

The system catalog can contain the following set of statistics for each relation name R:

- *Blocks(R)*. The number of blocks occupied by the instance of table R
- *Tuples(R)*. The number of tuples in the instance of R
- *Values(R.A)*. The number of distinct values of attribute A in the instance of R
- *MaxVal(R.A)*. The maximum value of attribute A in the instance of R
- *MinVal(R.A)*. The minimum value of attribute A in the instance of R.

Earlier we introduced the notion of attribute weight and used it to estimate sizes of selection and equi-join. We now define a more general notion, the *reduction factor*. Consider the following general query:

---

SELECT	<i>TargetList</i>
FROM	<i>R<sub>1</sub> V<sub>1</sub>, ..., R<sub>n</sub> V<sub>n</sub></i>
WHERE	<i>Condition</i>

---

The reduction factor of this query is the ratio

$$\frac{\text{Blocks}(\text{the result set})}{\text{Blocks}(R_1) \times \dots \times \text{Blocks}(R_n)}$$

At first, this definition seems cyclic: to find out the size of the result we need to know the reduction factor, but for this we need to know the size of the result set. However, the reduction factor can be *estimated* by induction on the query structure without knowing the size of the query result.

We assume that reduction factors associated with different parts of the query are independent of each other. Thus,

$$\text{reduction}(\text{Query}) = \text{reduction}(\text{TargetList}) \times \text{reduction}(\text{Condition})$$

where  $\text{reduction}(\text{TargetList})$  is the size reduction due to projection of rows on the attributes in the SELECT clause and  $\text{reduction}(\text{Condition})$  is the size reduction due to the elimination of rows that do not satisfy Condition.

We also assume that if  $\text{Condition} = \text{Condition}_1 \text{ AND } \text{Condition}_2$  then

$$\text{reduction}(\text{Condition}) = \text{reduction}(\text{Condition}_1) \times \text{reduction}(\text{Condition}_2)$$

and if  $\text{Condition} = \text{Condition}_1 \text{ OR } \text{Condition}_2$ , then

$$\text{reduction}(\text{Condition}) = \min(1, \text{reduction}(\text{Condition}_1) + \text{reduction}(\text{Condition}_2))$$

Thus, the size reduction due to a complex condition can be estimated in terms of the size reduction due to the components of that condition.

It remains to estimate the reduction factors due to projection in the SELECT clause and due to atomic conditions in the WHERE clause. We ignore nested sub-queries and aggregates in this discussion.

- $\text{reduction}(R_i.A = value) = \frac{1}{\text{Values}(R_i.A)}$ , where  $R_i$  is a relation name and  $A$  is an attribute in  $R_i$ . This estimate is based on the uniformity assumption—all values are equally probable.
- $\text{reduction}(R_i.A = R_j.B) = \frac{1}{\max(\text{Values}(R_i.A), \text{Values}(R_j.B))}$ , where  $R_i$  and  $R_j$  are relations and  $A$  and  $B$  are attributes. Using the uniformity assumption, we can decompose  $R_i$  (respectively,  $R_j$ ) into subsets with the property that all elements of a subset have the same value of  $R_i.A$  (respectively,  $R_j.B$ ). If we assume that there are  $N_{R_i}$  tuples in  $R_i$  and  $N_{R_j}$  tuples in  $R_j$  and that every element of  $R_i$  matches an element of  $R_j$ , then we can conclude that the number of tuples that satisfy the condition is  $\text{Values}(R_i.A) \times (N_{R_j}/\text{Values}(R_i.A)) \times (N_{R_i}/\text{Values}(R_j.B))$ . In general, the reduction factor is calculated assuming (unrealistically) that each value in the smaller range always matches a value in the larger range. Assuming that  $R_i.A$  is the smaller range, and dividing this expression by  $N_{R_i} \times N_{R_j}$ , yields the above reduction factor.
- $\text{reduction}(R_i.A > value) = \frac{\text{MaxVal}(R_i.A) - value}{\text{MaxVal}(R_i.A) - \text{MinVal}(R_i.A)}$ . The reduction factor for  $R_i.A < value$  is defined similarly. These estimates are also based on the assumption that all values are distributed uniformly.
- $\text{reduction}(\text{TargetList}) = \frac{\text{number-of-attributes}(\text{TargetList})}{\sum_i \text{number-of-attributes}(R_i)}$ . Here, for simplicity, we assume that all attributes contribute equally to the tuple size.

The weight of an attribute, which we used in Section 11.3, can now be estimated using the notion of reduction factor:

$$\text{weight}(R_i.A) = \text{Tuples}(R_i) \times \text{reduction}(R_i.A = value)$$

For instance, the reduction factor of the query `PROFESSOR.DeptId = value` is 1/50, since there are 50 departments. As the number of tuples in `PROFESSOR` is 1000, the weight of the attribute `DeptId` in `PROFESSOR` is 20.

## 11.5 Choosing a Plan

In Section 11.3 we looked at some query execution plans and showed how to estimate their cost. However, we did not discuss how to *produce* candidate plans. Unfortunately, the number of possible plans can be quite large, so we need an efficient way of choosing a relatively small, promising subset. We can then estimate the cost of each and choose the best. There are at least three major issues involved in this process:

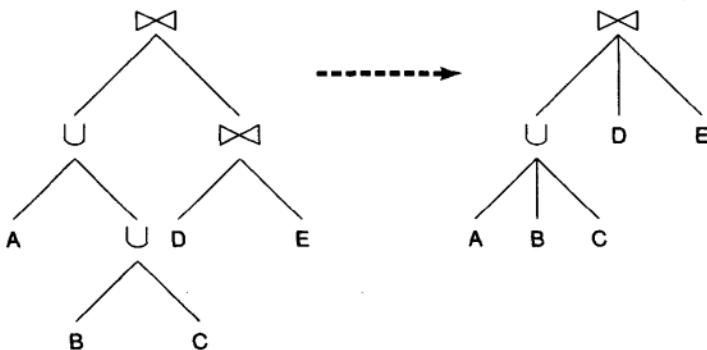
1. Choosing a logical plan
2. Reducing the search space
3. Choosing a heuristic search algorithm

We discuss each of these issues in turn.

**Choosing a logical plan.** We defined a query execution plan as a query tree with the relational implementation methods attached to each inner node. Thus, constructing such a plan involves two tasks: choosing a tree and choosing the implementation methods. Choosing the right tree is the more difficult job because of the number of trees involved, which, in turn, is caused by the fact that the binary associative and commutative operators, such as join, cross product, union, and the like, can be processed in so many different ways. We mentioned in Section 11.2 that the subtree of a query tree in which  $N$  relations are combined by such an operator can be formed in  $T(N) \times N!$  ways. We want to deal with this kind of exponential complexity separately, so we first focus on **logical query execution plans**, which avoid the problem by grouping consecutive binary operators of the same kind into one node, as shown in Figure 11.4.

The different logical query execution plans are created from the “master plan” (as in Figure 11.2(a) on page 411) by pushing selections and projections down and by combining selections and Cartesian products into joins. Only a few of all possible logical plans are retained for further consideration. Typically, the ones selected are fully pushed trees (because they are expected to produce the smallest intermediate results) plus all the “nearly” fully pushed trees. The reason for the latter should be clear from the discussion and examples in Section 11.3: Pushing selection or projection down to a leaf node of a query tree might eliminate the option of using an index in the join computation.

According to this heuristic, the query tree in Figure 11.2(a) will not be selected since nothing has been pushed. The remaining trees include the one in Figure 11.3(c), which has the least estimated cost and which is superior to the fully pushed query plan in Figure 11.3(b). In this example, all joins are binary, so the transformation shown in Figure 11.4 does not pertain.



**FIGURE 11.4** Transforming a query tree into a logical query execution plan.

**Reducing the search space.** Having selected candidate logical query execution plans, the query optimizer must decide how to evaluate the expressions that involve the commutative and associative operators. For instance, Figure 11.5 shows several alternative but equivalent ways of converting a commutative and associative node of a logical plan that combines multiple relations (a) into query trees (b), (c), and (d).

The space of all possible equivalent query (sub)trees that correspond to a node in a logical query plan is two-dimensional. First, we must choose the desired *shape* of the tree (by ignoring the labels on the nodes). For instance, the trees in Figure 11.5 have different shapes, with (d) being the simplest. Trees of such a shape are called **left-deep query trees**. A tree shape corresponds to a particular parenthesizing of a relational subexpression that involves an associative and commutative operator. Thus, the logical query execution plan in Figure 11.5(a) corresponds to the expression  $A \bowtie B \bowtie C \bowtie D$ , while the query trees (b), (c), and (d) correspond to the expressions  $(A \bowtie B) \bowtie (C \bowtie D)$ ,  $A \bowtie ((B \bowtie C) \bowtie D)$ , and  $((A \bowtie B) \bowtie C) \bowtie D$ , respectively. A left-deep query tree always corresponds to an algebraic expression of the form  $\dots ((E_{i_1} \bowtie E_{i_2}) \bowtie E_{i_3}) \bowtie \dots \bowtie E_{i_N}$ .

Query optimizers usually settle on one particular shape for the query tree: left-deep. This is because, even with a fixed tree shape, query optimizers have plenty of work to do. Indeed, given the left-deep tree of Figure 11.5(d), there are still  $4!$  possible ways to order the joins. For instance,  $((B \bowtie D) \bowtie C) \bowtie A$  is another ordering of the join of Figure 11.5(d) that leads to a different left-deep query execution plan. So, if computing cost estimates for  $4!$  query execution plans does not sound like a lot, think of what it would take to estimate the cost of  $10!$  or  $12!$  or  $16!$  plans. Incidentally, all commercial query optimizers give up at around 16 joins.

Apart from the general need to reduce the search space, there is another good reason to choose left-deep trees over the trees of the form Figure 11.5(b): pipelining. For instance, in Figure 11.5(d) we can first compute  $A \bowtie B$  and pipe the result to the next join with  $C$ . The result of this second join can also be piped up the tree

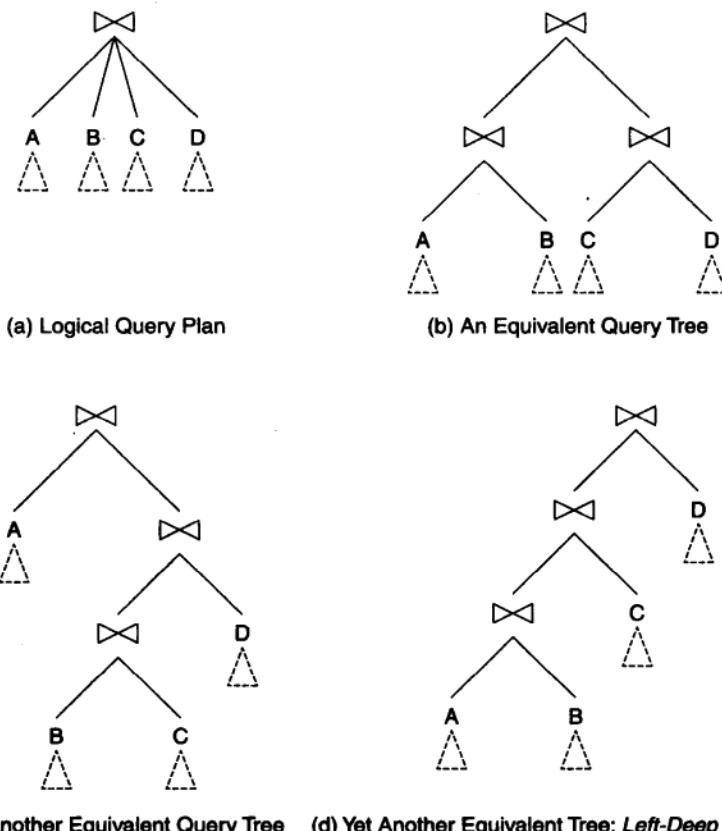


FIGURE 11.5 Logical plan and three equivalent query trees.

without materializing the intermediate relation on disk. The ability to do this is very important for large relations because the intermediate output from a join can be very large. For instance, if the size of the relations A, B, and C is 1000 pages, the intermediate relation can reach  $10^9$  pages just to shrink back to a few pages after joining with D. The overhead of storing such intermediate results on disk can be huge.

Note that the tree of Figure 11.5(b) does not lend itself to pipelining. For example, an attempt to pipe the result of  $A \bowtie B$  and the result of  $C \bowtie D$  to a module, M, that will join the two does not work. A row,  $t$ , of  $A \bowtie B$  must be compared to every row of  $C \bowtie D$  in order to compute the total join. That means that when  $t$  arrives, M must have received and stored all of  $C \bowtie D$ , which is exactly what we are trying to avoid. The alternative of storing one piece of  $C \bowtie D$  at a time is unsatisfactory since

**FIGURE 11.6** Heuristic search of the query execution plan space.

```

Input: A logical plan  $E_1 \bowtie \dots \bowtie E_N$ 
Output: A "good" left-deep plan  $(\dots ((E_{i_1} \bowtie E_{i_2}) \bowtie E_{i_3}) \bowtie \dots) \bowtie E_{i_N}$ 

1-Plans := all 1-relation plans
Best := all 1-relation plans with lowest cost
for ( $i := 1; i < N; i++$ ) do
    // Below,  $\bowtie^{\text{meth}}$  denotes join marked with an implementation method, meth*
    Plans := { best  $\bowtie^{\text{meth}}$  1-plan | best  $\in$  Best; 1-plan  $\in$  1-Plans, where
              1-plan is a plan for some  $E_j$  that has not
              been used so far in best }
    Best := { plan | plan  $\in$  Plans, where plan has the lowest cost }
end
return Best;

```

---

it implies that  $t$  has to be re-sent to  $M$  each time a new piece arrives. Observe that the tree (c) in that figure is *equivalent* to a left-deep tree but is not one of them. This means that even though the query optimizer limits the search to left-deep trees, the search space actually covers a much larger domain, which includes all the trees that are equivalent to the left-deep ones.

**A heuristic search algorithm.** The choice of left-deep trees has reduced the size of the search space from immense to huge. Next we must assign relations to the leaf nodes of the left-deep tree. There are  $N!$  such assignments, so estimating the cost of all is still a hopelessly intractable problem. Therefore, a heuristic search algorithm is needed to find a reasonable plan by looking at only a tiny portion of the overall search space. One such algorithm is based on *dynamic programming* and is used (with variations) in a number of commercial systems (e.g., DB2). We explain the main idea below; details are given in [Griffiths-Selinger et al. 1979]. A different heuristic search algorithm, described in [Wong and Youssefi 1976], is used in Ingres, another DBMS that was influential in the olden days.

A simplified version of the dynamic programming heuristic search algorithm is described in Figure 11.6. It builds a left-deep query tree by first evaluating the cost of all plans for computing each argument of an  $N$ -way join,  $E_1 \bowtie \dots \bowtie E_N$ , where each  $E_j$  is a 1-relation expression. These are referred to as *1-relation plans*. Note that each  $E_j$  can have several such plans (due to different possible access paths; e.g., one might use a scan and another an index), so the number of 1-relation plans can be  $N$  or larger. The *best* among all these plans (i.e., those with lowest cost) are expanded into 2-relation plans, then 3-relation plans, etc., as follows. To expand a best 1-relation plan,  $p$  (for definiteness, assume that  $p$  is a plan for  $E_{i_1}$ ) into a 2-relation plan,  $p$  is joined with every 1-relation plan, excluding the plans for  $E_{i_1}$  (because we already selected  $p$  as the plan for  $E_{i_1}$ ). We then evaluate the cost of all such plans

and retain the best 2-relation plans. Each best 2-relation plan,  $q$  (let us assume that it is a plan for  $E_{i_1} \bowtie E_{i_2}$ ), is expanded into a set of 3-relation plans by joining  $q$  with every 1-relation plan, except the plans for  $E_{i_1}$  and  $E_{i_2}$  (since the latter are already accounted for in  $q$ ). Again, only the lowest-cost plans are retained for the next stage. The process continues until a left-deep expression corresponding to the logical plan  $E_1 \bowtie \dots \bowtie E_N$  is fully constructed.

**Example 11.5.1 (Choosing the Best Plan).** We illustrate the overall process using our running example, query (11.6). First, the query processor generates a number of plausible logical plans—in our case, most likely the fully pushed tree of Figure 11.2(b) on page 411 plus the two partially pushed trees (c) and (d).

The shape of the trees depicted in Figure 11.2 are left-deep, but there are two query execution plans corresponding to each such tree: they differ in the order of relations in the join. Let us consider the query execution plans generated using the algorithm of Figure 11.6 starting with the logical plan of Figure 11.2(c).

The 1-relation plans for  $\sigma_{DeptId='CS'}(\text{PROFESSOR})$  can use the following access paths: a scan, the clustered index on  $\text{PROFESSOR.DeptId}$ , or a binary search (because  $\text{PROFESSOR}$  is sorted on  $\text{DeptId}$ ). The best plan uses the index, so it is retained. For the expression  $\text{TEACHING}$ , scan is all that can be done. We now have two 1-relation plans. In the next iteration, the algorithm expands the chosen 1-relation plans to 2-relation plans. This amounts to generating the two expressions  $\sigma_{DeptId='CS'}(\text{PROFESSOR}) \bowtie_{Id=ProfId} \text{TEACHING}$  and  $\text{TEACHING} \bowtie_{ProfId=Id} \sigma_{DeptId='CS'}(\text{PROFESSOR})$  and deciding on the evaluation strategy to use for the join in each. We estimated the different plans for the former expression in Section 11.3 and concluded that the index-nested loops join is the best. The second expression cannot be evaluated in the same way because the order of the arguments indicates that the relation  $\text{TEACHING}$  is scanned first. This expression can be evaluated using sort-merge or block-nested loops. Both methods are more expensive, so they are discarded.

Once the best plan for evaluating the join is selected, we can consider the result of the join as a 1-relation expression,  $E$ , and we now have to find a plan for  $\pi_{\text{Name}}(\sigma_{\text{Semester}='F1994'}(E))$ . Since the result of  $E$  is not sorted or indexed and since duplicate elimination is not requested, we choose a sequential scan access path to compute both selection and projection. Also, since  $E$  generates the result in main memory, we choose pipelining to avoid saving the intermediate result on disk. ■

The dynamic programming algorithm is likely to miss some good plans because it focuses on what is best at the current moment without trying to look ahead. One improvement here is to retain not only the best plans but also certain “interesting” plans. A plan might be considered interesting if its output relation is sorted or if it has an index, even if the cost of the plan is not minimal. This heuristic recognizes the fact that a sorted relation can significantly reduce the cost of subsequent operations, such as sort-merge join, duplicate elimination, and grouping. Likewise, an indexed relation can reduce the cost of subsequent joins.

## BIBLIOGRAPHIC NOTES

An extensive textbook treatment of query optimization can be found in [Garcia-Molina et al. 2000]. Heuristic search algorithms are described in [Griffiths-Selinger et al. 1979; Wong and Youssefi 1976].

For further reading on the latest query optimization techniques as well as additional references, see [Ioannidis 1996; Chaudhuri 1998].

## EXERCISES

- 11.1 Is there a *commutativity* transformation for the projection operator? Explain.
- 11.2 Write down the sequence of steps needed to transform  $\pi_A((R \bowtie_{B=C} S) \bowtie_{D=E} T)$  into  $\pi_A(\pi_E(T) \bowtie_{E=D} \pi_{ACD}(S)) \bowtie_{C=B} R$ . List the attributes that each of the schemas R, S, and T *must* have and the attributes that each (or some) of these schemas must *not* have in order for the above transformation to be correct.
- 11.3 Under what conditions can the expression  $\pi_A((R \bowtie_{cond_1} S) \bowtie_{cond_2} T)$  be transformed into  $\pi_A(\pi_B(R \bowtie_{cond_1} \pi_C(S)) \bowtie_{cond_2} \pi_D(T))$  using the heuristic rules given in Section 11.2?
- 11.4 Consider the join PROFESSOR  $\bowtie_{Id=ProfId}$  TEACHING used in the running example of Section 11.3. Let us change the statistics slightly and assume that the number of distinct values for TEACHING.ProfId is 10,000 (which translates into lower weight for this attribute).
  - a. What is the cardinality of the PROFESSOR relation?
  - b. Let there be an unclustered hash index on ProfId and assume that, as before, 5 PROFESSOR tuples fit in one page, 10 TEACHING tuples fit in one page, and the cardinality of TEACHING is 10,000. Estimate the cost of computing the above join using index-nested loops and block-nested loops with a 51-page buffer.
- 11.5 Consider the following query:

---

```
SELECT DISTINCT E.Ename
  FROM   EMPLOYEE E
 WHERE  E.Title = 'Programmer' AND E.Dept = 'Production'
```

---

Assume that

- 10% of employees are programmers
- 5% of employees are programmers who work for the production department
- There are 10 departments
- The EMPLOYEE relation has 1000 pages with 10 tuples per page
- There is a 51-page buffer that can be used to process the query

Find the best query execution plan for each of the following cases:

- a. The only index is on Title, and it is a clustered 2-level B<sup>+</sup> tree.
- b. The only index is on the attribute sequence Dept, Title, Ename; it is clustered and has two levels.

- c. The only index is on Dept, Ename, Title; it is a clustered 3-level B<sup>+</sup> tree.
- d. There is an unclustered hash index on Dept and a 2-level clustered tree index on Ename.

11.6 Consider the following schema, where the keys are underlined:

---

**EMPLOYEE(SSN, Name, Dept)**  
**PROJECT(SSN, PID, Name, Budget)**

---

The SSN attribute in PROJECT is the Id of the employee working on the project, and PID is the Id of the project. There can be several employees per project, but the functional dependency PID → Name, Budget holds (so the relation is not normalized). Consider the query

---

```
SELECT P.Budget, P.Name, E.Name
FROM EMPLOYEE E, PROJECT P
WHERE E.SSN = P.SSN AND
      P.Budget > 99 AND
      E.Name = 'John'
ORDER BY P.Budget
```

---

Assume the following statistical information:

- 10,000 tuples in EMPLOYEE relation
- 20,000 tuples in PROJECT relation
- 40 tuples per page in each relation
- 10-page buffer
- 1000 different values for E.Name
- The domain of Budget consists of integers in the range of 1 to 100
- Indices
  - EMPLOYEE relation
    - On Name: Unclustered, hash
    - On SSN: Clustered, 3-level B<sup>+</sup> tree
  - PROJECT relation
    - On SSN: Unclustered, hash
    - On Budget: Clustered, 2-level B<sup>+</sup> tree
- a. Draw the *fully pushed* query tree.
- b. Find the “best” execution plan and the second-best plan. What is the cost of each? Explain how you arrived at your costs.

11.7 Consider the following schema, where the keys are underlined (different keys are underlined differently):

---

**PROFESSOR(Id, Name, Department)**  
**COURSE(CrsCode, Department, CrsName)**  
**TEACHING(ProfId, CrsCode, Semester)**

---

Consider the following query:

---

```
SELECT C.CrsName, P.Name
FROM PROFESSOR P, TEACHING T, COURSE C
WHERE T.Semester='F1995' AND P.Department='CS'
AND P.Id = T.ProfId AND T.CrsCode=C.CrsCode
```

---

Assume the following statistical information:

- 1000 tuples with 10 tuples per page in PROFESSOR relation
  - 20,000 tuples with 10 tuples per page in TEACHING relation
  - 2000 tuples, 5 tuples per page, in COURSE
  - 5-page buffer
  - 50 different values for Department
  - 200 different values for Semester
  - Indices
    - PROFESSOR relation
      - On Department: Clustered, 2-level B<sup>+</sup> tree
      - On Id: Unclustered, hash
    - COURSE relation
      - On CrsCode: Sorted (no index)
      - On CrsName: Hash, unclustered
    - TEACHING relation
      - On ProfId: Clustered, 2-level B<sup>+</sup>-tree
      - On Semester, CrsCode: Unclustered, 2-level B<sup>+</sup> tree
- a. First, show the *unoptimized* relational algebra expression that corresponds to the above SQL query. Then *draw* the corresponding *fully pushed* query tree.
  - b. Find the best execution plan and its cost. Explain how you arrived at your costs.

**11.8** Consider the following relations that represent part of a real estate database:

---

```
AGENT(Id, AgentName)
HOUSE(Address, OwnerId, AgentId)
AMENITY(Address, Feature)
```

---

The AGENT relation keeps information on real estate agents, the HOUSE relation has information on who is selling the house and the agent involved, and the AMENITY relation provides information on the features of each house. Each relation has its keys underlined. Consider the following query:

---

```
SELECT H.OwnerId, A.AgentName
FROM HOUSE H, AGENT A, AMENITY Y
WHERE H.Address=Y.Address AND A.Id = H.AgentId
AND Y.Feature = '5BR' AND H.AgentId = '007'
```

---

Assume that the buffer space available for this query has 5 pages and that the following statistics and indices are available:

- **AMENITY**
  - 10,000 records on 1000 houses, 5 records per page
  - Clustered 2-level B<sup>+</sup> tree index on **Address**
  - Unclustered hash index on **Feature**, 50 features
- **AGENT**
  - 200 agents with 10 tuples per page
  - Unclustered hash index on **Id**
- **HOUSE**
  - 1000 houses with 4 records per page
  - Unclustered hash index on **AgentId**
  - Clustered 2-level B<sup>+</sup> tree index on **Address**

Answer the following questions (and explain how you arrived at your solutions).

- a. Draw a fully pushed query tree corresponding to the above query.
  - b. Find the best query plan to evaluate the above query and estimate its cost.
  - c. Find the next-best plan and estimate its cost.
- 11.9 None of the query execution plans in Figure 11.3 for queries (11.2)–(11.5) does duplicate elimination. To account for this, let us add one more relational operator,  $\delta$ , which denotes the operation of duplicate elimination. Modify the plans in Figure 11.3 by adding  $\delta$  in appropriate places so as to minimize the cost of the computation. Estimate the cost of each new plan.
- 11.10 Build a database for the scenario in Exercise 11.5 using the DBMS of your choice. Use the EXPLAIN PLAN statement (or an equivalent provided by your DBMS) to compare the best plan that you found manually with the plan actually generated by the DBMS.
- 11.11 Follow Exercise 11.10, but use the scenario in Exercise 11.6.
- 11.12 Follow Exercise 11.10, but use the scenario in Exercise 11.7.
- 11.13 Consider the query execution plans in Figure 11.3. Show how each of these plans can be *enhanced* by pushing the projection operator past the join *without altering the strategies used for computing the join*.
- 11.14 Using the result of Exercise 11.13, show which of the enhanced plans can be further enhanced by adding the duplicate elimination operator  $\delta$  introduced in Exercise 11.9.

# 12

## Database Tuning

Tuning is the process of modifying an application and adjusting the parameters of the underlying DBMS to improve performance. Performance is measured in terms of the response time seen by a user (the time it takes to perform a task—for example, to execute an SQL statement) and throughput (the amount of work completed in a unit of time). It is important to realize that tuning does not affect the semantics of the system: the tuned and the original systems return the same information to the user and are left in the same final state when subjected to the same sequence of requests.

The first step in tuning a system is to determine where the bottlenecks are. If the system spends only 2% of its time executing a particular (hardware or software) module, then no matter how inefficient it is, revising or replacing it can improve performance by at most 2%.

An application and DBMS, taken together, form an exceedingly complicated system, and many different aspects of it are subject to tuning. The SQL code and schema are at the highest level. Tuning at this level is concerned with such issues as how queries should be expressed and what indices should be created. These are application-related questions and, since this is an “application-oriented” text, it is the level to which we pay the most attention. You might have wondered why the material in Chapters 10 and 11 was included in an application-oriented text since those chapters describe algorithms internal to the DBMS. The reason lies in this chapter. While Chapters 10 and 11 described a number of different techniques that a DBMS can use to process the SQL statements that your application submits, this chapter discusses methods that you can use to encourage the DBMS to use the technique that performs the best for the particular application you are implementing.

The DBMS occupies the next level. Examples of performance issues at this level are the physical placement of data on secondary storage and how the DBMS manages its buffers. Decisions in this area are largely under the control of the database administrator, and hence the application programmer can influence them indirectly. As a result we spend some time in this chapter discussing tuning at the DBMS level.

The lowest tuning level is the hardware level. In order to perform well the system must be supported by a sufficient amount of main memory, a sufficient number of

CPUs and secondary storage devices, and adequate communication facilities. The specification of these resources is generally beyond the control of the application programmer, and we do not discuss these issues.

## 12.1 Disk Caches

In Chapter 9 we discussed the huge difference between the speed of the CPU and the time to transfer a page between the CPU and mass store. In recognition of this, the cost of a query plan is measured as the estimated number of page transfers it incurs, and the job of the query optimizer is to find the plan that minimizes this number. While that plan is generally a good one, its cost is often still significant, and other measures are necessary to make query processing efficient. One of the most significant of these is the cache. A *cache* is a main memory buffer in the DBMS in which recently accessed database pages are stored. When a transaction accesses a database item, the DBMS brings the database page(s) on disk that contain that item into the cache and then copies the value of the item from the cache into the application's buffer. The page is generally retained in the cache under the assumption that there is a high probability that the application will either update the item or read another item in the same page at a later time. Or another application might concurrently reference an item in the page. In either case, a disk access will have been avoided since the page will be directly accessible in the cache. For example, an index page has a high probability of being accessed frequently.

Although it is natural to think of the database item as a page of a table or an index, it can also be the execution plan for an SQL statement or a stored procedure. In fact, some DBMSs maintain a separate **procedure cache** for this purpose. Although I/O cost is the major limitation on the performance of an application, the CPU cost of building an appropriate execution plan is also substantial. Hence, once an execution plan has been determined, it is saved since it might be possible to reuse it. Prior to preparing a new execution plan, existing plans are scanned to see if any are usable.

If a database item is to be updated, the database page containing the item must first be brought into the cache (if it is not already there), and it is the cache copy of the page that is modified (not the original copy in the database).

Eventually the cache becomes full, and any new page fetched from the database must overwrite a page,  $p$ , in the cache. If  $p$  has not been updated since arriving in the cache, its contents are identical to the corresponding page in the database, and hence it can simply be overwritten by the new page. However, if  $p$  has been updated since arriving in the cache, it must be written back to the database before the space it occupies in the cache can be freed. In order to distinguish between these two cases, the DBMS marks pages that have been updated as **dirty** and those that have not as **clean**.

Decisions concerning which pages should be kept in the cache and which can be overwritten when a new page is to be fetched are made by a **page replacement algorithm** whose goal is to maximize the number of database accesses that can be

satisfied by pages in the cache. A least recently used (LRU) algorithm, for example, selects the least recently used page in the cache as the one to be replaced. It concludes that since no application has accessed the page recently it is no longer useful. Hence it tends to keep actively used pages in the cache.

A more sophisticated algorithm takes into account the circumstances under which a page was brought into the cache. For example, if the page was brought in as part of a table scan (which is typical, for example, when sorts are performed), once the rows in the page have been accessed it is not likely that the application will reference the page again. In this case a most recently used algorithm (MRU) is preferable. Hence a page replacement policy might use a combination of an LRU and an MRU algorithm depending on what information is contained in the page (index or data) and in what context the page is referenced.

If a transaction's access request can be satisfied from the cache, a **hit** is said to have taken place; if it cannot be satisfied, then a **miss** has occurred. To obtain a high throughput, many designers consider it mandatory to obtain a hit rate of over 90% (90% of the accesses can be satisfied from the cache). To achieve such a hit rate, the cache size must often be a significant percentage of the size of the database. Cache sizes in the megabyte range are normal. In some large applications, the cache size is measured in tens of gigabytes.

### 12.1.1 Tuning the Cache

Now that you have an understanding of how the cache works, the question is, "What can the application programmer or the database administrator do to optimize the way the DBMS uses the cache to improve the performance of her application?" DBMSs generally offer several mechanisms that can be invoked for this purpose.

- Some DBMSs allow pieces to be carved out of the (default) cache to be managed as separate caches. The programmer can then bind a particular item (e.g., a table or an index) to a specific cache and in so doing cause all pages of that item to be buffered in that cache. For example, if tables  $T_1$  and  $T_2$  are bound to different caches, a page of  $T_1$  can never overwrite a page of  $T_2$ . This approach might be useful if  $T_2$  was not used very often but fast response time was required of the application that referenced it.
- Some DBMSs allow a particular cache to be subdivided into several distinct pools of buffers of different sizes. For example, while by default all buffers in a cache might have 2K bytes, it might be possible to reallocate the cache storage area so that several buffer pools are created with sizes 2K, 4K, 8K, etc. If a table is bound to such a cache, the query optimizer then has the option of choosing the I/O size that best suits a query plan that accesses that table. For example, the page size on secondary storage might be 2K bytes, and the DBMS might allocate disk space to a table in contiguous blocks of eight pages. It then follows that the time to retrieve an eight-page block is not much larger than the time to retrieve a single page since the seek time is the same in both cases. If the query plan calls

for a table scan, and the table is bound to a cache that has a 16K pool of buffers, the query optimizer can save time by retrieving eight pages with a single I/O operation.

Some query optimizers take this idea one step further by prefetching pages. Ordinarily, during the scan of a table or index, the next page is requested when the page fetched by the previous I/O operation has been scanned. The scan must then wait until the I/O operation for the next page completes. It is possible to improve on this in situations, such as scans, in which the optimizer can anticipate future requests. In such cases the optimizer can initiate the I/O operation for a page that has not yet been requested. Then, if the time to process a page is long enough, the next page will already be in the cache when it is requested.

By using both prefetching and a large I/O size, the time to do a table scan can be greatly reduced. This possible reduction has an impact on the query optimizer's choice between an access path that involves an index and an access path that uses a table scan for a particular query. It also raises the question of whether the application programmer should create an index for that query.

- Some DBMSs provide commands that allow the page replacement policy for a cache to be specified. This is particularly useful when multiple caches are used. A policy appropriate to the items bound to the cache can then be chosen.

In addition to configuring a data cache to best suit the application, the programmer can design the application to make the best use possible of a procedure cache. For example, performance can be improved by not using explicit constants in SQL statements. The execution plan for the statement

---

```
SELECT P.Name
FROM PROFESSOR P
WHERE P.DeptId = 'EE'
```

---

is essentially the same as the execution plan for the statement with the constant EE replaced by CS. But since the two statements are different, the DBMS might miss this fact when it scans the procedure cache looking for an execution plan, and as a result the DBMS might create a separate execution plan for each statement. It is possible to eliminate this overhead by instead using the statement

---

```
SELECT P.Name
FROM PROFESSOR P
WHERE P.DeptId = :deptid
```

---

where deptid is a host variable (see Chapter 8), and successively assigning EE and CS to that variable. Since the same statement is now executed twice, the execution plan created when the statement is first executed will be reused when it is executed for the second time.

## 12.2 Tuning the Schema

The schema you design for your database is at the heart of the application. If the schema is well designed, it is possible to write SQL statements that perform efficiently. Your strategy in tuning at the application level is to first design a normalized database as described in Chapter 6 and then estimate the sizes of the tables, the distribution of column values, and the nature and frequency of the queries and updates that will be addressed to the database. Adjustments to the normalized schema to facilitate the most frequent operations follow from these estimates. Adding indices is the most important of these adjustments, and we discuss it first. Another technique is denormalization, which involves adding redundancy so that items of information that are generally associated with one another through frequently executed queries can be found in one place. Finally, we discuss partitioning, which is a rather specialized technique for dealing with very large tables.

### 12.2.1 Indices

In Chapter 11 we showed that different query plans for a particular query might have wildly different costs and that in many cases the differences were a function of the indices used in the plan. For better or worse, the choice of plan is made by the optimizer based on the indices available to it at the time the query is prepared. It is the role of the application programmer to "encourage" a good choice by making sure that appropriate indices have been created. In this section our goal is to expose the reasoning a programmer might use in deciding what indices to create.

Indices might seem like the ultimate database tuning device. However, free computational lunches are rare. Each index carries an associated storage overhead. More importantly, extra indices might significantly increase the processing time of statements that *modify* the database since every index must be updated whenever the table it references is changed. Thus, you should think twice before creating an index on a table where rows are frequently inserted or deleted. Similarly, you should think twice before creating an index with a search key involving a frequently updated column. Will the performance gain realized in processing queries be sufficient to compensate for the added cost of processing statements that modify the table? To illustrate some of the considerations involved in the tuning process, consider the following examples (which make use of the schema shown in Figure 3.4 on page 38).

1. Consider the query

---

```
SELECT P.DeptId
  FROM PROFESSOR P
 WHERE P.Name = :name
```

---

Since the primary key of PROFESSOR is Id, we can expect that the DBMS has created a clustered index on that attribute. That index is no help for this query because we need a quick way to find all professors with a particular name. One possibility is to explicitly create an unclustered index on Name. Assuming that

only a few professors have the same name, this index should speed things up. But suppose this is not the case: many professors have the same name. Then a better solution is to make the index on `Name` clustered and the index on `Id` unclustered. As a result, rows with the same name will be grouped together and can be retrieved in a single (or a few) I/O operations. The index could be a  $B^+$  tree or a hash (since the condition on `Name` involves equality).

The lesson here is that since a table can have only one clustered index, it is pointless to waste it on an attribute that cannot take advantage of clustering. DBMSs generally create a clustered index on the primary key, but you should not be intimidated by this. An unclustered index on the primary-key attribute is sufficient to guarantee the key's uniqueness, and since at most one row can have a particular key value, clustering cannot be justified as a means of grouping rows with the same value of the attribute. So, if we are unlikely to want to order rows based on the primary key (as is the case with `PROFESSOR`), there is no reason to use a clustered index for this purpose.

Keep in mind that replacing one clustered index with another is a time-consuming operation since it implies a complete reorganization of the storage structure. You certainly do not want to create a new clustered index each time you execute a query. You should analyze your application in advance, considering the kinds of queries you expect and their frequency, create the clustered index that will do the most good, and stick with it until performance considerations indicate that the system needs a tune-up.

2. Consider the query

---

```
SELECT T.Name, T.CrsCode
  FROM TRANSCRIPT T
 WHERE T.Grade = :grade
```

---

One is tempted to cluster the rows around `Grade` since we want to retrieve all rows with the same grade, but suppose that our first priority is to speed the response to a different query, a request for a class roster, and for that purpose we use a clustered index on the primary key (`CrsCode`, `Semester`, `StudId`). We could create an unclustered index on `Grade`, but using such an index might not be a good idea. In most cases the number of rows with a particular grade is a large fraction of the total number of rows (since the domain of `Grade` is small). In those cases we can expect that a large fraction of the table's pages will be fetched, one by one, in random order, through the unclustered index.<sup>1</sup> Unfortunately, the optimizer does not know what grade will be supplied at run time, and even if it did, it would not know which ones produced small result sets (we will correct this inadequacy shortly). Hence a table scan might be a better solution.

<sup>1</sup> If 10% of the rows are randomly fetched and each page contained 20 rows, then the probability that a particular page contains no rows in the result set is  $(.9)^{20}$ , which is approximately .12.

A number of lessons can be drawn from this example. First, an unclustered index is appropriate if only a few rows of a table are to be retrieved, and a full table scan is appropriate if a large fraction of the rows are to be retrieved. Determining a reasonable break-even point is not easy. One vendor states that a table scan is appropriate if more than 20% of the rows of the table are to be accessed. A more cautious approach would be to simulate the workload if more than a few rows are to be accessed to determine if building an index is a good idea. Second, do not create an index on a column with a small domain if attribute values tend to be evenly distributed over the domain. The query optimizer is *unlikely* to choose such an index since it will recognize that the selectivity of the access path through this index is large for any value of search key. Finally, do not create indices indiscriminately: they are costly to maintain, and, with the techniques described in Section 12.1, table scans can be quite fast.

3. Suppose that the most frequent access path to TRANSCRIPT selects rows based on a condition involving both StudId and CrsCode. A less frequently used path selects rows based on a condition on Semester. If we build one index on  $\langle \text{StudId}, \text{CrsCode} \rangle$  (actually an index on the primary key  $\langle \text{StudId}, \text{CrsCode}, \text{Semester} \rangle$  would work fine) and another on Semester, which should be clustered? At first glance, it might seem that the index on  $\langle \text{StudId}, \text{CrsCode} \rangle$  should be clustered because it is the main access path. However, even though  $\langle \text{StudId}, \text{CrsCode} \rangle$  is *not* a candidate key, the number of TRANSCRIPT rows that agree on both of these attributes will be one in almost all cases—only when a student retakes a course can this number be larger than one. Therefore, clustering around  $\langle \text{StudId}, \text{CrsCode} \rangle$  will not yield significant benefits. Also, it is not likely that range queries will be asked against this pair of attributes, so the overhead of a  $B^+$  tree index does not seem justified—a hash index is probably the best solution here. On the other hand, a clustered  $B^+$  tree index on Semester can greatly improve the efficiency of selections and joins on that attribute and makes an excellent choice for a secondary access path.

The lesson here is that clustering is useful to group together rows that might be output in a result set. These rows might be grouped because they all agree on the value of an attribute(s) or because they fall within a range of values of that attribute(s). In either case, when a choice has to be made as to what attribute to cluster on, you should make the choice based on the size of the result sets you expect in your application.

4. Assume the PROFESSOR table has the additional attribute Salary, and suppose we want to optimize the performance of the range query:

---

```
SELECT P.Name
FROM PROFESSOR P
WHERE P.Salary BETWEEN :lower AND :upper
```

---

The analysis here is similar to that of example 1 with the exception that we now want a clustered index on Salary and it must be a  $B^+$  tree.

5. If two different queries would benefit from two different clustered indices on the same table, we have a problem since only one clustered index is possible. One solution is to make it possible for the optimizer to use an **index-only strategy**. For example, suppose that TEACHING already has a clustered B<sup>+</sup> tree index on Semester, but another important query would benefit from a clustered index on ProfId in order to quickly access the course codes associated with a given professor. We can sidestep the problem by creating an *unclustered* B<sup>+</sup> tree index with search key (ProfId, CrsCode). Then all the information required by the query is contained in the index (and the index is often referred to as a **covering index**), and TEACHING does not have to be accessed at all! We simply search down the index using ProfId to the leaf level. Since the values of CrsCode at that level are clustered around ProfId, we can scan forward from that point at the leaf level of the index to get the required result set using only the index entries. This approach produces the same effect as that of a clustered index with search key (ProfId, CrsCode) on TEACHING (in fact, it is more efficient because the index is smaller and hence scanning a section of the leaf level requires fewer I/O operations than scanning a section of TEACHING).

Index-only query processing comes in two varieties. In this example we searched the index using ProfId to quickly locate the associated course codes. Suppose, however, another query required that we find the ProfIds of all professors who had taught a particular course. Unfortunately, although all the information we need is in the index, it cannot be searched because CrsCode is not the first attribute of the search key. But all is not lost. Another way to produce the desired result set is to scan the entire leaf level of the index. This is not as efficient as a search, but it might be better than having to scan the entire data file (the index is smaller!) or create and use an unclustered index on CrsCode.

6. The ability to nest queries is one of the most powerful features of SQL. Unfortunately, however, nested queries are very difficult to optimize. Consider the query

---

```
SELECT P.Name, C.CrsName
  FROM PROFESSOR P, COURSE C
 WHERE P.Department = 'CS' AND
       C.CrsCode IN
          (SELECT T.CrsCode
            FROM TEACHING T
           WHERE T.Semester = 'S2003' AND T.ProfId = P.Id)
```

---

that returns a set of rows in which the value of the first attribute is the name of a CS professor who has taught a course in the spring of 2003 and the value of the second is the name of one such course.

Typically, a query optimizer splits this query into two separate parts. The inner query is considered as an independently optimized unit. The outer query is also optimized independently (with the result set of the inner SELECT statement viewed as a database relation). In this case, the subquery is correlated, so it is crucial that it be executed efficiently since it will be executed many times. For example, a clustered index on TEACHING with search key (ProfId, Semester) would permit quick retrieval of all courses taught by a particular professor in a semester (and hopefully this is a small set). If possible (as in this example), the search key should involve all the attributes of the WHERE clause to avoid retrieving rows unnecessarily.

However, there is another point to note here. Because the two queries are optimized separately, certain alternatives might not be considered by the optimizer. For instance, the use of a clustered index on TEACHING with search key (ProfId, CrsCode) will not be considered since the correlated nested query simply produces a set of course codes for each value of P.Id that is supplied. On the other hand, it is easy to see that the above query is equivalent to

---

```
SELECT C.CrsName, P.Name
FROM PROFESSOR P, TEACHING T, COURSE C
WHERE T.Semester='S2003' AND P.Department='CS'
AND P.Id = T.ProfId AND T.CrsCode=C.CrsCode
```

---

and the use of that index *would be* considered in optimizing this query.

It should be remarked that some query optimizers do, in fact, try to eliminate nested subqueries and take other steps to reduce the cost of processing them. However, it is still a good idea to avoid query nesting whenever possible.

#### 7. Consider the query

---

```
SELECT T.Semester, COUNT(*)
FROM TRANSCRIPT T
WHERE T.Grade <= :grade
GROUP BY T.Semester
```

---

Our first inclination is to create a clustered B<sup>+</sup> tree on Grade since a range is indicated. Our intention is to influence the optimizer to first retrieve all rows satisfying the condition, sort them on Semester (which brings all the members of a group together), and then count the size of each group. But this is not necessarily a good idea. The condition is not selective, so we will have to sort a large intermediate table.

Suppose instead we reverse the order of operations: we do the sort before the selection. In fact, if we choose a clustered index on Semester, the table is sorted before the query is executed. Since the grouping is already done, all we

have to do is scan the table and count all the qualifying rows in each group—clearly a better plan when the condition is not selective. Note that the index can be either a B<sup>+</sup> tree or a hash. In both cases the rows in a group will be together.

The lesson here is that an index is not simply an access path to data; it is a way of storing the data. In this example, the query plan does not actually use the index to find a particular row but simply takes advantage of the way the rows are stored.

8. Consider the query

---

```
SELECT S.Name
FROM STUDENT S, TRANSCRIPT T
WHERE S.Id = T.StudId AND T.CrsCode = 'CS305'
```

---

If appropriate indices are not present, the optimizer might choose a block-nested loops join or a sort-merge join as the basis of a query plan. These choices are likely to be inefficient, since the size of the result set that we expect is considerably smaller than the size of the tables involved. As a general rule of thumb, you should investigate the possibility of an index-nested loops join when you expect a small result set, reserving other methods for large result sets.

So how can we encourage the optimizer to consider an index-nested approach? If we create a clustered index on TRANSCRIPT with search key CrsCode, the optimizer has a way of quickly finding, as part of the outer loop of the join, all students who have taken CS305. We can easily ensure that such an index exists since CrsCode is an attribute in the primary key of the table: all we have to do is make sure that it is declared as the first attribute of the key. The DBMS will generally oblige by creating a B<sup>+</sup> tree on the primary key.

For the inner loop of the join we need an index on STUDENT with search key Id. This is no problem at all since Id is the primary key. The DBMS will create an index, and we do not care whether it is clustered or unclustered, B<sup>+</sup> tree or hash, since Id is unique.

9. Consider the query

---

```
SELECT Te.ProfId, Tr.StudId
FROM TEACHING Te, TRANSCRIPT Tr
WHERE Te.Semester = Tr.Semester AND Te.CrsCode = Tr.CrsCode
```

---

We expect the size of the result set to be much larger than the size of either table. Hence, a sort-merge algorithm is likely to be efficient in performing the join. We can make such an algorithm attractive to the optimizer by using clustered B<sup>+</sup> indices on the tables involved. For example, if such an index (with search key (Semester, CrsCode)) is created on TRANSCRIPT, the relation will already be sorted on the join attributes and a significant part of the sorting step of the algorithm comes for free. Since these two attributes are a part of the primary key of the table, the DBMS has already created such an index—all we need to do

is make sure that the ordering of primary-key attributes is (`Semester`, `CrsCode`, `StudId`).

10. Consider a database with two tables: `PROJECTPART(ProjId, PartId)`, which relates a project to each part that it uses, and `PARTSUPPLIER(PartId, SupplId)`, which relates a part to each supplier that sells that part. The query

---

```
SELECT P.ProjId, S.SupplId
FROM PROJECTPART P, PARTSUPPLIER S
WHERE P.PartId = S.PartId
```

---

produces a (`ProjId`, `SupplId`) pair for each project that uses a part that the supplier sells. An index-nested loops join could scan `PROJECTPART` and use an index with search key `PartId` on `PARTSUPPLIER` to find the rows of that table that match each scanned row. Encouraging the use of such an algorithm, however, is probably a bad idea since many rows of `PARTSUPPLIER` join with each row of `PROJECTPART`. Reversing the tables so that `PARTSUPPLIER` is scanned produces the same result. Hence a sort-merge or hash join might be less expensive. The lesson here is that it is not a good idea to create an index unless you are sure it is going to be of use. In this case it might lead to the wrong query plan and result in added overhead when the indexed table is updated.

**Miscellaneous considerations.** A foreign-key constraint can be essential in supporting the integrity of your database but introduces a hidden cost since it must be checked when certain modifications are made to the tables that it relates. Suppose such a constraint is declared on attribute `A1` of table `T1` referring to attribute `A2` of table `T2`. When a row,  $t_1$ , is inserted in `T1`, the DBMS must ensure that there is a row in `T2` in which the value of `A2` matches the value of `A1` in  $t_1$ . Fortunately, this is not a problem since `A2` must be a key of `T2` and hence there is an index with search key `A2` that can be used to make the check quickly. Unfortunately, this approach does not work in reverse. If a row,  $t_2$ , of `T2` is deleted, the DBMS must check that there does not exist a row of `T1` that refers to it. Since `A1` is not a key of `T1`, `T1` might not have an index with search key `A1`, and if not, a table scan will be required to check the foreign-key constraint. If `T1` is large and rows of `T2` are deleted or updated frequently, this table scan can be a significant source of overhead. In that case, an index on `T1` with search key `A1` should be created.

A common query is one that counts the rows in a table using `COUNT`. Such a query can result in a table scan if a proper index is not available. The table scan can be replaced by an index scan if the index is over a column that has a `NOT NULL` constraint because a row in which that attribute is null would not be indexed. The I/O cost of an index scan can be substantially less since the leaf level of the index can be packed into fewer pages than the table. Note that even if the DBMS has created statistics describing the table, the values will generally not be current and so cannot be used.

### 12.2.2 Denormalization

In Chapter 6, we learned a great deal about schema decomposition. That discussion was motivated by concerns that redundancy leads to consistency-maintenance problems in the presence of frequent database updates. What if most of the transactions are read-only queries? Schema decomposition seems to make query answering harder because associations between columns that existed in one relation before the decomposition might be broken into separate relations afterward.

For instance, finding the hobbies of the person with a particular SSN is more efficient using the monolithic relation of Figure 4.13 than the pair of relations of Figure 6.1 because the latter requires a join. This is an example of the classic time/space trade-off: the redundancy present in the monolithic relation improves query performance and argues against decomposing the relation. Such a trade-off has to be evaluated in a particular application if the performance of a frequently executed query is found wanting.

**Denormalization** refers to situations in which an attempt is made to improve performance of read-only queries by adding redundant information to a table. It reverses the normalization process and results in a violation of normal form conditions.

Denormalization often takes the form of adding a redundant column. For example, in order to print a class roster that lists student names, a join is required between the tables STUDENT and TRANSCRIPT. The join can be avoided by adding a Name column to TRANSCRIPT. In contrast to the previous example, STUDENT contains other information (e.g., Address), so denormalization does not eliminate the need to retain STUDENT.

As another example, a join involving the tables STUDENT and TRANSCRIPT is needed to produce a result set that associates a student's name with her cumulative grade point average. If the query is performed frequently, we might improve performance by adding a GPA column to the STUDENT table. Although prior to the modification the GPA was not stored in the database, redundancy has been added since the GPA can be computed from TRANSCRIPT. This is a particularly attractive example of denormalization because the additional storage requirements are nominal.

But do not get carried away with denormalization. In addition to the extra storage required, a price has to be paid to maintain consistency. In this case, every time a grade is changed or a new row added to TRANSCRIPT, GPA has to be updated. This might be done by the transaction doing the modification, adding to its complication and degrading its performance. A better alternative is to add a trigger that updates STUDENT when the modification takes place. Although the performance penalty is not avoided, complication is reduced and the possibility that transactions do not properly maintain consistency is avoided.

There is no general rule on when to denormalize. Here is an incomplete list of conflicting guidelines that need to be evaluated against each particular mix of transactions:

1. Normalization can lower the demand for storage space since it usually eliminates redundant data and null values. Tables and rows are smaller, reducing the

amount of I/O that must be performed and allowing more rows to fit into the cache.

2. Denormalization increases storage requirements since redundant data is added. When the degree of redundancy is low, however, normalization can also increase storage requirements. For instance, in the PERSON relation of (6.14) on page 228, suppose that most people have just one phone number and one child. In this case, schema decomposition actually increases storage requirements (since SSN must be repeated in each table) without bringing tangible benefits. The same applies to the decomposition of HASACCOUNT in Figure 6.7, which can increase the overhead for update transactions. The reason is that verification of the FD

---

ClientId OfficeId → AccountNumber

---

after an update requires a join because the attributes ClientId and OfficeId belong to different relations in the decomposition.

3. Normalization generally makes answering complex queries (for example, in OLAP systems) less efficient because joins must be performed during query evaluation.
4. Normalization can make answering simple queries (for example, in OLTP systems) more efficient because such queries often involve a small number of attributes that belong to the same relation. Since decomposed relations have fewer tuples, the tuples that need to be scanned during the evaluation of a simple query are likely to be fewer.
5. Normalization generally makes simple update transactions more efficient since it tends to reduce the number of indices per table.
6. Normalization might make complex update transactions (such as *Raise the salary of all professors who taught every course required for computer science majors*) less efficient since they might involve complex queries (and thus might require complex joins).
7. Normalization results in more tables, and hence more clustered indices, which translates into more flexibility when tuning queries.

### 12.2.3 Repeating Groups

In some situations the same information can be stored in either columns or rows, and the choice can be based on performance considerations. For example, suppose one wanted to store the total sales of each salesperson in each sales region of the country. One possible solution is to store the data for each salesperson in separate rows:

---

```
CREATE TABLE SALES (
    Id          INTEGER,
    Region     CHAR(6),
    TotalSales  DECIMAL )
```

---

The pair (`Region`, `TotalSales`) is referred to as a repeating group. Unfortunately, this requires retrieving multiple rows to access information about a single salesperson. Alternatively, the information describing a salesperson could be compacted into a single row. Assuming three regions, we could store the data using this table:

---

```
CREATE TABLE SALES (
    Id      INTEGER,
    Region1Sales DECIMAL,
    Region2Sales DECIMAL,
    Region3Sales DECIMAL )
```

---

This schema has the limitation that only a fixed number of sales regions can be accommodated, but if it is generally the case that all of the information about a salesperson is retrieved at the same time, it might yield performance benefits.

### 12.2.4 Partitioning

The I/O cost of accessing a very large table can be reduced by explicitly splitting the table (in the schema) into partitions. One reason for doing this is to separate frequently accessed data in the table from data that is rarely referenced. By packing data that is frequently accessed into fewer pages, the number of I/O operations can be reduced and it is less likely that pages in the cache contain data that is not being referenced. A second reason is to make it possible to access different parts of the table concurrently, and we discuss this in Section 12.5.

With horizontal partitioning, all partitions have the same set of columns and each contains of a subset of the rows. The partitioning of the rows is based on a natural criterion that populates the partitions with disjoint subsets. For example, the table `STUDENT` might be partitioned into two partitions. Rows describing inactive students, those who have graduated, might be in a partition named `ALUMNI`. Rows describing active students, the current undergraduates, might be in a partition called `CURRENT_STUDENTS`. A page of `CURRENT_STUDENTS` in the cache is more likely to be referenced again than a page of `ALUMNI` since most references are to active students and a page of `CURRENT_STUDENTS` contains only those students. This reduces the number of I/O operations. Similarly, the cost of a scan to retrieve undergraduate information is greatly reduced.

With vertical partitioning, subsets of the columns of a table form the partitions. This can be useful when a table has many columns, and hence long rows, and some of the columns are infrequently referenced. Once again, without partitioning, performance is degraded by the need to transfer inactive data from the disk when active data is referenced. By storing the infrequently accessed columns in a separate partition, this problem can be alleviated. Oracle, for example, effectively separates infrequently accessed columns without requiring explicit partitioning. These columns are designated in the `CREATE TABLE` statement of a table that has an integrated, clustered index. In this case the infrequently accessed columns are not stored in the leaf level of the index but instead are stored in overflow pages linked to

leaf pages. Scans involving only frequently accessed columns can skip the overflow pages.

An astute reader must have noticed that vertical partitioning is conceptually the same as schema decomposition, discussed in Chapter 6. In particular, partitions must form a lossless decomposition of the original relation, which can be ensured by, for example, including a key of the relation in all partitions. However, partitioning is typically driven not by the need to normalize the schema but by other considerations. For instance, if in a STUDENT table the attributes `Address` and `Phone` are accessed infrequently, they (and the student `Id`) might be separated into a different partition even though the STUDENT table is already in BCNF. With this secondary information split off, the main partition of the STUDENT table becomes smaller and thus queries involving this table run faster.

Partitioning involves a trade-off, and in this case the price that must be paid is the additional complexity of managing and accessing multiple tables. Hence, it should be used only when the performance benefits are clear.

## 12.3 Tuning the Data Manipulation Language

A modification of the schema of a particular table can have a global impact: it can affect (hopefully improve) the performance of all the SQL statements that access the table. A modification to a query or a statement of the DBMS has a local impact: it affects the performance of only that statement. There are many nuggets of wisdom that we could include here. We have chosen just a few based on what we think offers interesting insights into SQL and the way it is processed by a DBMS.

**Avoid sorts.** Sorting is expensive and should be avoided if possible. You need to be aware of the kinds of queries that might cause an optimizer to introduce a sort into the query plan and avoid those queries if possible. In addition to the sort-merge join, duplicate elimination involves sorting. Hence, do not use `DISTINCT` unless it is important in the application. Set operators like `UNION` and `EXCEPT` also involve a sort to find duplicates, but their use may be unavoidable (however, some DBMSs provide the `UNION ALL` operator, which does not eliminate duplicates and hence does not involve a sort).

A sort is necessary to process an `ORDER BY` clause (so you should carefully consider whether an ordering on the output is necessary), and a `GROUP BY` clause will also frequently involve a sort. If sorting is unavoidable, consider presorting by using a clustered index (as in example 6 on page 436).

**Do not scan unnecessarily.** Use of “not equals” in a `WHERE` condition is likely to result in a scan. For example, the optimizer might not use an index on `CreditHours` when evaluating the condition `CreditHours ≠ 3`. This is unfortunate since it is likely that the vast majority of courses carry three credits. Accessing the few that do not through an index would therefore be appropriate. If a histogram showing the distribution of values (see Section 12.6) were available to the optimizer,

it might consider using the index if the condition were rewritten as `CreditHours IN (1,2,4)` or

CreditHours = 1 OR CreditHours = 2 OR CreditHours = 4

Similarly, a table scan will be used to resolve a condition of the form WHERE Name LIKE '%son' since a prefix of the search-key value is not provided.

An index on a column will not contain an entry for a row if the column value is null, so if you want to search for nulls you cannot use the index. A better way to handle the situation in that case is to use a default value (e.g., unknown) instead of null, and search for the default.

**Minimize communication.** Client/server communication is generally very expensive, so eliminate it where you can. A major culprit is the cursor, which invokes communication for every row fetched. Hence, if you are updating a table, try to use UPDATE statements instead of fetching the row, modifying it, and then writing it back. For example, an application might adjust the salary of employees based on the department in which they work. This might be done using a cursor in which the fetch is followed by a case statement with a branch for each department. The body of the branch for a particular department then makes the adjustment appropriate for that department. Alternatively, the application might use a sequence of UPDATE statements in which the WHERE clause of each statement in the sequence referred to a different department, and the SET clause performed the update appropriate to that department. The second approach involves far less communication, and this might compensate for any extra index searches or table scans.

If you are retrieving aggregate information, consider computing the aggregate in a stored procedure and then return only the result to the client. If you must analyze each row in the application code, see if your DBMS allows the fetch statement to retrieve multiple rows (some DBMSs support an array fetch).

**Be careful with views.** In Section 5.2.8 we discussed the fact that a query that names a view in its `FROM` clause is equivalent to a query with the view definition replacing the view name in the clause (and that it is the latter query that is analyzed by the DBMS). From this you can conclude that you are not going to get any performance gain by using a view since there is always an equivalent query that does not involve the view that will give exactly the same performance. This might seem like old news, but the really bad news is that the use of a view might actually impact performance negatively.

Consider the following view defined over the tables COURSE and CLASS of Section 4.8.

```

SELECT      C.CrsCode, C.DeptId, C.CrsName,
            CL.Enrollment, CL.MaxEnrollment
  FROM      COURSE C, CLASS CL
 WHERE      C.CrsCode = CL.CrsCode

```

---

The query

---

```

SELECT      C.CrsCode, C.CrsName
  FROM      CLASSES

```

---

pays the price of a join, whereas the query

---

```

SELECT      C.CrsCode, C.CrsName
  FROM      COURSE

```

---

achieves the same result without a join because the columns in the result set are all derived from the columns of a single base table.

Some optimizers, however, can recognize that a join is unnecessary and can eliminate the overhead.

**Consider restructuring the query.** There are often several different ways to formulate a complex query. The cost of each formulation will depend on the state of the tables involved and the indices available, and there is no easy rule that you can use to decide which formulation is best. For example, we could express the query that returns the Ids of all professors who taught a course in the spring 2003 semester in the following three ways:

1. \_\_\_\_\_

```

SELECT      *
  FROM      PROFESSOR P
 WHERE EXISTS
    (SELECT      *
      FROM      TEACHING T
      WHERE T.Semester = 'S2003' AND T.ProfId = P.Id)

```

---

2. \_\_\_\_\_

```

SELECT      *
  FROM      PROFESSOR P
 WHERE      P.Id IN
    (SELECT T.ProfId
      FROM TEACHING T
      WHERE T.Semester = 'S2003')

```

---

3.

---

```
SELECT DISTINCT P.Id, P.Name, P.DeptId
  FROM PROFESSOR P, TEACHING T
 WHERE P.Id = T.ProfId AND T.Semester = 'S2003'
```

---

The first formulation has a correlated subquery, so it looks bad. However, with an index on `(ProfId, Semester)`, the subquery can be executed efficiently since only a few rows match the condition. In the second formulation, the subquery is only executed once so even if no usable index were available and a table scan were necessary, the cost might not be excessive. The cost of the third formulation is difficult to predict without knowing more about the state of the relations involved, and so would also have to be investigated.

Although a sort is generally unavoidable in the plan for a query with a `GROUP BY` clause, you should attempt to minimize its cost by making the relation to be sorted as small as possible. One way to do this is to strengthen the `WHERE` clause. For example, the query

---

```
SELECT P.DeptId, MAX(P.Salary)
  FROM PROFESSOR P
 GROUP BY P.DeptId
 HAVING P.DeptId IN('CS', 'EE', 'Math')
```

---

produces the same result as

---

```
SELECT P.DeptId, MAX(P.Salary)
  FROM PROFESSOR P
 WHERE P.DeptId IN ('CS', 'EE', 'Math')
 GROUP BY P.DeptId
```

---

but the second formulation has lower cost since nonparticipating rows are eliminated earlier.

## 12.4 Tools

DBMS vendors usually provide a variety of tools to help with tuning. The use of these tools normally requires creation of a mock-up database in which the different plans can be tried out. A typical tool in most DBMSs is the `EXPLAIN PLAN` statement, which lets the user see the query plans the DBMS generates. This statement is not part of the SQL standard, so the syntax varies among vendors. The basic idea is first to execute a statement of the form

---

```
EXPLAIN PLAN SET queryno=123 FOR
      SELECT P.Name
```

---

```
FROM   PROFESSOR P, TEACHING T
WHERE  P.Id = T.ProfId AND T.Semester = 'F1994'
       AND T.Semester = 'CS'
```

---

which causes the DBMS to generate a query execution plan and store it as a set of tuples in a relation called PLAN\_TABLE. queryno is one attribute of that table. Some DBMSs use a different attribute name, for example, id. The plan can then be retrieved by querying PLAN\_TABLE as follows:

---

```
SELECT * FROM PLAN_TABLE WHERE queryno=123
```

---

Text-based facilities for examining query plans are extremely powerful, but these days they are used mostly by people who enjoy fixing their own cars. A busy database administrator uses text-based facilities only as a last resort because many vendors provide flashy graphical interfaces to their tuning tools. For instance, IBM has Visual Explain for DB/2, Oracle supplies Oracle Diagnostics Pack, and SQL Server from Microsoft has Query Analyzer. These tools not only show query plans, but they can also suggest indices that can speed up various queries.

By examining the query plan, you are in a position to determine whether or not the DBMS has chosen to ignore the hints you have provided (see page 450) and the indices you have so carefully created. If you are dissatisfied, you can try other strategies. More importantly, many DBMSs provide trace tools that allow you to trace the execution of a query as well as output the CPU and I/O resources used and the number of rows processed by each step. With a trace tool available, your strategy should be to coax the DBMS into using a variety of query plans and to evaluate the performance of each.

## 12.5 Managing Physical Resources

The physical resources—CPUs, I/O devices, etc.—available to the DBMS are an important factor in the performance of an application, but the application programmer is generally not in a position to control these resources. Some DBMSs, however, provide the programmer or database administrator, with mechanisms for controlling how the existing physical resources should be used.

A disk unit has a single doorway through which each read or write request for a table or index must pass in sequence. Hence, if many heavily used items are placed on the disk, a queue of waiting requests will form and response time will suffer. The lesson here is that many small disks can perform better than a single large disk because items can be spread across the disks and I/O can be performed concurrently on different disks. The discussion of RAID (Section 9.1.1) has already made this point. Since the assignment of items to disks can have a major impact on performance, DBMSs provide mechanisms that allow the user to specify the disk on which a particular item is placed.

In addition to spreading *different* tables across the available disks, concurrent access to a *single* table can be achieved by partitioning it and distributing the partitions on different disks. For example, the STUDENT table might be split into FRESH\_STUDENT, SOPH\_STUDENTS, JUN\_STUDENTS, and SEN\_STUDENTS. Note that in this case all partitions contain rows that are frequently referenced. If the partitions are placed on different disks, performance can be improved since multiple I/O requests for information about students can be performed concurrently.

Beyond distributing files across disks, the next point to note is that reading a file sequentially (e.g., a table scan) is generally more efficient than reading data randomly. This follows from the fact that DBMSs attempt to keep the pages of a file together, and as a result the seek time between the reads of two successive pages can be eliminated. But it is not so easy to take advantage of sequential I/O since, in general, a disk will store multiple files. Since requests for the files from different processes will be interleaved, the disk assembly will move from one cylinder to another. Thus, even though a process accesses a file sequentially, two successive requests from the process will pay a seek price since requests from other processes will be interleaved between them. Note that this is true even if *all* files on the disk are accessed sequentially. The lesson here is that if you want to take advantage of the fact that a file is accessed sequentially, place it on its own private disk. A good example of such a file is the log file maintained by a database system to implement atomicity.

In addition to influencing the way I/O devices are employed in an application, the programmer can influence the way CPUs are used. Generally, a single process (or thread) is assigned to execute the query plan for a particular SQL statement. Processes are sequential—they do one thing at a time. Either they require the services of a CPU to execute some code or they request an I/O transfer and wait until the operation completes. Hence, they make use of one physical device at a time. As a result, in an OLAP environment with only a few concurrent users, throughput may suffer because resource utilization is low. In an OLTP environment with many concurrent users, resource utilization will be high, but the response time possible when only a single process is assigned to execute a query plan can be unacceptable.

The response time of a query can often be improved using **parallel query processing** in which multiple concurrent processes are assigned to execute different components of the query plan. Improvement is likely when the system has multiple CPUs (so the processes can execute simultaneously), the query plan involves table scans, the query accesses very large tables (so considerable I/O is required), and the data is spread across multiple disks (so the processes can be using the disks simultaneously). DBMSs provide mechanisms, called *hints* (discussed on page 450), that the application programmer can use to request parallel query processing.

## 12.6 Influencing the Optimizer

In Chapters 10 and 11 we discussed algorithms used by the DBMS to create an efficient query execution plan. The plan selected depends on first identifying promising alternatives and then choosing from among those alternatives the plan that seems

best. The application programmer is in a position to affect this process in two ways: he can modify the schema—primarily by creating appropriate indices—to create new alternatives that the DBMS might find promising, and he can influence the choice among the alternatives. We discussed schema modification earlier. In this section we will discuss mechanisms for influencing choice.

**Statistics.** In Section 11.1 we discussed the fact that cost-based query optimizers use statistics to predict the size of the output produced by various relational expressions in order to estimate the cost of a query plan. These statistics describe not only tables but the indices that can be used to access the tables (for example, the depth, number of leaf pages, number of distinct search-key values at the leaf level, etc.). Optimizers that use this information are referred to as **cost-based optimizers**. They contrast with **rule-based optimizers** that make decisions using rules based on the structure of the SQL statement and the availability of indices but do not attempt to evaluate the costs involved. The trend in DBMS design is toward cost-based optimization.

If some statistics are good, more statistics might be even better. Additional statistics take the form of histograms describing the distribution of values in particular columns. Advanced query optimizers can make use of such information in certain cases. For example, an employee table might have an integer-valued column `Children` that gives the number of children of each employee. Without a histogram the optimizer might be able to determine from the available statistics that the maximum value in the column is 9 and that there are 10,000 rows in the table. It can then conclude that on average, for each value between 0 and 9 there are 1000 employees with that many children. As a result, a query whose result set contains the rows satisfying the WHERE condition `E.Children = 9` might use a table scan for the access path rather than an unclustered index on `Children`. (For example, if there were 500 pages in the table then it is likely that at least one row describing a fertile employee is contained in most pages.)

With a histogram the optimizer can do much better. Since the histogram contains the number of rows having each column value, the optimizer is in a position to determine that only two employees have nine children and, as a result, an access path that uses the index on `Children` is far superior to a table scan.

Maintaining a histogram is a time-consuming process. Hence, DBMSs that make use of histograms provide the programmer with a mechanism to specify the columns over which histograms are to be constructed.

If you have been reading carefully, you probably have noticed that we are describing an approach here that contradicts what was said in Section 12.1. There we argued that it was desirable to use host variables instead of literals so that query plans could be reused. Here we have made the point that literals are preferable since they allow the optimizer to use histograms. The choice of which to use has to be made for each specific application.

**Care and feeding.** Although a system might function efficiently when it is initially configured, you might discover that, over time, performance degrades even though the load is unchanged. This might be due to changes in the state of the database. Even though the size of tables might remain roughly the same, as rows are added and

deleted the organization of the tables and indices might deteriorate. For example, although the pages of a B<sup>+</sup> tree might initially be full, the steady state situation might be one in which the occupancy of pages might be low. Although this might not cause the tree to be deeper, it might substantially increase the number of leaf pages, and hence the cost of scans at the leaf level. Similarly, the space created by deleted rows in a heap file is often not recovered since rows are added at the end. As a result, the cost of table scans is increased. Each DBMS has its own quirks in the way it stores information that may result in similar inefficiencies. Check your manual.

Maintaining statistics is time consuming, and hence statistics are not normally updated each time the value of a table changes. Instead, the DBMS supports a command that causes it to reevaluate statistics. It can be invoked by the programmer at a time when the state of the table has substantially changed since the last time the statistics were evaluated. The use of outdated statistics can lead to poor query plans. Furthermore, since the query plans of stored procedures might be saved, stored procedures that access dynamically changing tables should be recompiled frequently. Similarly, if indices of tables referred to by a stored procedure change, the procedure should be recompiled.

**Hints.** Some DBMSs allow the programmer to insert suggestions, called **hints**, into an SQL statement that the query optimizer can use in constructing a query plan. For example, we saw in Chapter 11 that there are  $N!$  different orders in which  $N$  tables can be joined, and that the optimizer cannot explore all possibilities even when  $N$  is small. A major problem in joining tables is the I/O and storage costs of manipulating large intermediate tables. The wrong order can result in huge intermediate tables, which are reduced to just a few rows in the final step. Promising orders are those in which the first table to be joined is one in which the **WHERE** clause includes a selective condition that eliminates many rows that cannot possibly play a role in forming a row in the result set. Eliminating such rows early prevents them from producing useless rows at intermediate stages.

Unfortunately, it might be difficult for the optimizer to detect that a condition is selective. For example, although a condition such as `T.Model = 'Rolls Royce'` on a table containing the inventory of Slippery Joe's Used Cars might be very selective, the optimizer might have no way of knowing that. Even if a histogram were maintained on the attributes of the table, the optimizer would be stymied if '`Rolls Royce`' were replaced by a host variable `:model`. Although the optimizer might not have enough information, the programmer probably does. He can list the tables in the **FROM** clause in the desired join order and provide a hint to the effect that the optimizer should use that order in the query plan.

Hints can cover many issues. For example, different databases allow you to specify the join methodology to use (hash, sort-merge, etc.), the index to use, whether parallel query execution should be considered, and whether to optimize a query plan so that it retrieves the first row of a result set quickly (for fast response time for an interactive query) or whether it should minimize the time for retrieving the entire result set (for batch queries).

## BIBLIOGRAPHIC NOTES

A complete discussion of the principles and practices involved in tuning a DBMS (which is not specialized to any particular product) can be found in the book by [Shasha and Bonnet 2003]. The trade books and product manuals describing the measures taken in particular systems are also very informative: for SQL Server [Whalen et al 2001], for Oracle [Harrison 2001], for Sybase [Sybase 1999].

## EXERCISES

- 12.1 Choose an index for each of the following SELECT statements. Specify whether your choice is clustered or unclustered and whether it is a hash index or a B<sup>+</sup> tree.

a. \_\_\_\_\_

```
SELECT S.Name
FROM STUDENT S
WHERE S.Id = '111111111'
```

b. \_\_\_\_\_

```
SELECT S.Name
FROM STUDENT S
WHERE S.Status = 'Freshman'
```

c. \_\_\_\_\_

```
SELECT T.StudId
FROM TRANSCRIPT T
WHERE T.Grade = 'B' AND T.CrsCode = 'CS305'
```

d. \_\_\_\_\_

```
SELECT P.Name
FROM PROFESSOR P
WHERE P.Salary BETWEEN 20000 AND 150000
```

e. \_\_\_\_\_

```
SELECT T.ProfId
FROM TEACHING T
WHERE T.CrsCode LIKE 'CS' AND T.Semester = 'F2000'
```

f. \_\_\_\_\_

```
SELECT C.CrsName
FROM COURSE C, TEACHING T
WHERE C.CrsCode = T.CrsCode AND T.Semester = 'F2002'
```

- 12.2 Suppose both queries (e) and (f) from the previous exercise need to be supported. What indices should be chosen for TEACHING and COURSE?

- 12.3 The table **FACULTY** has 60,000 rows, each row occupies 100 bytes, and the database page size is  $4^k$  bytes. Assuming pages in the index and data files are 100% occupied, estimate the number of page transfers required for the following SELECT statement in each of the cases listed below.

---

```
SELECT F.DeptId
FROM   FACULTY F
WHERE  F.Id = '111111111'
```

---

- a. The table has no index.
  - b. The table has a clustered B<sup>+</sup> tree index on Id. Assume a (nonleaf) index entry has 20 characters.
  - c. The table has an unclustered B<sup>+</sup> tree index on Id. Assume a (nonleaf) index entry has 20 characters.
  - d. The table has an unclustered B<sup>+</sup> tree index on (Id, DeptId). Assume that an index entry now has 25 characters.
  - e. The table has an unclustered B<sup>+</sup> tree index on (DeptId, Id). Assume that an index entry now has 25 characters.
- 12.4 The table **FACULTY** has 60,000 rows, each row occupies 100 bytes, and the database page size is  $4^k$  bytes. The table contains an attribute **City**, indicating the city in which a professor lives, there are 50 cities with names **city10** ... **city50**, and professors are randomly distributed over the cities. Assuming that pages in the index and data files are 100% occupied, estimate the number of page transfers required for the following SELECT statement in each of the cases listed below.

---

```
SELECT F.Id
FROM   FACULTY F
WHERE  F.City > 'city10' AND F.City < 'city21'
```

---

- a. The table has no index.
  - b. The table has a clustered B<sup>+</sup> tree index on **City**. Assume a (nonleaf) index entry has 25 characters.
  - c. The table has an unclustered B<sup>+</sup> tree index on **City**. Assume an index entry has 25 characters.
  - d. The table has an unclustered B<sup>+</sup> tree index on (**City**, **Id**). Assume an index entry has 40 characters.
  - e. The table has an unclustered B<sup>+</sup> tree index on (**Id**, **City**). Assume an index entry has 40 characters.
- 12.5 Choose indices for the following SELECT statement. Specify whether your choices are clustered or unclustered, hash index or B<sup>+</sup> tree.

---

```
SELECT C.CrsName, COUNT(*)
FROM   COURSE C, TRANSCRIPT T
WHERE  T.CrsCode = C.CrsCode AND T.Semester = :sem
GROUP BY T.CrsCode, C.CrsName
HAVING COUNT(*) ≥ 100
```

---

- 12.6 Consider the following query:

---

```
SELECT T.CrsCode, T.Grade
FROM TRANSCRIPT T, STUDENT S
WHERE T.StudId = S.Id AND S.Name = 'Joe'
```

---

Assume that Id is the primary key of STUDENT, (CrsCode, Semester, StudId) is the primary key of TRANSCRIPT, and that Name is not unique. Set up a database containing these two tables on the DBMS available to you. Initialize the tables with a large number of rows. Write a program that measures the query execution time by reading the clock before and after submitting the query. Be sure to flush the cache between successive measurements (perhaps by executing a query that randomly reads a sufficient number of rows of a large dummy table).

- a. Test your understanding by making an educated guess of what query plan will be chosen by the DBMS assuming that there are no indices other than those for the primary keys. Run the query, output the query plan, and check your guess. Measure the response time.
  - b. Now assume that an unclustered index on StudId on TRANSCRIPT is added. What query plan would you expect? Run the query, check your answer, and measure the response time. Try the query under two conditions: Joe has taken very few courses; Joe has taken many courses.
  - c. In addition to the index added in (b), assume that an unclustered index on STUDENT on Name has been added and repeat the experiment.
- 12.7 Consider the table AUTHORS with attributes Name, Publ, Title, and YearPub. Assume that Name is the primary key (authors' names are unique) and hence one would expect that the DBMS would automatically create a clustered index on that attribute. Consider the statement

---

```
SELECT A.Publ, COUNT(*)
FROM AUTHORS A
WHERE . . . range predicate on YearPub . . .
GROUP BY A.Publ
```

---

- a. Assume that the statement is generally executed with a very narrow range specified in the WHERE clause (the publication year of only a few books will fall within the range). What indices would you create for the table and what query plan would you hope the query optimizer would use (include any changes you might make to the index on Name).
  - b. Repeat (a) assuming that a very broad range is generally specified.
- 12.8 Give the trigger that maintains the consistency of the database when a GPA column is added to the table STUDENT, as described in Section 12.2.2.
- 12.9 In applications that cannot tolerate duplicates it may be necessary to use DISTINCT. However, the query plan needed to support DISTINCT requires a sort, which is expensive. Therefore you should only use DISTINCT when duplicates are possible in the result set. Using the schema of Section 4.8, check the following queries to see if duplicates are possible. Explain your answer in each case.

a. \_\_\_\_\_

```

SELECT S.Name
FROM STUDENT S
WHERE S.Id LIKE '1'
_____
```

b. \_\_\_\_\_

```

SELECT S.Id
FROM STUDENT S, FACULTY F
WHERE S.Address = F.Address
_____
```

c. \_\_\_\_\_

```

SELECT C.CrsCode, COUNT(*)
FROM TRANSCRIPT T
GROUP BY T.CrsCode
_____
```

d. \_\_\_\_\_

```

SELECT F.Name, F.DeptId, C.ClassTime, C.CrsCode,
C.Semester, C.Year
FROM FACULTY, CLASS C
WHERE F.Id = C.InstructorId
_____
```

e. \_\_\_\_\_

```

SELECT S.Name, F.Name, T.Semester, T.Year
FROM FACULTY, CLASS C, TRANSCRIPT T, STUDENT S
WHERE F.Id = C.InstructorId AND S.Id = T.StudId AND
C.CrsCode = T.CrsCode AND
C.SectionNo = T.SectNo AND
C.Year = T.Year AND C.Semester = T.Semester
_____
```

**12.10** A particular query can have several formulations, and a query optimizer may produce different query plans with different costs for each.

- Assume that the Computer Science Department teaches only three 100-level courses: CS110, CS113, and CS114. Write an SQL statement whose result set contains the course codes of all courses that have these as prerequisites in three ways: using OR, UNION, and a nested subquery involving LIKE.
- Write an SQL statement whose result set contains the names of all computer science courses that are prerequisites to other courses in three ways: using a join, a nested subquery involving EXISTS, and a nested subquery involving IN.

# 13

## An Overview of Transaction Processing

The transactions of a transaction processing application should satisfy the ACID properties that we discussed in Chapter 2—atomic, consistent, isolated, and durable. As transaction designers, we are responsible for the consistency of the transactions in our system. We must ensure that, if each transaction is executed by itself (with no other transactions running concurrently), it performs correctly—that is, it maintains the database integrity constraints and performs the transformations listed in its specification. The remaining properties—atomicity, isolation, and durability—are the responsibility of the underlying transaction system. In this chapter we give an overview of how these remaining features are implemented.

### 13.1 Isolation

The transaction designer is responsible for designing each transaction so that, if it is executed by itself and the initial database correctly models the current state of the real-world enterprise, the transaction performs correctly and the final database correctly models the (new) state of the enterprise. However, if the transaction processing system executes a set of such transactions concurrently—in some interleaved fashion—the effect might be to transform the database to a state that does not correspond to the real-world enterprise it was modeling or to return incorrect results to the user.

The schedule of Figure 2.4 on page 23 is an example of an incorrect concurrent schedule. In that schedule, two registration transactions completed successfully, but the course became oversubscribed and the count of the total number of registrants was only incremented by one. The cause of the failure was the particular way the operations of the two transactions were interleaved. Both transactions read the same value of *cur\_reg*, so neither took into account the effect of the other. We referred to this situation as a lack of isolation—the I in ACID.

One way for the system to achieve isolation is to run transactions one after the other in some serial order—each transaction is started only after the previous transaction completes, and no two transactions run concurrently. The resulting **serial schedule** will be correct since we assume that transactions that run by themselves perform correctly: each transaction is consistent. Thus, assuming that the database

correctly models the real world when the schedule starts, and given that the first transaction is consistent, the database will correctly model the real world when that transaction completes. Hence the second transaction (which is also consistent) will run correctly and leave the database in a correct state for the third, and so forth.

Unfortunately, for many applications serial execution results in unacceptably small transaction throughput (measured in transactions per second) and unacceptably long response time for users. Although restricting transaction processing systems to run only serial schedules is impractical, serial schedules are important because they serve as the primary measure of correctness. Since serial schedules must be correct, a nonserial schedule is also correct if it has the same effect as a serial schedule.

Note that the implication goes in only one direction. Nonserial schedules that do not have the same effect as serial schedules are *not necessarily* incorrect. We will see that most DBMSs give the application designer the flexibility to run such nonserial schedules. First, however, we discuss serializable schedules—schedules that are equivalent to serial schedules.

### 13.1.1 Serializability

One way to improve performance over serial execution and yet achieve isolation is to allow interleaved schedules that are serializable. A **serializable schedule** is a schedule that is equivalent to a serial schedule. We discuss the meaning of equivalence below.

As a simple example, assume that in a banking system, transactions read and write database items  $\text{Balance}_i$ , where  $\text{Balance}_i$  is the value of the balance in  $\text{Account}_i$ . Assume that  $T_1$  reads and writes only  $\text{Balance}_a$  and  $\text{Balance}_b$  (perhaps it transfers money from one account to the other), and transaction  $T_2$  reads and writes only  $\text{Balance}_c$  (perhaps it makes a deposit in that account). Even if execution of the transactions is interleaved, as in the schedule

$$r_1(\text{Balance}_a) \ w_2(\text{Balance}_c) \ w_1(\text{Balance}_b)$$

$T_2$ 's write has no effect on  $T_1$ , and  $T_1$ 's read and write have no effect on  $T_2$ . Hence the overall effect of the schedule is the same as if the transactions had executed serially in either the order  $T_1 \ T_2$  or the order  $T_2 \ T_1$ —that is, in one of the following serial schedules:

$$r_1(\text{Balance}_a) \ w_1(\text{Balance}_b) \ w_2(\text{Balance}_c)$$

or

$$w_2(\text{Balance}_c) \ r_1(\text{Balance}_a) \ w_1(\text{Balance}_b)$$

Note that both of the equivalent serial schedules are obtained from the original schedule by interchanging operations that commute. In the first case, the two write operations have been interchanged. They commute because they operate on distinct items and hence leave the database in the same final state no matter in

which order they execute. In the second case, we have interchanged  $r_1(Balance_a)$  and  $w_2(Balance_c)$ . These operations also commute because they operate on distinct items, and hence, in both orders, the same value of  $Balance_a$  is returned to  $T_1$  and  $Balance_c$  is left in the same final state.

Suppose that in addition both  $T_1$  and  $T_2$  read a common item, today's date,  $date$ . Again, the overall effect is the same as if the transactions had executed serially in either order. Thus, the schedule

$$r_1(Balance_a) \ r_2(date) \ w_2(Balance_c) \ r_1(date) \ w_1(Balance_b) \quad \text{13.1}$$

has the same effect as does the serial schedule

$$r_1(Balance_a) \ r_1(date) \ w_1(Balance_b) \ r_2(date) \ w_2(Balance_c) \quad \text{13.2}$$

in which all of  $T_1$ 's operations precede those of  $T_2$ . The equivalence between the two schedules is again based on commutativity. The new feature illustrated by this example is that operations do not have to access distinct items in order to commute. In this case,  $r_1(date)$  and  $r_2(date)$  commute because they both return the same value to the transactions in either execution order.

In general, requests (from different transactions) commute if either of the following holds:

- They refer to different data items.
- They are both read requests.

In all of the above cases we say that the interleaved schedule is serializable since we can find at least one equivalent serial schedule. Furthermore, the equivalent serial schedule can be produced from the interleaved schedule by a sequence of interchanges of adjacent, commuting operations of different transactions. For example, the first interchange in going from schedule (13.1) to schedule (13.2) is to interchange  $w_2(Balance_c)$  and  $r_1(date)$ .

In a serial schedule a transaction can affect the execution of a subsequent transaction (one that starts after that transaction commits) by causing a transition to a new database state. For example, the new balance established by executing a deposit transaction affects the balance reported by a subsequent read-balance transaction. However, the two transactions do not affect each other in any other way, and we say their execution is isolated. Because of the equivalence between serial and serializable schedules, we say that the transactions in a serializable schedule are also isolated.

In the schedule shown in Figure 2.4 on page 23 the two registration transactions have affected each other in a way that could not have happened in a serial schedule. Since they both increment the same value of  $cur\_reg$ , the schedule produces an erroneous final state. The schedule is not serializable, and it is easy to see that we cannot obtain an equivalent serial schedule by a series of interchanges of adjacent commuting operations. A read and a write operation on the same data item do not commute: the value returned by the read depends on whether it precedes or follows

the write. Similarly, two write operations on the same item do not commute: the final value of the item depends on which write came last.

In general, we are interested in specifying when a schedule,  $S$ , of some set of concurrently executing transactions is serializable: it is equivalent to (i.e., has the same effect as) some serial schedule,  $S_{ser}$ , of that set. Informally, what is required is that in both schedules

- The values returned by the corresponding read operations in the two schedules are the same.
- The write operations to each data item occur in the same order in both schedules.

To understand these conditions, note that the computation performed by a program depends on the values of the data items that it reads. Hence, if each read operation in a transaction returns the same value in schedules  $S$  and  $S_{ser}$ , the computations performed by the transaction will be identical in both schedules, and hence the transaction will write the same values back to the database. If the write operations occur in the same order in both schedules, they leave the database in the same final state. Thus,  $S$  has the same effect as (and hence is equivalent to)  $S_{ser}$ .

Database systems can guarantee that schedules are serializable. By allowing serializable, in addition to serial, schedules, they allow more concurrency, and hence performance is improved. In addition, database systems offer less stringent notions of isolation that do not guarantee that schedules will be serializable, and hence they support even more concurrency and better performance. Since nonserializable schedules are not necessarily equivalent to serial schedules, correctness is not guaranteed. Therefore, less stringent notions of isolation must be used with caution.

The part of the transaction processing system responsible for enforcing isolation is called the **concurrency control**. The concurrency control enforces isolation by controlling the schedule of database operations. When a transaction wishes to read or write a database item, it submits its request to the concurrency control. On the basis of the sequence of requests it has granted up to that point and given the fact that it does not know what requests might arrive in the future, the concurrency control decides whether isolation can be guaranteed if it grants the request at that time. If isolation cannot be guaranteed, the request is not granted. The transaction is either made to wait or is aborted. We describe one way a concurrency control might make these decisions in Section 13.1.2.

### 13.1.2 Two-Phase Locking

Most concurrency controls in commercial systems implement serializability using a strict **two-phase locking protocol** [Eswaran et al. 1976]. The protocol associates a lock with each item in the database and requires that a transaction hold the lock before it can access the item. When a transaction wishes to read (write) a database item, it submits a request to the concurrency control, which must grant to the transaction a **read lock (write lock)** on the item before passing the request on to the database system module that performs the access. The locks are requested, granted, and released according to the following rules:

Granted Mode		
Requested Mode	read	write
read		X
write	X	X

**FIGURE 13.1** Conflict table for a concurrency control. Conflicts between lock modes are denoted by X.

1. If a transaction,  $T$ , requests to read an item and no other transaction holds a write lock on that item, the control grants a read lock on that item to  $T$  and allows the operation to proceed. Note that since other transactions might be holding read locks that were granted at an earlier time, read locks are often referred to as **shared locks**. Note that the requested read operation commutes with the previously granted read operations.
2. If a transaction,  $T$ , requests to read an item and another transaction,  $T'$ , holds a write lock on that item,  $T$  is made to wait until  $T'$  completes (and releases its lock). We say that the requested read operation **conflicts** (does not commute) with the previously granted write operation.
3. If a transaction,  $T$ , requests to write an item and no other transaction holds a read or write lock on that item, the control grants  $T$  a write lock on that item and allows the operation to proceed. Because a write lock excludes all other locks, it is often referred to as an **exclusive lock**.
4. If a transaction,  $T$ , requests to write an item and another transaction,  $T'$ , holds a read or write lock on that item,  $T$  is made to wait until  $T'$  completes (and releases its lock). We say that the requested write operation **conflicts** (does not commute) with the previously granted read or write operation.
5. Once a lock has been granted to a transaction, the transaction retains the lock. A read lock on an item allows the transaction to do subsequent reads of that item. A write lock on an item allows the transaction to do subsequent reads or writes of that item. When the transaction completes, it releases all locks it has been granted.

Notice that the effect of the rules is that, if a request does not conflict with (that is, it commutes with) the previously granted requests of other active transactions, it can be granted. Figure 13.1 displays the conflict relation for an item in tabular form. An X indicates a conflict.

The concurrency control uses locks to remember the database operations previously performed by currently active transactions. It grants a lock to a transaction to perform an operation on an item only if the operation commutes with (that is, does not conflict with) all other operations on the item that have previously been performed by currently active transactions. For example, since two reads on an item commute, a read lock can be granted to a transaction even though a different transaction currently holds a read lock on that item. In this way, the control guarantees that the operations of active transactions commute, and hence the schedule

of these operations is equivalent to a serial schedule. This result forms the basis of a proof that any schedule produced by the concurrency control is serializable.

The concurrency control has the property that once a transaction acquires a lock, it holds the lock until it completes. This is a special case of a more general class of concurrency controls referred to as **two-phase** controls. In general, with a two-phase control, each transaction goes through a locking phase in which it obtains locks on the items that it accesses, and then an unlocking phase in which it releases locks. Once it enters the second phase it is not permitted to acquire any additional locks. In our case the second phase is collapsed to a single point in time when the transaction completes. This makes the concurrency control **strict**.

In a **nonstrict** two-phase concurrency control, the unlocking phase starts after the transaction has obtained all of the locks it will ever need and continues until the transaction completes. During the second phase the transaction can release a lock at any time.

Note that since, in a strict control, locks are held until a transaction completes, the database system does not have to provide an explicit command that a transaction can use to release a lock. Such a mechanism (e.g., an unlock command) has to be provided, however, by a database system that supports a nonstrict control.

While the nonstrict two-phase protocol guarantees serializability, problems arise when transactions abort. If transaction  $T_1$  modifies a data item,  $\text{Balance}_a$ , and then unlocks it in phase two, a second transaction,  $T_2$ , can read the new value and subsequently commit. Since  $T_1$  unlocked  $\text{Balance}_a$  before completing, it might subsequently abort. Atomicity requires that an aborted transaction have no effect on the database state, so if  $T_1$  aborts,  $\text{Balance}_a$  will be restored to its original value. These events are recorded in the following schedule:

$w_1(\text{Balance}_a)$   $r_2(\text{Balance}_a)$   $w_2(\text{CreditLimit})$   $\text{commit}_2$   $\text{abort}_1$  13.3

$T_2$  has written a new value to  $\text{CreditLimit}$ , and that value is based on the value of  $\text{Balance}_a$  that it read. Since that value was produced by a transaction that subsequently aborted, a violation of atomicity has occurred. Even though  $T_1$  aborted, it had an effect on the value  $T_2$  wrote.  $T_1$  might have deposited money into the account, and  $T_2$  might have based its computation of  $\text{CreditLimit}$  on the new balance. Since  $T_1$  aborted, the value of  $\text{CreditLimit}$  is in all probability incorrect.

For this reason, in a nonstrict two-phase concurrency control, although read locks can be released during phase two, write locks are not released early but are held until commit time.

Concurrency controls that use strict two-phase locking produce schedules that are serializable in commit order. By this we mean that a schedule,  $S$ , is equivalent to a serial schedule,  $S_{\text{ser}}$ , in which the order of transactions is the same as the order in which they commit in  $S$ . To understand why this is so, observe that write locks are not released until commit time, so a transaction is not allowed to read (or write) an item that has been written by a transaction that has not yet committed. Thus, each transaction "sees" the database produced by the sequence of transactions that committed prior to its completion. Nonstrict two-phase locking protocols produce schedules that are serializable, but not necessarily in commit order.

For many applications, users intuitively expect transactions to be serializable in commit order. For example, you expect that after your bank deposit transaction has committed, any transaction that commits later will see the effect of that deposit.

The idea behind a two-phase protocol is to hold locks until it is safe to release them. Early release of locks beyond what is allowed by a nonstrict two-phase control can result in an inconsistent database state or can cause transactions to return incorrect results to the user. Since performance considerations force database systems to provide mechanisms that support early release, the database community has developed special jargon to describe some of the problems, or **anomalies**, that can occur.

- **Dirty read.** Suppose that transaction  $T_2$  reads an item,  $\text{Balance}_a$ , written by transaction  $T_1$  before  $T_1$  completes. This might happen if  $T_1$  releases the write lock it has acquired on  $\text{Balance}_a$  before it commits. Since the value of  $\text{Balance}_a$  returned by the read was not written by a committed transaction it might never appear in the database. This is referred to as a **dirty read**. The problem in schedule (13.3) is caused by a dirty read.
- **Nonrepeatable read.** Suppose that transaction  $T_1$  reads an item,  $\text{Balance}_a$ , and then releases the read lock it has acquired before it completes. Another transaction,  $T_2$ , might then write  $\text{Balance}_a$  and commit. If  $T_1$  reacquires a read lock on  $\text{Balance}_a$  and reads it again, the value returned by the second read will not be the same as the value returned by the first. We refer to this as a **nonrepeatable read**. This situation is illustrated by the schedule

$$r_1(\text{Balance}_a) \quad w_2(\text{Balance}_a) \quad \text{commit}_2 \quad r_1(\text{Balance}_a)$$

Note that in this example  $T_2$  has committed prior to the second read, and so the second read is not dirty. However, since  $T_1$  releases the read lock and then reacquires it, it is not two-phase. While a nonrepeatable read might seem to be an unimportant issue (why would a transaction read the same item twice?), it is a symptom of a more serious problem. For example, suppose  $\text{List}$  is a list of passengers on an airline flight and  $\text{Count}$  is the count of passengers on the list. In the following schedule,  $T_2$  reserves a seat on the flight and hence adds an entry to  $\text{List}$  and increments  $\text{Count}$ .  $T_1$  reads both  $\text{List}$  and  $\text{Count}$ , thus sees the passenger list before the new passenger was added and the passenger count after it was incremented—an inconsistency.

$$r_1(\text{List}) \quad r_2(\text{List}) \quad w_2(\text{List}) \quad r_2(\text{Count}) \quad w_2(\text{Count}) \quad \text{commit}_2 \quad r_1(\text{Count})$$

This schedule is directly related to the previous one. In both cases, locking is not two-phase and  $T_2$  overwrites an item that an active transaction,  $T_1$ , has read.

- **Lost update.** Suppose that a deposit transaction in a banking system reads the balance in an account,  $\text{Balance}_a$ , calculates a new value based on the amount deposited, and writes the new value back to  $\text{Balance}_a$ . If the transaction releases the read lock it has acquired on  $\text{Balance}_a$  before acquiring a write lock, two deposit transactions on the same account can be interleaved, as illustrated in the following schedule:

$$r_1(\text{Balance}_a) \quad r_2(\text{Balance}_a) \quad w_2(\text{Balance}_a) \quad \text{commit}_2 \quad w_1(\text{Balance}_a) \quad \text{commit}_1 \quad \mathbf{13.4}$$

Unfortunately, the amount deposited by  $T_2$  does not appear in the final value of  $\text{Balance}_a$  and hence this problem is referred to as a **lost update**. The effect of  $T_2$  is lost because the value written by  $T_1$  is based on the original value of  $\text{Balance}_a$  rather than the new value written by  $T_2$ .

These anomalies, as well as other as-yet-unnamed anomalies, can cause transactions to return incorrect results and the database to become inconsistent.

### 13.1.3 Deadlock

Suppose that transactions  $T_1$  and  $T_2$  both want to deposit money into the same account and hence they both want to execute the sequence

$$r(\text{Balance}_a) \text{ } w(\text{Balance}_a)$$

In one possible partial schedule,  $T_1$  read locks and reads  $\text{Balance}_a$ ;  $T_2$  read locks and reads  $\text{Balance}_a$ ;  $T_1$  requests to write  $\text{Balance}_a$  but is made to wait because  $T_2$  has a read lock on it;  $T_2$  requests to write  $\text{Balance}_a$  but is made to wait because  $T_1$  has a read lock on it.

$$r_1(\text{Balance}_a) \text{ } r_2(\text{Balance}_a) \text{ Request\_}w_1(\text{Balance}_a) \text{ Request\_}w_2(\text{Balance}_a)$$

At this point,  $T_1$  is waiting for  $T_2$  to complete, and  $T_2$  is waiting for  $T_1$  to complete. Both will wait forever because neither will ever complete.

This situation is called a **deadlock**. More generally, a deadlock occurs whenever there is a wait loop—that is, a sequence of transactions,  $T_1, T_2, \dots, T_n$ , in which each transaction,  $T_i$ , is waiting to access an item locked by  $T_{i+1}$ , and  $T_n$  is waiting to access an item locked by  $T_1$ . Although two-phase locking is particularly prone to deadlock, deadlock can occur with any concurrency control that allows a transaction to hold a lock on one item when it requests a lock on another item. Such controls must have a mechanism for detecting a deadlock and then for aborting one of the transactions in the wait loop so that at least one of the remaining transactions can continue.

With one such mechanism, whenever a transaction is forced to wait, the control checks to see whether a loop of waiting transactions will be formed. Thus, if  $T_1$  must wait for  $T_2$ , the control checks to see if  $T_2$  is waiting and, if so, for what. As this process continues, a chain of waiting transactions is uncovered and a deadlock is detected if the chain loops back on itself. Another mechanism uses **timeout**. Whenever a transaction has been waiting for a “long” time (as defined by the system administrator), the control assumes that a deadlock exists and aborts the transaction.

Even with detection and abortion, deadlocks are undesirable because they waste resources (the computation performed by the aborted transaction must be redone) and slow down the system. Application designers should design tables and transactions so as to reduce the probability of deadlocks.

### 13.1.4 Locking in Relational Databases

Up to this point, our discussion of locking has assumed that a transaction requests access to some named item (for example,  $\text{Balance}_a$ ). Locking takes a different form in a relational database, where transactions access tuples. Although tuples can be locked, a transaction describes the tuples it wants to access not by naming them (they do not have names), but by using a condition that the tuples must satisfy. For example, the set of tuples read by a transaction using a **SELECT** statement is specified by a selection condition in the **WHERE** clause.

For example, an **ACCOUNTS** table in a banking system might contain a tuple for each separate account, and a transaction  $T_1$  might read all tuples that describe the accounts controlled by depositor Mary as follows:

---

```
SELECT *
FROM ACCOUNTS A
WHERE A.Name = 'Mary'
```

---

In this case,  $T_1$  reads all tuples in **ACCOUNTS** that satisfy the condition that the value of their **Name** attribute is Mary. The condition **A.Name = 'Mary'** is called a **predicate**.

As with nonrelational databases, we can ensure serializability with a locking protocol. In designing such a protocol, we must decide what data items to lock. One approach is to always lock an entire table, even if only a few tuples in it are accessed. In contrast to tuples, tables are named in the statements that access them. Thus, the **SELECT** statement reads the data item(s)—tables—named in the **FROM** clause. Similarly, **DELETE**, **INSERT**, and **UPDATE** write the named tables. With this approach to locking, the concurrency control protocols described in the previous sections can be used and will yield serializable schedules. The problem is that table locks are coarse: a table might contain thousands of tuples, and locking an entire table because a small number of its tuples are being accessed might result in a serious loss of concurrency.

A second approach is to lock only the tuples returned by the **SELECT** statement. For example, in processing the above **SELECT** statement, only tuples describing Mary's accounts are locked. Unfortunately, this approach does not work. To understand the problem consider a database consisting of two tables: the **ACCOUNTS** table introduced earlier that has a tuple for every account in the bank, and a **DEPOSITORS** table that has a tuple for every depositor. One attribute of **DEPOSITORS** is **TotalBalance**, whose value is the sum of the balances in all the accounts owned by a particular depositor.

Two transactions access these tables. An audit transaction,  $T_1$ , for Mary might execute the **SELECT** statement

---

```
SELECT SUM(Balance)
FROM ACCOUNTS A
WHERE A.Name = 'Mary'
```

---

to compute the sum of the balances in all of Mary's accounts, and then it might compare the value returned with the result of executing

---

```
SELECT D.TotalBalance
FROM DEPOSITORS D
WHERE D.Name = 'Mary'
```

---

Concurrently,  $T_2$ , a new account transaction for Mary, might create a new account for Mary with an initial balance of \$100 by inserting a tuple into ACCOUNTS using the statement

---

```
INSERT INTO ACCOUNTS
VALUES ('10021', 'Mary', 100)
```

---

and then updating TotalBalance by 100 in the appropriate tuple in DEPOSITORS using

---

```
UPDATE DEPOSITORS
SET TotalBalance = TotalBalance + 100
WHERE Name = 'Mary'
```

---

The operations on ACCOUNTS performed by  $T_1$  and  $T_2$  conflict since  $T_2$ 's INSERT does not commute with  $T_1$ 's SELECT. If INSERT is executed before SELECT, the inserted tuple will be returned by SELECT; otherwise, it will not be returned. In our earlier discussion we saw that a request to perform an operation should not be granted if it conflicts with an operation executed earlier by a still active transaction. Hence, we are justified in expecting that invalid results will be obtained if the execution of  $T_2$  is interleaved between the time  $T_1$  reads ACCOUNTS and the time it reads DEPOSITORS. In this case, the value of TotalBalance read by  $T_1$  will not be equal to the sum of the Balances it read in its first statement, and the schedule is not serializable. (The operations on DEPOSITORS similarly conflict.)

The question is "Will tuple locking prevent this interleaving?" Unfortunately, the answer is "No." The locks that  $T_1$  acquires on Mary's tuples in ACCOUNTS as a result of executing the first SELECT statement do not prevent  $T_2$  from inserting an entirely new tuple into the table. We can conclude then that tuple locking does not guarantee serializable schedules. Table locking, on the other hand, would prevent the problem since it inhibits all accesses to the table.

The root cause of the problem is that  $T_2$  has altered the contents of the set of tuples referred to by the predicate Name = 'Mary' by adding the new tuple. In this situation, the new tuple is referred to as a **phantom** because  $T_1$  thinks it has locked all the tuples that satisfy the predicate but, unknown to  $T_1$ , a tuple satisfying the predicate has been inserted by a concurrent transaction. A phantom can lead to nonserializable schedules and hence invalid results. Hence, we have discovered a new anomaly.

### 13.1.5 Isolation Levels

Locks impede concurrency and hence performance. It is therefore desirable to minimize their use. Table locking used in a two-phase protocol produces serializable schedules but, because of the size of the item locked, has the greatest impact on concurrency. Tuple locking is more efficient but can result in nonserializable schedules even when used in a two-phase protocol (because of phantoms). Because locks are held until commit time, strict protocols inhibit concurrency more than nonstrict protocols.

For these reasons, most commercial DBMSs allow the application designer to choose among several locking protocols. The protocols differ in the items that they cause to be locked and in how long the locks are held. These options are often described in terms of the ANSI standard isolation levels. The application designer should choose a level that guarantees both that the application will execute correctly and that concurrency will be maximized. Levels other than the most stringent one permit nonserializable schedules. However, *for a particular application*, the nonserializable schedules permitted by a particular level might not lead to incorrect results or, for that application, all the schedules produced by the lower isolation level might be serializable and hence correct.

Each ANSI standard isolation level is specified in terms of the anomalies that it prevents. An anomaly that is prevented at one level is also prevented at each stronger (higher) level. The levels are (in the order of increasing strength):

- **READ UNCOMMITTED.** Dirty reads are possible.
- **READ COMMITTED.** Dirty reads are not permitted (but nonrepeatable reads and phantoms are possible).
- **REPEATABLE READ.** Nonrepeatable and dirty reads are not permitted (but phantoms are possible).
- **SERIALIZABLE.** Nonrepeatable reads, dirty reads, and phantoms are not permitted. Transaction execution must be serializable.

Describing isolation levels in terms of the anomalies they do or do not permit is a dangerous business. What about other anomalies that you might not have thought about? The standards organization must have had this in mind when they specified the **SERIALIZABLE** isolation level. *All* anomalies are ruled out if schedules are serializable, not just nonrepeatable reads, dirty reads, and phantoms. We will see an example of another anomaly shortly.

We need to discuss one other issue related to isolation. Recall that the query plan that implements a particular SQL statement is a complex program that generally involves a significant amount of computation and I/O and whose execution might take minutes or more. In order to provide reasonable throughput for demanding applications, many DBMSs support the concurrent execution of several SQL statements. This is a micro form of interleaving that we have not considered. It is interleaving at the instruction level rather than at the SQL statement level. Is the execution of the query plans for individual statements to be isolated? Clearly the answer must be "yes." Since a query plan involves access to data structures internal to the DBMS

(e.g., buffers, tables for implementing locks) in addition to the database itself, internal locks—called **latches**—are used to provide this type of isolation. We do not discuss this aspect of isolation any further.

While a particular isolation level can be implemented in a variety of ways, locking is a common technique. It is instructive to propose a particular discipline for doing this. Each level uses locks in different ways. For most levels, locks are acquired by transactions on items they access when executing an SQL statement. Depending on how the item is accessed, the lock can be either a read lock or a write lock. Once acquired, it can be held until commit time—in which case it is referred to as a **long-duration lock**—or it can be held only as long as the statement is being executed—in which case it is referred to as a **short-duration lock**. In the implementation we describe, write locks at all isolation levels are long-duration locks on the entire table and read locks are handled differently at each level.

- **READ UNCOMMITTED.** A read is performed without obtaining a read lock. Hence, a transaction executing at this level can read a tuple on which another transaction holds a write lock. As a result the transaction might read uncommitted (dirty) data.
- **READ COMMITTED.** Short-duration read locks are obtained for each tuple before a read is permitted. Hence, conflicts with write locks will be detected, and the transaction will be made to wait until the write lock is released. Since write locks are of long duration, only committed data can be read. However, read locks are released when the read is completed, so two successive reads of the same tuple by a particular transaction might be separated by the execution of another transaction that updates the tuple and then commits. This means that reads might not be repeatable.
- **REPEATABLE READ.** Long-duration read locks are obtained on each tuple returned by a SELECT. Hence, a nonrepeatable read is not possible, although phantoms are.
- **SERIALIZABLE.** Serializable schedules can be guaranteed if long-duration read locks are acquired on all tables read. This eliminates phantoms, although it reduces concurrency. A better implementation involves locking a table and/or portions of an index that is used to access a table. A description of that algorithm, however, is beyond the scope of this chapter.

Note that, while the standard does not explicitly speak of lost updates, this anomaly is allowed at **READ COMMITTED** but eliminated at **REPEATABLE READ**. Take a look at schedule (13.4) on page 461. The long-term read lock acquired on  $Balance_a$  by  $T_1$  would have prevented the request  $w_2(Balance_a)$  from being executed. A closer examination of this situation shows that the two transactions would have become deadlocked—not a happy situation, but at least the lost update does not occur.

The SQL standard specifies that different transactions in the same application can execute at different isolation levels, and each such transaction sees or does not see the phenomena corresponding to its level. The locking implementation described above enforces this. For example, a transaction that executes at **REPEATABLE**

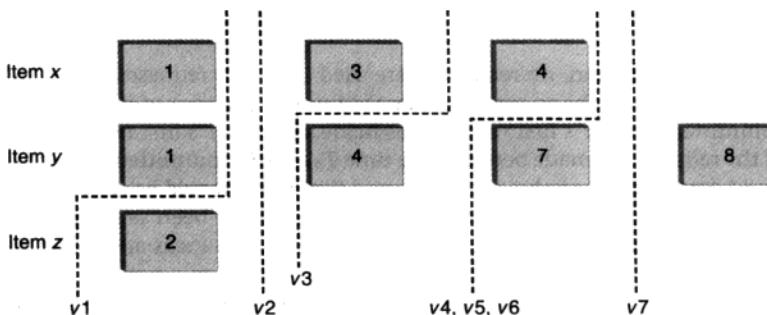


FIGURE 13.2 Multiversion database.

READ will see the same value if it reads a particular tuple several times, even though other transactions execute at other levels, since it gets a long-duration read lock on the tuple, while all other transactions must get long-duration write locks to update it. Similarly, a transaction that executes at SERIALIZABLE sees a view of the database that is serialized with respect to the changes made by all other transactions, regardless of their levels: it sees either all updates made by such a transaction or none. This follows from the same considerations.

One particularly troublesome type of nonrepeatable read occurs when a transaction accesses a tuple through a cursor. Since at READ COMMITTED read locks are short term, the tuple can be changed by a concurrent transaction while the cursor is pointing to it. Some commercial database systems support an isolation level called CURSOR STABILITY, which is essentially READ COMMITTED with the additional feature that a read lock is maintained on a tuple as long as a cursor is pointing at it. Thus, the read of the tuple is repeatable if the cursor is not moved. You can think of this as a medium-duration lock.

**SNAPSHOT isolation.** An isolation level that is not part of the ANSI standard, but is provided by at least one commonly used DBMS (Oracle), is called SNAPSHOT isolation. SNAPSHOT isolation uses a **multiversion** database: When a (committed) transaction updates a data item, the item's old value is not discarded. Instead, the old value (or version) is preserved and a new version is created. At any given time, therefore, multiple versions of the same item exist in the database. The system can construct, for any  $i$ , the value of each item that includes the effects of the  $i^{\text{th}}$  transaction to commit and of all transactions that committed at a prior time.

Figure 13.2 illustrates this situation assuming that transactions are consecutively numbered in the order in which they commit and that each version of an item is tagged with the index of the transaction that produced it. The successive values of an item are aligned from left to right. A version of the database as a whole is indicated with a dashed line. Thus, in version  $v_4$ —the version of the database at the time of the completion of the fourth transaction to commit,  $T_4$ —the values of  $x$  and  $y$  were

written by  $T_4$  and the value of  $z$  was written by  $T_2$ . Each database version is referred to as a **snapshot**.

With SNAPSHOT isolation, no read locks are used. All reads requested by transaction  $T$  are satisfied using the snapshot produced by the sequence of transactions that were committed when  $T$ 's first request was made. Thus, if  $T$ 's first read request was  $r(z)$  and the request was made between the time  $T_4$  and  $T_5$  committed, the value returned would have been the value of  $z$  in  $v4$ , and that value would have been created by  $T_2$ . All of  $T$ 's subsequent read requests would also have been satisfied from  $v4$ . Since a snapshot once produced is never changed, no read locks are necessary and read requests are never delayed.

The updates of each transaction are controlled by a protocol called **first-committer-wins**. A transaction,  $T$ , is allowed to commit if there is no other transaction that (1) committed between the time  $T$  made its first read request and the time it requested to commit and (2) updated a data item that  $T$  had also updated. Otherwise,  $T$  is aborted.

The first-committer-wins feature eliminates dirty reads since a transaction reads values from a snapshot that, by definition, contains values written by committed transactions. It eliminates lost updates, which occur when two concurrently active transactions write to the same data item. The problem encountered by the audit in Section 13.1.4 is also eliminated since all of its reads would be satisfied using the same snapshot.

However, SNAPSHOT isolation allows nonserializable—and hence possibly incorrect—schedules. For example, suppose a database integrity constraint asserts that the sum of  $x$  and  $y$  must be nonnegative. In the schedule

$$r_1(x) \ r_1(y) \ r_2(x) \ r_2(y) \ w_1(y) \ w_2(x)$$

$T_1$  and  $T_2$  each read the values of  $x$  and  $y$  from the same snapshot. Assuming the sum is 5,  $T_1$  might elect to subtract 5 from  $x$  and  $T_2$  might elect to subtract 5 from  $y$ . Each transaction is consistent, but the schedule causes a violation of the constraint. The schedule is not serializable because it is not possible to produce an equivalent serial schedule by a series of interchanges of adjacent commuting operations. Unfortunately, it is allowed by SNAPSHOT isolation since the transactions write to different items in the database. This is an example of a new anomaly, called **write skew**.

The implementation of SNAPSHOT isolation is complicated by the fact that a multiversion database must be maintained. In practice, however, it is not possible to maintain all versions. Old versions eventually must be discarded. This can be a problem for long-running transactions since they must be aborted if they request access to a version that no longer exists.

### 13.1.6 Lock Granularity and Intention Locks

For performance reasons many commercial concurrency controls lock a unit larger than an individual data item. For example, instead of an item that might occupy only several bytes, the concurrency control might lock the entire disk page on which that item is stored. Since such a control locks *more* items than are actually necessary,

it produces the same or a higher level of isolation than a concurrency control that locks only the item whose lock has been requested. For example, instead of locking a page containing some of the tuples in a table, the concurrency control might lock the entire table.

The size of the unit locked determines the lock **granularity**—it is **fine** if the unit locked is small and **coarse** otherwise. Granularity forms a hierarchy based on containment. Typically, a fine-granularity lock is a tuple lock. A medium-granularity lock is a lock on the page containing a desired tuple. A coarse-granularity lock is a table lock that covers all the pages of a table.

Fine-granularity locks have the advantage of allowing more concurrency than coarse-granularity locks since transactions lock only the data items they actually use. However, the overhead associated with fine-granularity locking is greater. Transactions using fine-granularity locks generally hold more locks since a single coarse-granularity lock might grant access to several items used by the transaction. (For example, the tuples on a disk page are generally elements of a single table, and there is a reasonable probability that a transaction accessing one such tuple will also access another. A single page lock grants permission to access both.) Therefore, more space is required in the concurrency control to retain information about fine-granularity locks. Also, more time is expended in requesting locks for each individual unit.

Because of these trade-offs, database systems frequently offer granularity at several different levels, and different levels can be used within the same application. Locking at multiple granularities introduces some new implementation problems. Suppose that transaction  $T_1$  has obtained a write lock on a particular tuple in a table and transaction  $T_2$  requests a read lock on the entire table (it wants to read all of the tuples in the table). The concurrency control should not grant the table lock because  $T_2$  should not be allowed to read the tuple that was locked by  $T_1$ . The problem is how the concurrency control detects the conflict since the conflicting locks are on different items. The control needs a mechanism for recognizing that the locked tuple is contained within the table.

The solution is to organize locks hierarchically. Before obtaining a lock on a fine-granularity item (such as a tuple), a transaction must obtain a lock on all containing items (such as a page or table). But what kind of a lock? Clearly, it should not be a read or write lock since, in that case, there would be no point in acquiring the additional fine-granularity lock, and the effective lock granularity would be coarse.

For this reason, database systems provide a new lock mode, the **intention lock**. In a system supporting tuple and table locks, for example, before a transaction can obtain a read (shared) or write (exclusive) lock on a tuple, it must obtain an appropriate intention lock on the table containing that tuple. More generally, an intention lock must be acquired on all ancestors in the hierarchy.

Intention locks are weaker than read and write locks and come in three flavors:

1. If a transaction wants to obtain a shared lock on a tuple, it must first get an **intention shared**, or IS, lock on the table. The IS lock indicates that the transaction *intends* to obtain a shared lock on some tuple within that table.

		Granted Mode				
Requested Mode		IS	IX	SIX	S	X
IS						X
IX				X	X	X
SIX		X		X	X	X
S		X		X		X
X	X	X	X	X	X	X

**FIGURE 13.3** Conflict table for intention locks. Conflicts between lock modes are denoted X.

2. If a transaction wants to obtain an exclusive lock on a tuple, it must first obtain an **intention exclusive**, or IX, lock on the table. The IX lock indicates that the transaction *intends* to obtain an exclusive lock on some tuple within the table.
3. If a transaction wants to update some tuples in the table but needs to read all of them to determine which ones to change (for example, it wants to change all tuples in which the value of a particular attribute is less than 100), it must first obtain a **shared intention exclusive**, or SIX, lock on the table. The SIX lock is a combination of a shared lock and an intention exclusive lock on the table. This allows it to read all the tuples in the table and subsequently to get exclusive locks on those it wants to update.

The conflict table for intention locks is given in Figure 13.3. It shows, for example, that after a transaction,  $T_1$ , has been granted an IX lock on a table, another transaction,  $T_2$ , will not be granted an S lock on that table (the entry in column IX and row S). To understand this, note that the shared lock allows  $T_2$  to read all tuples in the table and that this conflicts with the fact that  $T_1$  is updating one (or more) of them. On the other hand,  $T_2$  can be granted an X lock on a different tuple in the table, but it must first acquire an IX lock on the table. This does not present a problem because, as shown in the figure, IX locks do not conflict. Note that if table locking were used, both transactions would need X locks on the table and one would have to wait. This is just one example of a situation in which intention locking outperforms table locking.

**Lock escalation.** The space and time overhead of granular locking becomes excessive when a transaction accumulates too many fine-grain locks. When a transaction begins acquiring a large number of page or tuple locks on a table, it will likely continue to do so. Therefore, it is beneficial to trade in those locks for a single lock on the entire table.

This technique is known as **lock escalation**. A threshold (which the application can often control) is set in the concurrency control that limits the number of fine-grained locks a transaction,  $T$ , can obtain on a particular table. When  $T$  reaches the threshold, the concurrency control attempts to escalate the fine-grained locks for a single coarse-grained lock (in the same mode). Since the coarse-grained lock might

conflict with locks that are currently held by concurrent transactions,  $T$  might have to wait. When the coarse-grained lock is granted, the fine-grained locks are released. Note the danger of deadlock in this scheme. For example, if two transactions are acquiring X locks on pages and both reach their threshold, a deadlock results since neither can escalate their locks to a table lock.

**Serializable execution with granular locking.** In Section 13.1.5 we discussed a simple implementation of the SERIALIZABLE isolation level that uses long-duration read and write locks on tables. We pointed out that a more efficient algorithm is often used when an index is used in the query plan. With granular locking, however, it is possible to improve on the table-locking protocol even when no index is available.

- If a transaction wants to read one or more tuples in a table, it gets an S lock on the entire table.
- If a transaction wants to write one or more tuples in a table, it gets a SIX lock on the table and write locks on the tuples it wants to write.

This protocol increases concurrency compared with the table-locking protocol. The table-locking protocol requires that transactions that want to update rows obtain a write lock on the entire table, which precludes *all* concurrent access to the table. With the granular protocol, after a transaction writes some tuples, the SIX lock it has obtained allows other transaction to read all the tuples in the table other than those that were written.

To see that this protocol guarantees serializable schedules note that since the updated tuples are write locked, the only issue is to show that phantoms cannot occur. But the SIX lock (which includes a shared lock on the entire table) prevents other transactions from updating or inserting any tuples in the table, and hence no phantoms can occur.

### 13.1.7 Summary

The question of how to choose an isolation level for a particular application is far from straightforward. Serializable schedules guarantee correctness for all applications but might not meet an application's performance requirements. However, some applications execute correctly at an isolation level lower than SERIALIZABLE. For example, a transaction that prints a mailing list of depositors might not need an up-to-date database view, but it might want to be sure that the addresses it reads are not partially updated. In that case, READ COMMITTED might be an appropriate isolation level.

Commercial DBMS vendors usually support a number of options (including, but not limited to, those discussed above), and they expect application designers to select the one appropriate for their particular application.

## 13.2 Atomicity and Durability

Atomicity requires that a transaction either successfully completes (and commits) or aborts (undoing any changes it made to the database). A transaction might be aborted by the user (perhaps using a cancel button), by the system (perhaps because of a deadlock or because some database update violated a constraint check), or by itself (when, for example, it finds unexpected data). Another way the transaction might not successfully complete is if the system crashes during its execution. Crashes are a bit more complicated than user- or system-initiated aborts because the information stored in main memory is assumed to be lost when the system crashes. Hence, the transaction must be aborted and its changes rolled back using only the information stored on mass storage. Furthermore, *all* transactions active at the time of the crash must be aborted.

*Durability requires that, after a transaction commits, the changes it made to the database are not lost even if the mass storage device on which the database is stored fails.*

While the application programmer must be involved in directing how isolation is to be performed (for example, by setting isolation levels and lock escalation thresholds), the implementation of atomicity and durability is generally hidden. Still it is important for you to be knowledgeable about a few of the important features in this area, so we introduce them here.

### 13.2.1 The Write-Ahead Log

We discuss atomicity and durability in the same section because they are frequently implemented using the same mechanism—the write-ahead log.

A log is a sequence of records that describes database updates made by transactions. Records are appended to the log as the transactions execute and are never changed. The log is consulted by the system to achieve both atomicity and durability. For durability, the log is used to restore the database after a failure of the mass storage device on which the database is stored. Therefore, it is generally stored on a different mass storage device than the database itself. Typically, a log is a sequential file on disk, and it is often duplexed (with the copies stored on different devices) so that it survives any single media failure.

A common technique for achieving atomicity and durability involves the use of **update** records. When a transaction updates a database item, an update record is appended to the log. No record needs to be appended if the operation merely reads the data item. The update record describes the change made and, in particular, contains enough information to permit the system to undo that change if the transaction is later aborted.

There are several ways to undo changes. In the most common, an update record contains the **before image** of the modified database item—a physical copy of the item before the change. If the transaction aborts, the update record is used to restore the item to its original value—which is why the update record is sometimes referred to as an **undo record**. In addition to the before image, the update record

identifies the database item and the transaction that made the change—using a **transaction Id**.

If a transaction,  $T$ , is aborted, rollback using the log is straightforward. The log is scanned backwards, and, as  $T$ 's update records are encountered, the before images are written to the item named in the record, undoing the change. Since the log might be exceedingly long, it is impractical to search back to the beginning to make sure that all of  $T$ 's update records are processed. To avoid a complete backward scan, when  $T$  is initiated a **begin record** containing its transaction Id is appended to the log. The backward scan can be stopped when this record is encountered.

Rollback due to a crash is more complex than the abort of a single transaction since the system must abort all transactions that were active at the time of the crash. The hard part here is to identify these transactions using only the information contained in the log since the contents of main memory has been lost.

To do this we introduce two additional records. When a transaction commits, a **commit record** is written to the log. If it aborts, its updates are rolled back and an **abort record** is written to the log. Using these records, a backward scan can record the identity of transactions that completed prior to the crash and thus can ignore their update records as each is subsequently encountered during the backward scan. If, during the backward scan, the first record relating to  $T$  is an update record,  $T$  was active when the crash occurred and must be aborted.

Note the importance of writing a commit record to the log when  $T$  requests to commit. The commit request itself does not guarantee durability. If a crash occurs after the transaction has made the request but before the commit record is written to the log, the transaction will be aborted by the recovery procedure and will not be durable. Hence, a transaction is not actually committed until its commit record has been appended to the log on mass store.

One last issue has to be addressed. A mechanism must be provided to help the recovery process identify the transactions to be aborted. Without such a mechanism, the recovery process has no way of knowing when to stop the backward scan since a transaction that was active at the time of the crash might have logged an update record at an early point in the log and then have made no further updates. The recovery process will thus find no evidence of the transaction's existence unless it scans back to that record.

To deal with this situation, the system periodically appends a **checkpoint record** to the log that lists the currently active transactions. The recovery process must scan backward at least as far as the most recent checkpoint record. If  $T$  is named in that record and the backward scan prior to reaching the checkpoint record did not encounter a commit or abort record for  $T$ , then  $T$  was still active when the system crashed. The backward scan must continue past the checkpoint record until the begin record for  $T$  is reached. The scan terminates when all such transactions are accounted for.

An example of a log is shown in Figure 13.4. As the recovery process scans backward, it discovers that  $T_6$  and  $T_1$  were active at the time of the crash because the last records appended for them are update records. Since the first record it encounters for  $T_4$  is a commit record, it learns that  $T_4$  was not active at the time of the crash.

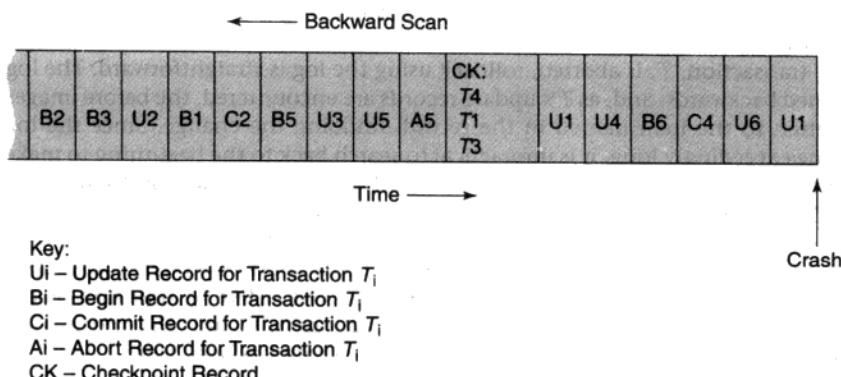


FIGURE 13.4 Log example.

When it reaches the checkpoint record, it learns that, at the time the record was written,  $T_1$ ,  $T_3$ , and  $T_4$  were active ( $T_6$  is not mentioned in the checkpoint record since it began after the checkpoint was taken). Thus, it concludes that, in addition to  $T_1$  and  $T_6$ ,  $T_3$  was active at the time of the crash (since it has seen no completion record for  $T_3$ ). No other transaction could have been active, and hence these are the transactions that must be aborted. Recovery involves processing update records for  $T_1$ ,  $T_3$ , and  $T_6$  in a backward scan in the order they are encountered until their begin records have been reached.

We have assumed that an update record for a database item,  $x$ , is written to the mass storage device containing the log at the time the new value of  $x$  is written to the mass storage device containing the database. In fact, the transfer of the new value and the update record occur in some order. Does it make a difference which occurs first?

Consider the possibility of a crash occurring at the time the operations are performed. If neither operation has completed, there is no problem. The update record does not appear in the log, but there is nothing for the recovery process to undo since  $x$  has not been updated. If the crash occurs after both operations have completed, recovery proceeds correctly, as described above. Suppose, however, that  $x$  is updated on mass store first and the crash occurs between that time and the time the update record is appended to the log. Then the recovery process has no way of rolling the transaction back and returning the database to a consistent state since the original value of  $x$  appears nowhere on mass store—an unacceptable situation.

Suppose, on the other hand, the transfers are done in the opposite order: the update record is appended first, and, when that has completed, the new value of  $x$  is transferred to the database. If the crash occurs after the log has been updated, then, on restart, the recovery process will find the update record and use it to restore  $x$ . It makes no difference whether the crash occurred after  $x$  was updated or before the update could take place. If the crash occurred after both the log and  $x$  were updated,

recovery proceeds as described earlier. If the crash occurred after the log was updated but before  $x$  could be updated, the value of  $x$  in the database and the before image in the update record are identical when the system is restarted. The recovery process uses the before image to overwrite  $x$ —which does not change its value—but the final state after recovery is correct.

Hence, the update record must always be appended to the log before the database is updated. When logging is done in this way, and it always is, the log is referred to as a **write-ahead log**.

**Performance issues.** Recovery from a crash is actually more complex than we have described. It would appear from what we have said so far that two I/O operations must be performed each time a data item is updated: one to update the database and another to append an update record to the log. Furthermore, the operations must be done in sequence. If this were the case, the performance of the system would be inadequate for all but the lightest loads. Two techniques are used to overcome this problem.

Most database systems maintain a cache—a set of page buffers in main memory used to store copies of database pages in active use (the cache is discussed in more detail in Section 12.1). Changes to the database are made in copies of database pages in the cache, and the copies need not be transferred to the database on mass store immediately. Instead, they might be kept in main memory for some time so that additional accesses to them (reads or writes) can be handled quickly. Since the probability is high that once an item in a page is accessed additional accesses to the page will follow, a considerable amount of I/O can be avoided.

Furthermore, and for similar reasons, log records are not immediately written to the log but are temporarily stored in a log buffer in main memory. To avoid a write to the log when each update record is appended, the entire buffer is written to the log when it is full. As a result, the amount of I/O to the log is reduced by a factor equal to the average number of records that fit in the buffer.

The fact that the most recent database and log information might be stored only in buffers in main memory at the time of a crash adds considerable complexity to recovery since the buffers are lost when a crash occurs. We will not describe the full recovery procedure here, except to discuss one problem. A page updated by a committed transaction,  $T$ , might still be in the cache at the time of the crash. Since durability requires that all of  $T$ 's updates survive in the database, we need a mechanism for reconstructing them on recovery. The solution lies in the update records, but there are two issues that must be dealt with: do the update records contain the necessary information, and can we be sure that they were written to mass store before the crash occurred?

The second issue is easily dealt with. If  $T$  is committed, its commit record must have been written to mass store prior to the crash. Since its update records precede it in the log, they must also be on mass store and hence available to the recovery procedure.

The first issue is more problematic. The before image in an update record is of no use in restoring a data item written by  $T$ : it has the initial value of the item, not the new value. As a result, the update record is often augmented to include both an

after image as well as a before image. The **after image** (or **redo record**) contains the new value of the item written by  $T$ . The recovery procedure can use the after image to roll the item forward and hence guarantee  $T$ 's durability.

The management of the cache and log buffer and their use in updating the log and database on mass storage must be carefully coordinated to maintain the write-ahead feature in implementing atomicity and durability.

### 13.2.2 Recovery from Mass Storage Failure

Durability requires that no changes to the database made by a committed transaction be lost. Therefore, since mass storage devices can fail, the database must be stored on different devices redundantly.

One simple approach to achieving durability is to maintain separate copies of the database on different disks (perhaps supported by different power supplies). Since the simultaneous failure of both disks is unlikely, the probability is high that at least one copy of the database will always be available. Mirrored disks implement this approach. A **mirrored disk** is a mass storage system in which, transparently to the user, whenever an application requests a disk write operation, the system simultaneously writes the same information on two different disks. Thus one disk is an exact copy, or a mirror image, of the other.

In transaction processing applications, a mirrored-disk system can achieve increased system **availability** since, if one of the mirrored disks fails, the system can continue, without slowing or stopping, using the other one. When the failed disk is replaced, the system then must resynchronize the two. By contrast, when durability is achieved using only the log (as described next), the recovery after a disk failure might take a significant period of time, during which the system is unavailable to its users.

Keep in mind, however, that even when a transaction processing system uses mirrored disks, it must still use a write-ahead log to achieve atomicity—for example, to roll back transactions after a crash.

A second approach to achieving durability involves restoring the database from the log after a disk failure. Since update records contain after images, all we have to do is play the log forward from the beginning, writing the after images in each update record as it is encountered to the database item named in the record. The problem with this approach is the time it takes since the log can be quite long.

To overcome this problem, the entire database is periodically copied, or **dumped**, to mass storage. Then, to recover from a disk failure, the most recent dump is used to initialize a new copy of the database, and then the log records appended after that dump are used to roll the copy forward to the state the database was in at the time of the failure.

An important issue in this approach is how to produce the dump. For some applications, it can be produced offline after shutting down the system: no new transactions are admitted, and all existing transactions are allowed to complete. The dump then contains a snapshot of the database resulting from the execution of all transactions that committed prior to the start of the dump. Unfortunately,

however, with many applications the system cannot be shut down, and the dump must be completed while the system is operating.

A **fuzzy dump** is a dump performed while the system is operating and transactions are executing. These transactions might later commit or abort. The algorithm for restoring the disk using a fuzzy dump must deal properly with the effects of these transactions.

## 13.3 Implementing Distributed Transactions

We have been assuming that the information accessed by a transaction is stored in a single DBMS. This is not always the case. The information supporting a large enterprise might be stored in multiple DBMSs at different sites in a network. For example, a manufacturing facility can have databases describing inventory, production, personnel, billing, and so forth. As these enterprises move to higher and higher levels of integration, transactions must access information at more than one DBMS. Thus, a single transaction initiating the assembly of a new component might allocate the parts by updating the inventory database, specify a particular employee for the job by accessing the personnel database, and create a record to describe the new activity in the production database. Systems of this type are referred to as **multidatabase systems**.

Transactions accessing multidatabase systems are often referred to as **global transactions** because they can access all the data of an enterprise. Since the databases often reside at different sites in a network, such transactions are also referred to as **distributed transactions**. Many of the considerations involved in designing global transactions do not depend on whether or not the data is distributed, and so the terms are frequently used interchangeably. Distribution across a network introduces new failure modes (e.g., lost messages, site crashes) and performance issues that do not exist when all databases are stored at the same site.

We assume that each individual database in a multidatabase exports a set of (local) transactions (perhaps as stored procedures) that can be used to access its data. For example, a bank branch might have deposit and withdraw transactions for accessing the accounts it maintains. Each such transaction can be invoked locally to reflect a purely local event (e.g., a deposit is made at the branch to a branch account) or remotely as part of a distributed transaction (e.g., money is transferred from an account at one branch to an account at another). When a transaction at a site is executed as a part of a distributed transaction, we refer to it as a **subtransaction**.

The database at each site has its own local integrity constraints relating data stored there. The multidatabase might, in addition, have global integrity constraints relating data at different sites. For example, the database at the bank's main office might contain a data item whose value is the sum of the balances of the accounts at all local branches. We assume that each distributed transaction is consistent and maintains all integrity constraints. Each subtransaction maintains the local integrity constraints at the site at which it executes, and all of the subtransactions of a distributed transaction taken together maintain the global integrity constraints.

A desirable goal in implementing distributed transactions over a multidatabase system is to ensure that they are globally atomic, isolated, and durable, as well as consistent. We have seen, however, that designers often choose to execute transactions at a single site at the weakest isolation level possible in the interest of enhancing system performance. With distributed transactions, it is sometimes necessary not only to reduce the isolation level, but also to sacrifice atomicity and isolation altogether. To better understand the underlying issues, we first present the techniques required to provide globally atomic and serializable distributed transactions.

### 13.3.1 Atomicity and Durability—The Two-Phase Commit Protocol

To make a distributed transaction,  $T$ , globally atomic, either all of  $T$ 's subtransactions must commit or all must abort. Thus, even if some subtransaction completes successfully, it cannot immediately commit because another subtransaction of  $T$  might abort. If that happens, all of  $T$ 's subtransactions must be aborted. For example, if  $T$  is a distributed transaction that transfers money between two accounts at different sites, we do not want the subtransaction that does the withdrawal at one site to commit if the subtransaction that does the deposit at the other site aborts.

The part of the transaction processing system responsible for making distributed transactions atomic is the **transaction manager**. One of its tasks is to keep track of which sites have participated in each distributed transaction. When all subtransactions of  $T$  have completed successfully,  $T$  sends a message to the transaction manager stating that it wants to commit. To ensure atomicity, the transaction manager and the database managers at which the subtransactions have executed then engage in a protocol, called a **two-phase commit protocol** [Gray 1978; Lampson and Sturgis 1979]. In describing this protocol, it is customary to call the transaction manager the **coordinator** and the database managers the **cohorts**.

The coordinator starts the first phase of the two-phase commit protocol by sending a *prepare message* to each cohort. The purpose of this message is to determine whether the cohort is willing to commit and, if so, to request that it prepare to commit by storing all of the subtransaction's update records on nonvolatile storage. If all update records are durable and the coordinator subsequently decides that the transaction should be committed, the cohort will be able to do so (since the update records contain after images), even if it subsequently crashes.

If the cohort is willing to commit, it appends a **prepared record** to the log buffer, writes the buffer to mass store, and waits until the I/O operation completes. This is referred to as a **force write** and guarantees that the prepared record, and hence all preceding records (including the transaction's update records), are durable when the cohort continues its participation in the protocol. The cohort is then said to be in the **prepared state** and can reply to the *prepare message* with a *vote message*.

The vote is **ready** if the cohort is willing to commit and **aborting** if not. Once a cohort votes ready, it cannot change its mind since the coordinator uses the vote to decide whether the transaction as a whole is to be committed. If the cohort votes **aborting**, it aborts the subtransaction immediately and exits the protocol. Phase 1 of the protocol is now complete.

The coordinator receives each cohort's vote. If all votes are ready, it decides that  $T$  can be committed globally and forces a commit record to its log. As with single-resource transactions,  $T$  is committed once its commit record is safely stored in mass storage. All update records for all cohorts are in mass storage at that time because each cohort forced a prepared record before voting. Note that we are assuming that the transaction manager and each of the local database managers have their own independent logs.

The coordinator then sends each cohort a *commit message* telling it to commit. When a cohort receives a *commit message* it forces a commit record to the database manager's log, releases locks, and sends a *done message* back to the coordinator indicating that it has completed the protocol.

It is now apparent why the coordinator's commit record must be forced. If a *commit message* were sent to a cohort before the commit record was durable, the coordinator might crash in a state in which the message had been sent, but the record was not durable. Since each cohort commits its subtransaction when the message is received, this would result in an inconsistent state: the distributed transaction is uncommitted, but the subtransaction is committed.

When the coordinator receives a *done message* from each cohort, it appends a **complete record** to the log. Phase 2 (and the complete protocol) is now complete. For a committed transaction, the coordinator makes two writes to its log, only one of which is forced. The cohort forces two records in the commit case: the prepared record and the commit record.

If the coordinator receives any aborting votes, it sends *abort messages* to each cohort that voted to commit (cohorts that voted to abort have already aborted and exited from the protocol). On receiving the message, the cohort aborts the subtransaction and writes an abort record in its log.

The sequence of messages exchanged between the application, the coordinator (transaction manager), and cohort (database manager) is shown in Figure 13.5.

For each cohort, the interval between sending a ready *vote message* to the coordinator and receiving the *commit* or *abort message* from the coordinator is called its **uncertain period** because the cohort is uncertain about the outcome of the protocol. The cohort is dependent on the coordinator during that period because it cannot commit or abort the subtransaction unilaterally since its decision might be different from the decision made by the coordinator. Locks held by the subtransaction cannot be released until the coordinator replies (since the coordinator might decide to abort and new values written by the subtransaction should not be visible in that case). This negatively impacts performance, and the cohort is said to be **blocked**. The uncertain period might be long because of communication delays.

There is also the possibility that the coordinator will crash or become unavailable because of communication failures during the uncertain period. Since the coordinator might have decided to commit or abort the transaction and then crashed before it could send *commit* or *abort messages* to all cohorts, a cohort has to remain blocked until it finds out what decision, if any, has been made. Again, this might take a long time. Because such long delays generally imply an unacceptable performance penalty, many systems abandon the protocol (and perhaps atomicity) when the uncertain period exceeds some predetermined time. They arbitrarily commit or abort

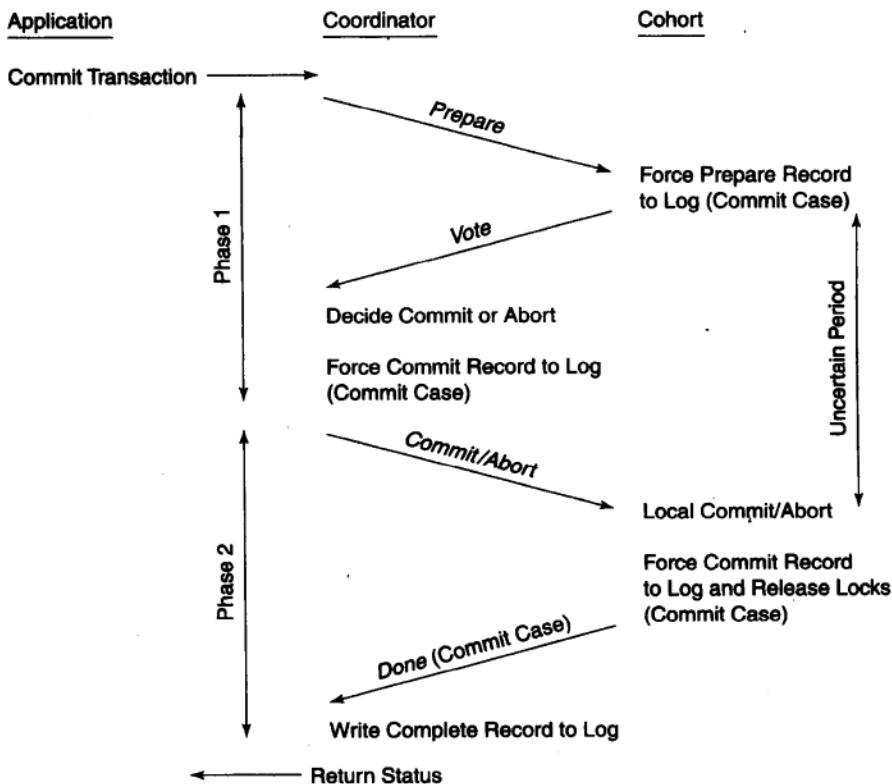


FIGURE 13.5 Exchange of messages in a two-phase commit protocol.

the local subtransaction (even though other subtransactions might have terminated in a different way) and let the system administrator clean up the mess.

In some situations, a site manager might refuse to allow the DBMS to participate in a two-phase commit protocol because of its possible negative effect on performance. In other situations, sites *cannot* participate because the DBMS is an older, legacy, system that does not support the protocol. Alternatively, client-side software (such as some versions of embedded SQL) might not support the protocol. In this circumstance, global atomicity is not realized.

### 13.3.2 Global Serializability and Deadlock

Each site in a multidatabase system might maintain its own, strict two-phase locking concurrency control, which schedules operations so that the subtransactions *at that*

*site* are serializable. Furthermore, the control might ensure that deadlocks among subtransactions *at that site* are resolved. Unfortunately, this is not sufficient to guarantee that distributed transactions are serializable and deadlock free.

**Global serializability.** To guarantee the serializability of distributed transactions, we must ensure not only that the subtransactions are serializable at each site but that there is an equivalent serial order on which all sites can agree. For example, transactions  $T_1$  and  $T_2$  might have subtransactions at sites  $A$  and  $B$ . At site  $A$ ,  $T_{1A}$  and  $T_{2A}$  might have conflicting operations and be serialized in the order  $T_{1A}, T_{2A}$ , while at site  $B$  conflicting subtransactions of the same two transactions might be serialized  $T_{2B}, T_{1B}$ . In that case, there is no equivalent serial schedule of  $T_1$  and  $T_2$  as a whole.

Surprisingly, global serializability can be achieved with no additional mechanisms beyond those we have already discussed:

If the concurrency control at each site independently uses a strict two-phase locking protocol, and the system uses a two-phase commit protocol, every global schedule is serializable (in the order in which their coordinators have committed them). [Weihl 1984]

While we do not prove that result here, it is not hard to see the basic argument that can be generalized into a proof. Suppose that sites  $A$  and  $B$  use strict two-phase locking concurrency controls, that a two-phase commit algorithm is used to ensure global atomicity, and that transactions  $T_1$  and  $T_2$  are as described above. We argue by contradiction. Suppose that the conflicts described above occur (so that the transactions are not serializable) and that both transactions commit.  $T_{1A}$  and  $T_{2A}$  conflict on some data item at site  $A$ , so  $T_{2A}$  cannot complete until  $T_{1A}$  releases the lock on that item. Since the concurrency control is strict and a two-phase commit algorithm is used,  $T_{1A}$  does not release the lock until after  $T_1$  has committed. Since,  $T_2$  cannot commit until after  $T_{2A}$  completes,  $T_1$  must commit before  $T_2$ . But if we use the same reasoning at site  $B$ , we conclude that  $T_2$  must commit before  $T_1$ . Hence, we have derived a contradiction, and it follows that both transactions cannot have committed. In fact, the conflicts we have assumed at sites  $A$  and  $B$  yield a deadlock, and one of the transactions will have to be aborted.

We have discussed circumstances under which the two-phase commit protocol is not implemented. In such situations, global transactions are not guaranteed to be globally atomic or globally serializable.

**Global deadlocks.** Distributed systems are subject to a type of deadlock that involves subtransactions at different sites. For example, a subtransaction of  $T_1$  at site  $A$  might be waiting for a lock held by a subtransaction of  $T_2$  at  $A$ , while a subtransaction of  $T_2$  at site  $B$  might be waiting for a lock held by a subtransaction of  $T_1$  at  $B$ . Since all subtransactions of a global transaction must commit at the same time,  $T_1$  and  $T_2$  are in a distributed deadlock—they will both wait forever. Unfortunately, the deadlock cannot be detected at any single site: a distributed algorithm must be

used. Fortunately, the techniques developed for deadlock detection at a single site can be generalized to detect a global deadlock.

### 13.3.3 Replication

In a distributed system, replicas of a data item can be stored at different sites in a network. This has two potential advantages. It can reduce the time it takes to access the item since the nearest (perhaps even local) replica can be used. It can also improve the availability of the item in case of failures since, if a site containing a replica crashes, the item can still be accessed using a different replica at another site.

The price that must be paid for these advantages, in addition to the added complexity and storage involved, is the cost of maintaining **mutual consistency**: all the replicas of an item should have the same value. Mutual consistency comes in two flavors. With **strong mutual consistency**, every committed replica of an item always has the same value as every other committed replica. Unfortunately, performance considerations often make this goal impractical to achieve, and so most replica controls maintain a more modest goal, **weak mutual consistency**: all committed replicas of an item *eventually* have the same value, although at any particular time some may have different values. A variety of algorithms are used to maintain these goals.

In most implementations of replication, the individual transactions are unaware that replicas exist. A transaction simply requests to access a data item, and the system performs the access on one or more replicas in accordance with the replication algorithm that it implements. The system knows which items are replicated and at what sites the replicas are stored. The portion of the system responsible for implementing replication is called the **replica control**.

The most common form of replication uses a **read-one/write-all** algorithm. When a transaction requests to read an item, the system fetches its value from the nearest replica; when a transaction requests to update an item, the system updates all replicas. Read-one/write-all replication has the potential for improving the speed with which a read request can be satisfied compared with nonreplicated systems since with no replication, a read request might require communication with a distant DBMS. The performance of write requests, however, might be worse since all replicas must be updated. Hence, read-one/write-all replication has a potential performance benefit in applications in which reading occurs substantially more frequently than writing.

Read-one/write-all systems can be characterized as synchronous update or asynchronous update.

- **Synchronous-update systems.** When a transaction updates an item, all replicas are locked and updated before the transaction commits. Updates to replicas are thus treated in the same way as updates to any other data item, and strong mutual consistency is enforced since locks are not released until all replicas are mutually consistent. In addition to performance, availability is a problem for writes since if a replica site is down, a write operation on the item cannot be

**FIGURE 13.6** Schedule illustrating the possibility of inconsistent views with asynchronous-update replication.  $T_1$  updates  $x$  and  $y$  at sites  $A$  and  $B$ .  $T_{nu}$  propagates the updates after  $T_1$  commits. Because propagation is asynchronous,  $T_2$  sees the new value of  $y$  but the old value of  $x$ .

$T_1:$	$w(x_A)$	$w(y_B)$	<i>commit</i>					
$T_2:$				$r(x_C)$	$r(y_B)$	<i>commit</i>		
$T_{nu}:$							$w(x_C)$	$w(x_B)$

---

completed. However, synchronous-update systems based on two-phase locking and two-phase commit protocols produce globally serializable schedules.

- **Asynchronous-update systems.** Only one replica is updated before the transaction commits. The other replicas are updated after the transaction commits by another transaction that is triggered by the commit operation or that executes periodically at fixed intervals. Hence, even in systems based on two-phase locking and two-phase commit protocols, schedules might not be globally serializable. For example, transaction  $T_1$  might write a new value to data items  $x$  and  $y$ . The replica control might choose to update the copy of  $x$  at site  $A$  and the copy of  $y$  at site  $B$  before  $T_1$  commits. Transaction  $T_2$  might be concurrently reading  $x$  and  $y$ , and the replica control might return the value of the replica of  $x$  at site  $C$  (an old value) and the value of  $y$  at site  $B$  (a new value). A transaction to propagate  $T_1$ 's updates,  $T_{nu}$ , is executed when  $T_1$  commits. The schedule is shown in Figure 13.6.

Asynchronous-update systems come in two varieties.

- **Group replication.** A transaction can lock and update any replica (presumably the nearest one). After the transaction commits, the update is propagated asynchronously to the other replicas. Unfortunately, without further restrictions, even weak mutual consistency is not guaranteed by this protocol since transactions executing concurrently can update different replicas of the same item and although the new values produced by these transactions will ultimately arrive at all replicas, they might arrive at different replicas in different orders. If each replica site simply applies the updates in the order in which they arrive, the replicas might not converge to a common value. Weak mutual consistency can be enforced by attaching time stamps to each update and to each replica (the time stamp of a replica is the time stamp of the latest update that has been applied to it) and only apply an update if its time stamp is greater than the time stamp of the replica. With this scheme, some updates might be discarded.

Situations such as these are called **conflicts**. Although the time stamp algorithm guarantees weak mutual consistency, it does not guarantee that anomalies are eliminated. Commercial replication systems offer a variety of ad hoc conflict resolution strategies, including: “oldest update wins,” “youngest update wins,” “update from the highest-priority site wins,” and

"user provides a procedure for conflict resolution." The application designer can select the strategy most suitable for the application.

- **Primary copy replication.** A unique replica of each data item is designated the **primary copy**. All other replicas are designated **secondary copies**. Transactions requesting to update an item must lock and update its primary copy. When a transaction commits, the update it has made to the primary copy is propagated to the other replicas. By filtering all updates through the primary, write conflicts will be detected. Weak mutual consistency is ensured if propagation messages are delivered to all replicas in the order sent since all replicas will go through the same sequence of changes. Read requests are satisfied in the conventional way by accessing the nearest replica.

Since asynchronous update produces greater transaction throughput than does synchronous update, it is the most widely used form of replication. However, the designer should be aware that asynchronous-update systems (even primary copy systems) can produce nonserializable schedules and hence can produce incorrect results.

### 13.3.4 Summary

Many aspects of distributed transaction processing systems are surprisingly simple. If each site uses a strict two-phase locking concurrency control, a two-phase commit protocol is used to synchronize cohorts at commit time, and synchronous-update replication is used, then distributed transactions will be globally atomic and serializable. Global deadlocks can be resolved with algorithms that are generalizations of single-site deadlock detection and prevention algorithms.

Frequently these conditions do not hold. The application at some sites might use one of the lower isolation levels, sites might not participate in a two-phase commit protocol, and/or asynchronous replication might be used. In such situations, distributed transactions are not guaranteed to be serializable. Nevertheless, systems might have to be designed under these constraints, and the application designer must carefully assess the implications of nonserializable schedules on the correctness of the database and the ultimate utility of the application.

## BIBLIOGRAPHIC NOTES

A comprehensive discussion of issues related to the implementation of distributed transactions can be found in [Gray and Reuter 1993]. [Ceri and Pelagatti 1984] is more theoretical in its orientation and describes the algorithms underlying the implementation. Two-phase locking was introduced in [Eswaran et al. 1976]. Isolation levels are discussed in [Gray et al. 1976]. The two-phase commit protocol was introduced in [Gray 1978; Lampson and Sturgis 1979]. See [Weihl 1984] for a proof that the two-phase commit protocol together with two-phase locking local concurrency controls guarantees global serializability. One of the first discussions on logging and

recovery technology can be found in [Gray 1978]. Excellent summaries of the technology are in [Haerder and Reuter 1983; Bernstein and Newcomer 1997; Gray and Reuter 1993]. Primary copy replication was introduced in [Stonebraker 1979].

## EXERCISES

- 13.1 State which of the following schedules are serializable.
- $r_1(x) r_2(y) r_1(z) r_3(z) r_2(x) r_1(y)$
  - $r_1(x) w_2(y) r_1(z) r_3(z) w_2(x) r_1(y)$
  - $r_1(x) w_2(y) r_1(z) r_3(z) w_1(x) r_2(y)$
  - $r_1(x) r_2(y) r_1(z) r_3(z) w_1(x) w_2(y)$
  - $w_1(x) r_2(y) r_1(z) r_3(z) r_1(x) w_2(y)$
- 13.2 In the Student Registration System, give an example of a schedule in which a deadlock occurs.
- 13.3 Give an example of a schedule that might be produced by a nonstrict two-phase locking concurrency control that is serializable but not in commit order.
- 13.4 Give an example of a transaction processing system (other than a banking system) that you have interacted with, for which you had an intuitive expectation that the serial order was the commit order.
- 13.5 Suppose that the transaction processing system of your university contains a table in which there is one tuple for each current student.
- Estimate how much disk storage is required to store this table.
  - Give examples of transactions in the student registration system that have to lock this entire table if a table locking concurrency control is used.
- 13.6 Give an example of a schedule at the READ COMMITTED isolation level in which a lost update occurs.
- 13.7 Give examples of schedules at the REPEATABLE READ isolation level in which a phantom is inserted after a SELECT statement is executed and
- The resulting schedule is nonserializable and incorrect.
  - The resulting schedule is serializable and hence correct.
- 13.8 Give examples of schedules at the SNAPSHOT isolation that are
- Serializable and hence correct.
  - Nonserializable and incorrect.
- 13.9 Give examples of schedules that would be accepted at
- SNAPSHOT isolation but not REPEATABLE READ.
  - SERIALIZABLE but not SNAPSHOT isolation. (*Hint:*  $T_2$  performs a write after  $T_1$  has committed.)
- 13.10 Give an example of a schedule of two transactions in which a two-phase locking concurrency control
- makes one of the transactions wait, but a control implementing SNAPSHOT isolation aborts one of the transactions.

- b. aborts one of the transactions (because of a deadlock), but a control implementing SNAPSHOT isolation allows both transactions to commit.
- 13.11 A particular read-only transaction reads data entered into the database during the previous month and uses it to prepare a report. What is the weakest isolation level at which this transaction can execute? Explain.
- 13.12 What intention locks must be obtained by a read operation in a transaction executing at REPEATABLE READ when the locking implementation given in Section 13.1.5 is used?
- 13.13 Explain how the commit of a transaction is implemented within the logging system.
- 13.14 Explain why the write-ahead feature of a write-ahead log is needed.
- 13.15 Explain why a cohort in the two-phase commit protocol cannot release locks acquired by the subtransaction until its uncertain period terminates.
- 13.16 Two distributed transactions execute at the same two sites. Each site uses a strict two-phase locking concurrency control, and the entire system uses a two-phase commit protocol. Give a schedule for the execution of these transactions in which the commit order is different at each site but the global schedule is serializable.
- 13.17 Give an example of an incorrect schedule that might be produced by an asynchronous-update replication system.
- 13.18 Explain how to implement synchronous-update replication using triggers.

## PART FOUR

---

# Software Engineering Issues and Documentation

This part has two purposes:

1. To describe in some detail the Student Registration System, which is used as a case study throughout the book.
2. To describe the software engineering concepts that are needed to actually implement such a system. We illustrate how the Unified Modeling Language (UML) can be used in this process.

Chapter 14 presents a Requirements Document for the Student Registration System and describes how those requirements can be analyzed and expanded into a Specification Document.

Chapter 15 discusses design, coding, testing, and project management. It completes the design of the database schema for the Student Registration System (which was started in Section 4.8) and presents the design of one of the transactions.



# 14

## Requirements and Specifications

The implementation of the Student Registration System is proceeding on schedule. A team consisting of faculty, students, and representatives of the registrar has met several times with an analyst and has refined the informal Statement of Objectives given in Section 2.1 into a formal Requirements Document, which we reproduce in Section 14.2. Now it is time to complete the remaining parts of the project. We have already jumped ahead and implemented the database design part of the project in Chapters 4 and 6. In this chapter and in Chapter 15 we review the entire software engineering process in more detail. Our main interest is in the software engineering issues involved in the design and implementation of transactions and databases.

### 14.1 Software Engineering Methodology

The implementation of a transaction processing system is a significant engineering endeavor. The project must complete on time and on budget, and, when operational, the system must meet its requirements and operate efficiently and reliably. The documentation and coding for the project must be such that the system can be maintained and enhanced over its lifetime. Most important, it must meet the needs of its users.

Many years' experience with both successful and unsuccessful software projects has given rise to a number of procedures and methodologies generally agreed to be "good engineering practice." Many books and entire courses are devoted to software engineering. Here we sketch one approach and apply it to the Student Registration System.

In particular we talk about what software engineers call the **Waterfall model**, in which the project is divided into separate phases: requirements, specification, design, code, and test. And after the system is delivered, there is a final phase: maintenance. Similar phases exist in the development of most large engineering systems—for example a commercial airliner. In this chapter we talk about the requirements and specification phases, and in Chapter 15 we discuss the remaining phases.

**Requirements Document.** Projects usually begin with an informal Statement of Objectives as given at the beginning of Section 2.1. The next step is for the customers and users of the system, perhaps with the help of a system analyst, to expand these objectives into a formal **Requirements Document** for the system as given in Section 14.2. The Requirements Document describes in some detail what the system is supposed to do, not how it will do it. In many contexts, the Requirements Document is a Request for Proposals (RFP) to the implementors, describing what the customer wants them to build.

**Specification Document.** The implementation group analyzes the Requirements Document in detail and produces a **Specification Document**, which is an expanded version of the Requirements Document that describes in still more detail what the system will do. In many contexts, the Specification Document is a contract proposal, describing exactly what the implementation group intends to build. The description is so precise that the User Manual and the Specification Document can be written and published at the same time. The following examples illustrate the different levels of detail in the requirements and specification documents.

- In the Requirements Document, the set of user interactions with the system is listed, together with what each interaction is intended to do. In the Specification Document, the forms associated with each interaction are specified, together with exactly what happens when each button is pressed and each menu item is accessed.
- The Requirements Document lists the information that must be contained in the system. The Specification Document includes the domains of all items of information.

### 14.1.1 UML Use Cases

The Requirements Document specifies what the system is supposed to do from the user's point of view. Specifically, it specifies the user interactions with the system.

A common way to describe user interactions with the system is as a set of *use cases*. A software engineering text might define a **use case** as a sequence of actions that are performed to produce an observable result of benefit to one or more users (called **actors**). For example, in the Requirements Document, we have a use case called *Registration* in which a student registers for a course. Analysts often develop use cases by asking potential users of the system, "How do you accomplish such and such?". Thus we might ask a student, "How do you register for a course?" and then ask the registrar, "What are some situations in which the registration should not succeed?". Their responses might be the basis for developing the *Registration* use case, in which a student is the actor. Such a use case might be described as follows:

#### **Registration.**

*Purpose.* Register a student in a course to be taught next semester.

*Actor.* A student.

*Input.* A course number.

*Result.* The student is registered for the course, and an appropriate message is displayed.

*Exception.* The registration shall not be successful for any of the following reasons, which shall be contained in the output.

- A. There exists a prerequisite course that the student is not currently enrolled in or has not completed with a grade of at least C.
- B. And so on. (The complete list of exceptions is in the Requirements Document.)

Different software engineering texts use different formats for describing use cases. We use a particular format that seems appropriate for this application. Other formats might include *Preconditions*—what must be true before the use case starts; for example a precondition for the Registration use case might be that the Authentication use case for that actor has been successful; or *Actions*—for example, the student first does this, then that happens, then the student does this, etc.

Note that we describe user interactions using use cases rather than transactions because at this stage we do not yet know how many transactions will be required to implement each use case—that is part of the design.

Use cases are part of the *Unified Modeling Language* (or UML). The UML is a graphical language for modeling the static and dynamic behavior of a system. It provides a standard set of diagrams, each of which models a different aspect of the system's behavior. Because these diagrams are graphical, they are particularly appropriate for communicating information between the customer and the implementation group and between different members of the implementation group. Also, because the UML has become a widely adopted standard, UML diagrams can be used to communicate with other people not directly involved in the project, perhaps consultants invited in for a project review.

UML diagrams are one of the sources on which the requirements, specification, and design documents are based. In particular they are used to capture and display certain key aspects of the system behavior needed for these documents. Use cases and, as we shall see, use case diagrams are a part of the UML that deals with formulating requirements. In Section 14.4, we discuss the use of UML sequence diagrams for formulating specifications. In Chapter 4 we discuss the use of UML class diagrams for database design, and in Section 15.1.2 we discuss the use of UML state diagrams for describing the dynamic behavior of objects as part of the design process.

Although use cases are not inherently graphic in nature, the UML provides a graphic way to display the use cases in an application: the **use case diagram**. Figure 14.1 shows a use case diagram for all the use cases in the Student Registration System (as described in Section 14.2). Each actor in the use case is represented as a labeled stick figure, and each use case is represented as a labeled oval. Arrows connect each actor with the use cases in which the actor participates. Such a use case diagram might be developed while interviewing various potential users of the system and then discussed with these users to ensure that the set of use cases is complete and that the final system will satisfy their needs and meet their goals.

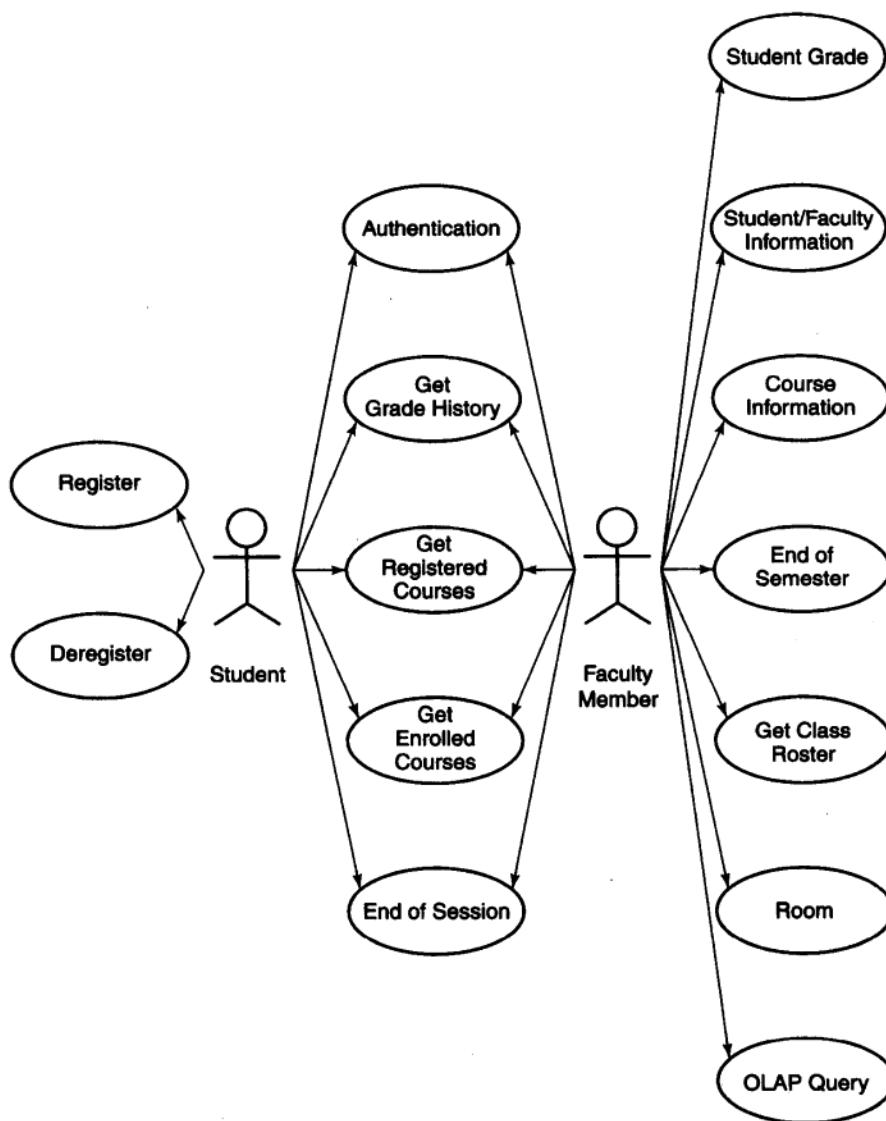


FIGURE 14.1 A use case diagram for the Student Registration System.

for the system. Use case diagrams provide a visually clear model for displaying the requirements of an application and might be included as part of the Requirements Document.

## 14.2 The Requirements Document for the Student Registration System

### I. Introduction

The objectives of the Student Registration System are to allow students and faculty (as appropriate) to

- A. Authenticate themselves as users of the system
- B. Register and deregister for courses (offered for the next semester)
- C. Obtain reports on a particular student's status
- D. Maintain information about students and courses
- E. Enter final grades for courses that a student has completed

In this document, the term "enrolled" refers to courses a student is currently taking and the terms "registered" and "deregistered" refer to courses to be taken or dropped by the student in the following semester.

### II. Related Documents

- A. *Statement of Objectives* of the Student Registration System (including date and version number)
- B. This university's *Undergraduate Bulletin* (including date)

### III. Information to Be Contained in the System

The information to be stored in the system includes four major categories of data: personal information about students and faculty members, academic records of students, teaching records of faculty members, and information about courses and course offerings. Information about classrooms and other auxiliary data is also stored in the system.

- A. **Personal records.** The system shall contain a name, an Id number, and a password for each student and faculty member allowed to use the system.<sup>1</sup> The password and the Id authenticates users. Id numbers are unique. It is assumed

<sup>1</sup> Note that the requirements are numbered so that they can be referred to in later documents, such as the Test Plan, which must test that the system meets every one of its requirements. Also, requirements that are stated using words such as "shall" and "must" are mandatory. Words such as "should" and "can" do not connote a mandatory requirement and should be avoided unless the requirement is optional. For example, in one of the earliest recorded Requirements Documents (even then the requirements were numbered), the commandment is "Thou shalt not kill," not "Thou should not kill."

## CHAPTER 14 Requirements and Specifications

that at least one faculty member has been initialized as a valid user at startup time.

- B. **Academic records.** The system shall contain the academic record of each student.
  - 1. Each course the student has completed, the semester the student took the course, and the grade the student received (all grades are in the set {A, B, C, D, F, I})
  - 2. Each course for which the student is enrolled this semester
  - 3. Each course for which the student has registered for next semester
- C. **Course information.** The system shall contain information about the courses offered, and for each course the system shall contain
  - 1. The course name, the course number (must be unique), the department offering the course, the textbook, and the credit hours
  - 2. Whether the course is offered in spring, fall, or both
  - 3. The prerequisite courses (there can be an arbitrary number of prerequisites for each course)
  - 4. The maximum allowed enrollment, the number of students who are enrolled (unspecified if the course is not offered this semester), and the number of students who have registered (unspecified if the course is not offered next semester)
  - 5. If the course is offered this semester, the days and times at which it is offered; if the course is offered next semester, the days and times at which it will be offered. The possible values shall be selected from a fixed list of weekly slots (e.g., MWF10).
  - 6. The Id of the instructor teaching the course this semester and next semester (the Id is unspecified if the course is not offered in the specified semester; it must be specified before the start of the semester in which the course is offered)
  - 7. The classroom assignment of the course for this semester and next semester (the classroom assignment is unspecified if the course is not offered in the specified semester; it must be specified before the start of the semester in which the course is offered)

All course information shall be consistent with the *Undergraduate Bulletin*.

- D. **Teaching information.** The system shall contain a record of all courses that have been taught, including the semester in which they were taught and the Id of the instructor.
- E. **Classroom information.** The system shall contain a list of classroom identifiers and the corresponding number of seats. A classroom identifier is a unique three-digit integer.
- F. **Auxiliary information.** The system shall contain the identity of the current semester (e.g., F2004).

#### IV. Integrity Constraints

The database shall satisfy the following integrity constraints.

- A. Id numbers are unique.
- B. If in item III.B.2 (or III.B.3) a student is listed as enrolled (or registered) for a course, that course must be indicated in item III.C.2 as offered this semester (or next semester).
- C. In item III.C.4, the number of students registered or enrolled in a course cannot be larger than the maximum enrollment.
- D. The count of students enrolled (or registered) in a course in item III.B.2 (or III.B.3) must equal the current enrollment (or registration) indicated in item III.C.4.
- E. An instructor cannot be assigned to two courses taught at the same time in the same semester.
- F. Two courses cannot be taught in the same room at the same time in a given semester.
- G. If a student is enrolled in a course, the corresponding record must indicate that the student has completed all prerequisite courses with a grade of at least C.
- H. A student cannot be registered (or enrolled) in two courses taught at the same hour.
- I. A student cannot be registered for more than 20 credits in a given semester.
- J. The room assigned to a course must have at least as many seats as the maximum allowed enrollment for the course.
- K. Once a letter grade of A, B, C, D, or F has been assigned for a course, that grade cannot later be changed to an I.<sup>2</sup>

#### V. Use Cases

Use cases are performed during *sessions*. A session starts when a user executes an Authentication use case and ends when a user executes an End of Session use case. During a session, a user can execute one or more use cases. The use case diagram for these use cases is shown in Figure 14.1 (which is assumed to be part of the Requirements Document).

##### A. Authentication.

*Purpose.* Identify the actor and determine whether she is a student or faculty member. Subsequent use cases in the same session depend on this distinction.

*Actor.* A student or a faculty member.

*Input.* The actor's Id number and password.

<sup>2</sup> This is an example of a *dynamic* integrity constraint, which limits the allowable changes to the state of a database, in contrast to a *static* integrity constraint, which limits the allowable states of the database. We discuss dynamic integrity constraints in Section 3.2.2.

## CHAPTER 14 Requirements and Specifications

**Result.** The actor is authenticated and can perform other use cases she is authorized to perform.

**Exception.** If the actor enters an incorrect Id or password, authentication does not occur and the actor is given another chance to enter an Id and password.

### B. Registration.

**Purpose.** Register a student in a course to be taught next semester.

**Actor.** A student.

**Input.** A course number.

**Result.** The student is registered for the course, and an appropriate message is displayed.

**Exception.** The registration shall not be successful for any of the following reasons, which shall be contained in the output:

1. There exists a prerequisite course that the student is not currently enrolled in or has not completed with a grade of at least C.
2. The number of students registered for the course would exceed the allowed maximum.
3. The initiator of the use case is not a student.
4. The student has registered for another course scheduled at the same time.
5. The student is enrolled in the course or has taken the course and has received a grade of C or better.
6. The course is not offered next semester.
7. The student is already registered for the course.
8. The student would be taking more than 20 credits if the registration were to succeed.

### C. Deregistration.

**Purpose.** Deregister the student from a course to be taught next semester for which that student previously registered.

**Actor.** A student.

**Input.** A course number.

**Result.** The student is no longer registered for the course, and an appropriate message is displayed.

**Exception.** If the student is not registered for the course, the deregistration shall be unsuccessful.

### D. Get Grade History.

**Purpose.** Produce a report describing the grade history of a student for each semester in which he has completed courses.

**Actor.** A student or a faculty member.

**Input.** A student Id number. If a student is executing the use case, the number need not be entered because the student can request only his or her own report, and the Id of the invoker has been determined as part of authentication.

**Result.** The report shall include

1. Current semester
2. Student name and Id number

## 14.2 The Requirements Document for the Student Registration System

3. List of courses completed with grade and instructor grouped by semester
  4. Semester GPA and total number of credits for each semester in which the student has completed courses
  5. Cumulative GPA and total number of credits of all courses completed so far
- Exception.* If a faculty member is executing the use case and an invalid student Id is input, no report shall be produced.

### E. Get Registered Courses.

*Purpose.* Produce a report listing the courses for which a particular student has registered for the next semester.

*Actor.* A student or a faculty member.

*Input.* A student Id number. If a student is executing the use case, the number need not be entered because the student can request only his or her own report, and the Id of the invoker has been determined as part of authentication.

*Result.* The report shall include

1. Student's name and Id number
2. Course number and credit hours
3. Time schedule for every course
4. Classroom assignment (if available)
5. Instructor (if available)

*Exception.* If a faculty member is executing the use case and an invalid student Id is input, no report shall be produced.

### F. Get Enrolled Courses.

*Purpose.* Produce a report listing the courses in which a particular student is enrolled this semester.

*Actor.* A student or a faculty member.

*Input.* A student Id number. If a student is executing the use case, the number need not be entered because the student can request only his or her own report, and the Id of the invoker has been determined as part of authentication.

*Result.* The report shall include

1. Student's name and Id number
2. Course number and credit hours
3. Time schedule for every course
4. Classroom assignment
5. Instructor

*Exception.* If a faculty member is executing the use case and an invalid student Id is input, no report shall be produced.

### G. Student Grade.

*Purpose.* Assign or change a grade for a course a student has completed.

*Actor.* The faculty member who taught the course.

*Input.* A student Id number, a course number, a semester, and a grade.

## **CHAPTER 14 Requirements and Specifications**

### *Result.*

1. The student shall no longer be shown as enrolled in that course, but shall be shown as having completed that course.
2. If the course is a prerequisite for some course in the following semester for which the student is currently registered and if the grade is less than C, the student shall be deregistered from that course.

*Exception.* The grade shall not be assigned or changed if

1. The invoker is not the faculty member who taught the course in the semester indicated.
2. The student Id number is invalid.
3. The student is not currently enrolled in the course or did not take the course in a previous semester.
4. The use case would change a grade (A, B, C, D, F) to an I.

## **H. Student/Faculty Information.**

*Purpose.* Add, delete, or edit an entry specified in item III.A.

*Actor.* a faculty member.<sup>3</sup>

*Input.* If an entry is to be added, the name, Id number, faculty/student status, and password must be supplied. If an entry is to be deleted or edited, the Id number must be provided as well as any fields to be changed.

*Result.* The specified information is added, edited, or deleted.

## **I. Course Information.**

*Purpose.* Display or edit the information describing an existing course (item III.C) or enter information describing a new course.

*Actor.* A student or a faculty member.

*Input.* A course number.

*Result.* The requested information is displayed and can be edited. A faculty member can change any characteristic of a course but cannot delete the course. Students shall be allowed only to display (not to enter or edit) information about a course.

*Exception.* If the edited course information would violate any integrity constraint, no update shall take place.

## **J. End of Semester.**

*Purpose.* Update the database to reflect the end of the semester.

*Actor.* A faculty member.

*Result.*

1. The identity of the current semester, as specified in item III.F, shall be advanced.
2. For each student, an I grade shall be assigned for all courses in which that student is currently enrolled and for which no grade has yet been assigned.

<sup>3</sup> In a real system, this information would be controlled by a database administrator using a special set of transactions. In this project, we assume for simplicity that the database has been initialized with at least one faculty member's name, Id, and password.

3. Each student shall be indicated as enrolled in those courses for which the database previously indicated that the student was registered.
4. For each course listed in item III.C.4, the number of students enrolled shall be set equal to the number registered, and the number registered shall be set to 0.

*Exception.* If a course is scheduled to be taught next semester to which an instructor or classroom has not yet been assigned, the semester shall not be updated, no database changes shall be made, and an appropriate message shall be displayed.

**K. Get Class Roster.**

*Purpose.* Produce a list of the names and Id numbers of students currently enrolled in or registered for a course.

*Actor.* A faculty member.

*Input.* A course number and an indication of whether an enrollment or a registration list is requested.

*Result.* The requested class roster is displayed.

*Exception.* If an enrollment list for a course not currently being taught or a registration list for a course not to be taught next semester is requested, no display is returned.

**L. Room.**

*Purpose.* Display the size (i.e., number of seats) of an existing classroom (item III.E) or enter the identifier and size of a new classroom.

*Actor.* A faculty member.

*Input.* A classroom identifier and the number of seats (if a new classroom is to be entered) or just the identifier (if the size of an existing classroom is requested).

*Result.* The requested information is displayed or the identity and size of the new classroom is stored in the database.

*Exception.* If an existing classroom size is requested and the specified classroom identifier is incorrect, an appropriate error message is displayed.

**M. OLAP Query.**

*Purpose.* Allow the user to input an arbitrary query from the screen.

*Actor.* A faculty member (who, it is assumed, knows the database schema).

*Input.* A query in the form of a single SELECT statement.

*Result.* The table produced by the query is output on the screen with attribute names (where possible) heading each column.

*Exception.* If a statement other than a SELECT is input or if the statement is incorrect, an appropriate error message is returned.

**N. End of Session.**

*Purpose.* End the session.

*Actor.* A student or a faculty member.

*Input.* The actor clicks the logout button.

*Result.* Any subsequent use cases with the system require a new authentication.

## **VI. System Issues**

- A. The system shall be implemented as a client/server system. The client computer shall be a PC on which the application programs will execute.
- B. The user interface shall be graphical and easy to use by students and faculty with little or no training.
- C. The database can be any SQL database that executes on an available server computer and provides a transactional interface (in other words, it can perform the commit and abort operations).

## **VII. Deliverables**

- A. A Specification Document that describes in detail the sequence of events (input/output) that occurs for each use case, including
  1. The forms and controls to be used
  2. The effect of using each control on each form, including any new forms that are displayed as a result of each possible action
  3. The errors for which the system checks and the error messages that are output
  4. Integrity constraints
- B. A Design Document that describes in detail
  1. An entity-relationship (E-R) diagram that describes the system
  2. The declaration of all database elements (including tables, domains, and assertions)
  3. The decomposition of each use case into transactions and procedures
  4. The behavior of each transaction and procedure
- C. A Test Plan describing how the system will be tested, including how each of the numbered requirements and specifications will be tested
- D. A demonstration of the completed system (including running the tests in the Test Plan)
- E. Fully documented code for the system
- F. A User Manual with separate sections for student and faculty
- G. Version 2 of the Specification Document, the Design Document, and the Test Plan, describing the as-built system

## **14.3 Requirements Analysis—New Issues**

The next phase of the project is to analyze the Requirements Document and produce a formal Specification Document. Experience has shown that, no matter how carefully the Requirements Document is written, when the implementation team analyzes the requirements in order to prepare the Specification Document, a number

of new issues will be identified. Parts of the Requirements Document will be found to be inconsistent or incomplete, and questions will be raised about the desired behavior of the system in certain previously unforeseen situations. The implementation team customarily presents these issues to the customer, who resolves them in a written document. The resolved issues then become part of a revised version of the Requirements Document and part of the initial version of the Specification Document. This entire scenario underscores the difficulties in precisely specifying the desired behavior of a proposed system.

When the Requirements Document given in Section 14.2 was analyzed by our local implementation team, a number of issues were identified. Below we present some of these together with their resolution. Your local implementation team is likely to discover other issues.

**Issue 1.** What if, during the Course Information use case, an attempt is made to add a new prerequisite for a course such that the prerequisites form a cycle? For example, course A is a prerequisite for course B, B is a prerequisite for course C, and C is a prerequisite for A. In other words, course A is a prerequisite for itself.

*Resolution.* A new database integrity constraint must be added to deal with this situation: there must not be a cycle of prerequisites. Any transaction that implements the Course Information use case shall check for this condition, and, if it exists, the prerequisite shall not be added and an appropriate message shall be presented to the user. (The check for circularity in the general case is not simple. We might, however, require that the prerequisite for a course have a lower number than the course itself. Such a requirement eliminates the possibility of circularity.)

**Issue 2.** What if, during the Course Information use case, an attempt is made to add a new prerequisite for a course, and a student who does not have that prerequisite is already registered for that course?

*Resolution.* A new prerequisite for a course does not apply to the offering of the course (if any) in the following semester.

**Issue 3.** What if, during the Course Information use case, the maximum number of students allowed in a course is reduced to a value that is less than the number of students who have already registered for the course?

*Resolution.* Room rescheduling is a fact of academic life, so this use case must be allowed. However, the appropriate number of students must be deregistered from the course to bring the total number of registered students to the new maximum. The students shall be deregistered in the reverse order that they were registered. All deregistered students shall be notified in writing.

**Issue 4.** What if, during the Course Information use case, an attempt is made to change the day and/or time a course is offered?

*Resolution.* A change in day and/or time does not apply to the offering of the course (if any) in the following semester.

**Issue 5.** What if, during the Course Information use case, a course is canceled?

*Resolution.* Cancellation applies to the next semester. Students registered for the next semester shall be deregistered and notified in writing.

**Issue 6.** What if, during a Student/Faculty Information use case, an attempt is made to change a student's Id number?

*Resolution.* An Id number (in contrast to a name or password) is permanently associated with an individual, so an attempt to change it makes no sense, except if it was originally entered in error. As a result, a change in Id number is allowed only if there is no other information relevant to the student in the system.

**Issue 7.** Several of the use cases, such as Get Grade History, Get Registered Courses, and Get Enrolled Courses, produce reports that describe the state of the database at a particular instant. Should those reports include the date and time they were produced?

*Resolution.* Yes, all such reports shall include the date and time.

**Issue 8.** Should the information in items III.D and III.F contain the year as well as the semester?

*Resolution.* Yes, and the End of Semester use case shall appropriately update the year. This information shall be initialized at startup time.

**Issue 9.** How many digits should be used to indicate the year in the system?

*Resolution.* Four.

## **14.4 Specifying the Student Registration System**

A Specification Document contains a complete description of what the system is supposed to do from the viewpoint of its end users—it is an expanded version of the Requirements Document. For a transaction processing system, the Specification Document should include

- The integrity constraints of the enterprise
- A complete description of the user interface
  - A picture of every form with every control specified
  - A description of what happens when each control is used, including
    - What application procedure is executed
    - What changes occur in the form or what new form is displayed
    - What error situations can occur and what happens in each such situation
- A description of each use case, including
  - The information input by the user and what events cause the use case to be executed

- A textual description of what the use case does (for example, “the student is registered for the course”)
- A list of conditions under which the use case succeeds or fails, and what happens in each case

The Specification Document might also contain other information related to project planning (such as schedules, milestones, deliverables, cost information, etc.), information related to system issues (such as software and hardware on which the system must run), and any time or memory constraints. The Table of Contents for the Specification Document for the Student Registration System has the following sections:

- I. Introduction
- II. Related Documents
- III. Forms and Use Cases
- IV. Project Plans
  - A. Milestones
  - B. Deliverables

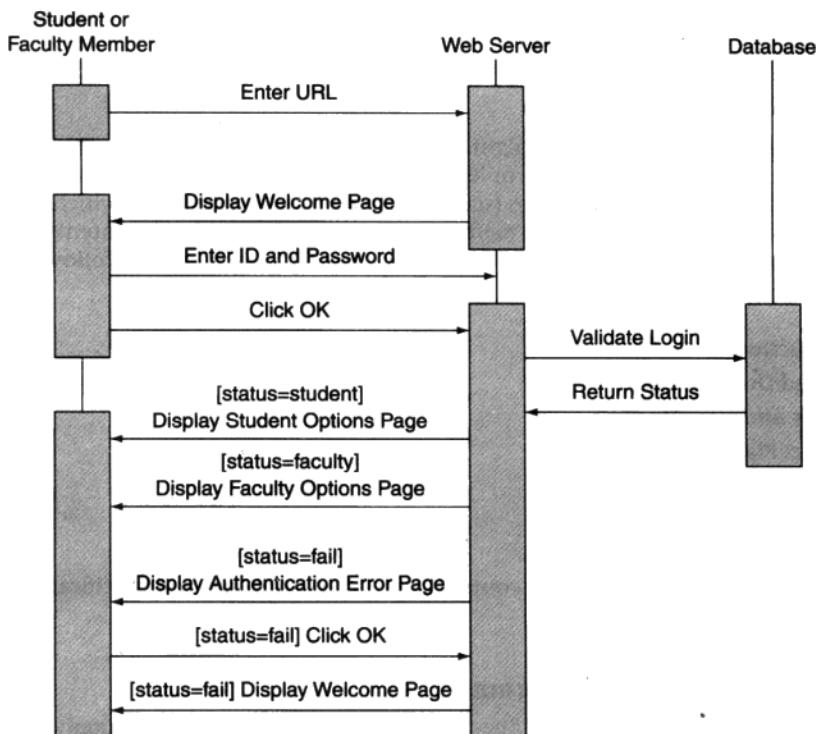
Note the relationship between the contents of the requirements and specification documents.

### 14.4.1 UML Sequence Diagrams

Part of the plan for developing a Specification Document from a Requirements Document might be to expand each use case into a UML sequence diagram. A **sequence diagram** is a graphic display of the temporal order of the interactions between the actors in a use case and the other modules in the system.

Figure 14.2 is a sequence diagram for the Authentication use case. The actors in the use case (in the figure, a student or a faculty member) and the pertinent modules in the system (in the figure, the Web server and the database) are labelled at the top of the diagram. Time moves downward through the diagram, and the vertical line descending from each actor and module show its lifetime during the use case. The boxes on the vertical line show when that actor (or module) is active in the use case. The horizontal lines show particular actions taken by an actor or module. Thus, the sequence diagram starts with the student or faculty member typing in the URL of the Student Registration System, after which the Web server displays the Welcome Form (Figure 14.3). Note the notation used for conditional actions: *[status = student] Display Student Options Form*. This means that if the status returned by the Authentication interaction is “student,” the Web server displays the Student Options Form.

Note that Specification III.A in the Specification Document given in Section 14.5 is an English-language description of the sequence diagram in Figure 14.2.



**FIGURE 14.2** A sequence diagram for the Authentication use case.

In the following section, we present the initial part of Section III of the Specification Document. We give a design for the database in the the Student Registration System in Section 4.8 and the design and part of the code for the Registration Transaction in Section 15.7.

## 14.5 The Specification Document for the Student Registration System: Section III

Section III of the Specification Document—"Forms and Use Cases"—contains a detailed description of all interactions with users. Its initial part might look like this:

### III. Forms and Use Cases

- When the Student Registration System is entered, Form 1, the Welcome Form (Figure 14.3) is displayed. In Form 1
  - The Id and Password text boxes are filled in.

#### 14.5 The Specification Document for the Student Registration System: Section III

The diagram shows a rectangular window representing a computer screen. Inside, a larger rounded rectangle contains the user interface. At the top, the text "Welcome to the Student Registration System" is displayed in bold. Below it, a message reads "Please enter your login Id and password". There are two input fields: one labeled "Id" and another labeled "Password", both represented by empty rectangles. At the bottom of this inner area are two command buttons: "OK" on the left and "Exit" on the right. The entire window is enclosed in a double-line border.

**FIGURE 14.3** Introductory form for the Student Registration System.

2. The OK command button is clicked to run the *Authentication* use case.
  - a. If the Authentication use case fails, Form 2, the Authentication Error Form, is displayed.<sup>4</sup> In Form 2, clicking the OK command button returns to Form 1.
  - b. If the Authentication use case succeeds and the authenticated user is a student, Form 3, the Student Options Form, is displayed (as described in Specification III.B).
  - c. If the Authentication use case succeeds and the authenticated user is a faculty member, Form 4, the Faculty Options Form, is displayed.
3. The Exit command button is clicked to display Form 5, the Do You Really Want To Exit Form. In Form 5
  - a. The Yes command button is clicked to exit the Student Registration System.
  - b. The No command button is clicked to return to Form 1.

<sup>4</sup> We omit the figures for the other forms; they would be included in the actual specification.

- B. In Form 3, the Student Options Form,
1. The Course description menu item is selected to run the *Get Course Names* use case, which displays Form 6, the Course Name Form. In Form 6
    - a. A course option box is selected.
    - b. The OK command button is clicked to run the *Get Course Description* use case, which displays Form 7, the Course Description Form.
    - c. The Cancel command button is clicked to return to Form 3.

The remainder of Section III of the Specification Document is similar and is therefore omitted.

## **14.6 The Next Step in the Software Engineering Process**

After the implementation group has expanded the Requirements Document into the Specification Document and the customer has signed off on the Specification Document, the design portion of the project can begin. In contrast with specifications, which describe *what* the system is supposed to do, the design describes *how* the system is to do what it does. We discuss design in Chapter 15 and the specific issues involved in designing databases in Chapters 4 and 6. We gave a complete database design for the Student Registration System in Section 4.8 and the complete design and part of the code for the Registration Transaction in Section 15.7.

One reason so much time and effort is put into producing requirements and specification documents is that experience has shown it to be surprisingly difficult to build a system that actually satisfies the customer's needs. Often the system's requirements are quite complex, and the customer has difficulty articulating his needs in the precise manner needed for programming, or he leaves out important details (such as what is supposed to happen if a course is canceled after a number of students have registered for it) or specifies some feature and then is unhappy with that feature when it is implemented.

The U.S. Department of Defense, which is probably the largest customer for software systems in the world, says that over 56% of all the defects in software systems it contracts for are due to errors in the specifications. It is cheaper and more efficient to work with the customer at the beginning of the project to sharpen and refine the specifications than it is to reimplement the system at the end of the project if it does not meet the customer's needs.

### **BIBLIOGRAPHIC NOTES**

There are many excellent books on software engineering—for example, [Sommerville 2000]; [Pressman 2002]; [Schach 1999]. One of the very few books that address software engineering for database and transaction processing systems is [Blaha and Premerlani 1998].

## EXERCISES

- 14.1 Prepare a Requirements Document for a simple calculator.
- 14.2 According to the Requirements Document for the Student Registration System, one session can include a number of use cases. Later, during the design, we will decompose each use case into one or more transactions. The ACID properties apply to all transactions, but a session that involves more than one transaction might not be isolated or atomic. For example, the transactions of several sessions might be interleaved. Explain why the decision was made not to require sessions to be isolated and atomic. Why is a session not one long transaction?
- 14.3 Suppose that the database in the Student Registration System satisfies all the integrity constraints given in Section IV of the Requirements Document Outline (Section 14.2). Is the database necessarily correct? Explain.
- 14.4 The Requirements Document for the Student Registration System does not address security issues. Prepare a section on security issues, which might be included in a more realistic Requirements Document.
- 14.5 In the resolution of issue 2 in Section 14.3, the statement was made that new prerequisites do not apply to courses offered in the next semester. How can a Registration Transaction know whether or not a prerequisite is “new”?
- 14.6 Suppose that the Student Registration System is to be expanded to include graduation clearance. Describe some additional items that must be stored in the database. Describe some additional integrity constraints.
- 14.7 Prepare a Specification Document for a simple calculator.
- 14.8 Prepare a Specification Document for the controls of a microwave oven.
- 14.9 Specify a use case for the Student Registration System that assigns a room to a course to be taught next semester.



# 15

## Design, Coding, and Testing

Now that the Specification Document for the Student Registration System has been approved by the customer, we are ready to continue with the project. The next step is design.

### 15.1 The Design Process

In contrast to specifications, which describe *what* the system is supposed to do, the design describes *how* the system is to do what it does. Thus, the design of a transaction processing system includes

- The declaration of every global data structure used in the system, including the database schema and any data structures kept by application programs between transaction invocations
- The decomposition of each interaction described in the Specification Document into transactions and procedures
- Detailed description of the behavior of every module, object, procedure, and transaction contained in the system

The results of the design phase are presented in a Design Document, which, in a sense, is an extension of the Specification Document. The Specification Document describes in detail the capabilities of the system, whereas the Design Document describes in detail how each of these capabilities is to be implemented.

The design process itself is often viewed as the most creative part of the implementation project. Good designs are simple and elegant. The designer formulates and evaluates various design alternatives (for example, various table designs) to achieve the desired functionality and then, based on her judgment and experience, makes decisions that significantly influence the system implementation and its ultimate performance. Unfortunately, while making these decisions is the enjoyable part of the process, documenting the details in the Design Document is often one of the designer's more tedious (but nevertheless necessary) tasks.

The users of the Design Document include

- The coding group, who use it as their sole source of information in their coding

- The quality control group, who use it, together with the Specification Document, to design tests and to determine what went wrong when an error is discovered
- The maintenance group, who use it (at a later date) to implement enhancements to the system

An important part of the design process is to make all global decisions (i.e., those that affect multiple transactions and procedures) so that later, during the coding phase, each coder can implement each individual transaction or procedure with no knowledge of the overall system other than that provided in the Design Document. If the Design Document is incomplete to the extent that the coder of a particular transaction or procedure has to make a global decision, that decision is liable to be inconsistent with a decision made by the coder of another transaction or procedure about the same global issue—thus causing an error.

For example, suppose procedure  $P_1$  calls procedure  $P_2$  with certain arguments that must be in some specified range, and suppose the Design Document does not specify whether that range check is to be made in  $P_1$  or  $P_2$ . Then if the programmer of  $P_1$  expects it to be made in  $P_2$  and the programmer of  $P_2$  expects it to be made in  $P_1$ , that check will not be made at all, and a potentially serious error has occurred.

### 15.1.1 Database Design

The design document must contain a complete design of the database, including a set of executable statements that declare the database schema (such as SQL's CREATE statements).

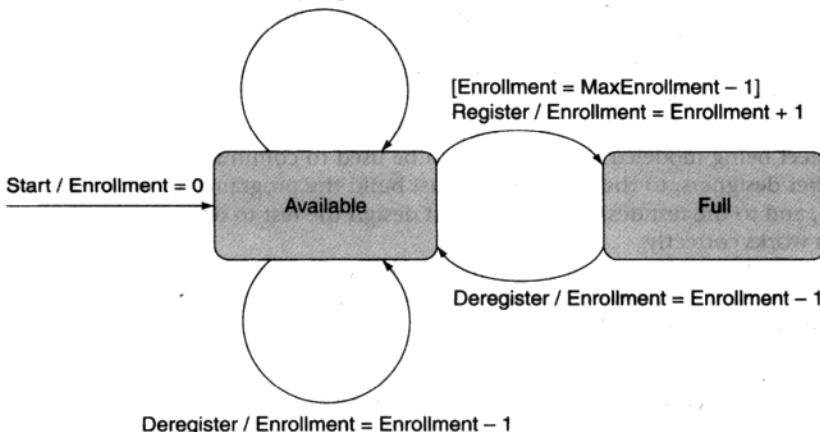
A typical approach to database design, discussed in this book, includes several stages. First E-R or UML class diagrams are used to model the business objects involved in the enterprise. These diagrams are then converted into relational schema designs. This part of the process was discussed in Chapter 4, and a concrete case study for the Student Registration System was presented in Section 4.8. The next step is to normalize the schema and bring it into compliance with one of the desirable normal forms. This step was the subject of Chapter 6, and a case study was performed in Section 6.12. The last step in database design is tuning. Indexing, described in Chapter 9, is one important part of this step. The overall strategy for tuning databases was described in Chapter 12.

Database schema design is an integral part of the overall design document. Other parts of this document are described below.

### 15.1.2 Describing the Behavior of Objects with UML State Diagrams

UML class diagrams or E-R diagrams can be used to model the business objects stored in the database. However, these models are *static* in that they model only the data stored in the database, but not how that data changes when operations are performed on it. *UML state diagrams* are one way to model the *dynamic* behavior of these objects—how their internal state changes when their methods are invoked.

[Enrollment < MaxEnrollment - 1] Register / Enrollment = Enrollment + 1



**FIGURE 15.1** A UML state diagram for the CLASS object.

For example, in the Student Registration System, one part of the internal state of a CLASS object is its Enrollment attribute. One of the integrity constraints is that the value of Enrollment must be less than or equal to the value of the MaxEnrollment attribute. We might say that when Enrollment is equal to MaxEnrollment, the CLASS is *Full*; otherwise it is *Available*. Now we can describe the dynamic behavior of CLASS as follows:

1. When the CLASS is *Available*, a *Register* operation causes the value of Enrollment to be increased by one. If that increase causes Enrollment to equal MaxEnrollment, then CLASS becomes *Full*.
2. When the CLASS is *Full*, a *Register* operation cannot occur (is unsuccessful if attempted).
3. A *Deregister* operation causes the value of Enrollment to be decreased by one. If the CLASS is *Full*, it becomes *Available*.

We can model that behavior by the UML state diagram shown in Figure 15.1. CLASS has two states, *Available* and *Full*. The arrows between the states are *transitions*, which model the operations and denote when an operation causes the state of the object to change. Each transition is of the form

---

[guard] operation / action

---

for example,

---

[Enrollment < MaxEnrollment-1] Register / Enrollment = Enrollment+1

---

The *guard* is a Boolean expression, which states the condition under which the transition can take place. In the example, the transition can take place only if Enrollment is less than MaxEnrollment – 1. The *operation* specifies the method that causes the transition. The *action* then specifies how the attributes within the object change when the transition occurs.

Thus the state diagram provides a graphic model of the dynamic behavior of the object being modeled. That model can be used to communicate the design to the other designers, to the coders who must build the programs to implement the design, and to the test designers, who must design the test to demonstrate that the system works correctly.

UML state diagrams have many more options that are shown in this simple example. A more complete discussion, as well as more complex examples, can be found in any book on UML.

### 15.1.3 Structure of the Design Document

The sections of a design document for a transaction processing system might include

- A. **Title, author(s), date, version number.**
- B. **Introduction.** A brief description of the goals of the system
- C. **Related documents.** References to the Requirements and Specification documents and to any other documents used in the design or implementation—for example, a Visual Basic user manual, an ODBC specification document, or a specification document for some object library used in the design
- D. **High-level design.** An informal description of the design so the reader can more easily follow the detailed descriptions in later sections. Among the items that might be included:
  1. The decomposition of the system into modules or objects
  2. The decomposition of the interactions defined in the Specification Document into transactions and procedures
  3. A procedure calling tree
  4. A UML class diagram or an E-R diagram for the database design, including the rationale for any choices made in designing the diagram

Also included in this section can be the rationale for any design decisions that need to be documented. Examples might include:

5. Why a session was decomposed into transactions in one way rather than in another, perhaps more intuitive, way
6. Why a table was normalized, denormalized, or partitioned in a particular way, perhaps to meet some performance requirement
7. Why certain indices were defined
8. Why certain integrity constraints were to be checked by transactions rather than embedded in the database schema
9. Why certain transactions were allowed to execute at lower isolation levels
10. Why any other decisions were made to achieve performance requirements

**E. Declaration of the database schema and other global data structures**

1. *Database schema.*
  - a. The complete (compilable) set of statements that declare the database schema, including tables, indices, domain specifications, assertions, and access privileges—documented with the intended use of each table and column, and the rationale for each index
  - b. A list of the integrity constraints, together with a description of where each constraint will be enforced: in the schema or by the individual transactions
2. *Global data structures.* The complete (compilable) declaration of any other global data structures; for example, those kept by an application program for use by the transactions it initiates. Each item must be documented with its intended use and any constraints on its values.

**F. Graphical user interface.** Since a detailed description of the user interface appears in the Specification Document, only a reference to the appropriate document section is needed. However, any missing details of the user interface must be supplied here—including the specification of all events and objects (including their methods).**G. Detailed description of transactions and procedures.** For each transaction and procedure,

1. *Transaction or procedure name.*
2. *Description.* An informal, one-sentence description of what the transaction or procedure does. For example: "This transaction registers a student in a specified course after checking that the prerequisites are satisfied." (The goal is to explain the general purpose of the transaction, not to give its detailed functional specification.)
3. *Arguments.* *In* and *out* arguments of the transaction or procedure. Each argument must be documented, with its type and intended use.
4. *Return values.* The values that can be returned by the transaction or procedure, together with their type and intended use.
5. *Called from.* The procedures, or (GUI) events, that call this transaction. (For example, a transaction might be called when a particular mouse-click event occurs on some form object.)
6. *Calls.* The procedures called by this transaction, including any events it causes and any exceptions it raises. (These last two items are useful when the design or code needs to be changed, and the designer or coder must propagate changes throughout the design.)
7. *Preconditions.* Assumptions that the transaction or procedure can make (and does not have to check at run time) about the state of the database, the global data structure, and its arguments when it starts. For example, a transaction to register a student in a course might be able to assume that a previously executed transaction authenticated the student. As another example, a procedure invoked by a particular mouse-click event might be able to assume that an input value it needs was previously stored by the user

in a particular field of a particular form object. (Most of the global errors in large system implementations occur because of miscommunication about preconditions.)

8. *Isolation level.* The isolation level at which the transaction will execute.
9. *Actions.*
  - a. Textual description of the actions taken by the transaction or procedure. (Perhaps one or two paragraphs, compared to the one-sentence description given earlier—the goal here is to help the coder write the code.)
  - b. Database tables and global data structures accessed, together with the changes the transaction or procedure is supposed to make. For example, after a successful registration transaction, the student must be listed in the appropriate table(s) as being registered for the course, and the number of registrants for that course must be incremented.
  - c. Error situations.
    - i. Validity checks that the transaction or procedure must make about its arguments, the global data structure, or the database. For example, a transaction to register a student for a course might be required to check that the student has taken (or is taking) all the prerequisites for that course. Or a transaction accessing some field that is allowed to be null (for example, a registration transaction reading the course days and time) might be required to check that that field is not null at the time the transaction or procedure is executed. A description must be given of the actions to be taken when such a check fails.
    - ii. Automatic constraint checks the system will make on the updates of this transaction and the actions to be taken if these updates fail.
    - iii. Any other error or anomalous situations that might occur and the actions to be taken in each case. For example, what should happen if a database CONNECT statement fails? What exceptions should be raised and under what conditions?
  - d. Forms to be displayed in various circumstances—perhaps on successful completion or if some specified error situation occurs.

#### 15.1.4 Design Review

Near the end of the design process, a formal **design review** is often held in which all members of the design and quality assurance groups and perhaps some outside people participate. The participants are given the latest version of the specification and design documents before the review and are expected to have studied them before the meeting.

The designers make a formal presentation, and the participants are expected to gain an understanding of the design and to identify issues such as:

- Places where the Design Document is inconsistent with the Specification Document

- Places where the Design Document is incorrect, inconsistent, incomplete, or ambiguous
- Places where the efficiency of the design can be improved (perhaps by using a different table structure or a different indexing structure for a table)
- Any areas of risk to the project in meeting its goals. For example, does a particular search algorithm meet the response time specifications? Does the design require a new version of a database driver that might not be available in time to meet the schedule? Does some suspect assumption underlie a global decision?
- Any risks inherent in the execution of the delivered system. For example, is a person's life or health at risk if the system does not work according to its specifications. Does the design adequately address those risks?

The goal of the design review is to identify issues, not resolve them. Each issue identified is assigned to a member of the design team for resolution by some specified date and for inclusion in a later version of the Design Document.

Any errors found during the design review are much easier and cheaper to correct than those found later, during the coding or testing phases. Errors found after the system has been delivered to the customer are still more expensive to correct.

The design review meeting might also include a review of the Test Plan, as described in the next section.

## 15.2 Test Plan

Testing is an important part of all software projects, not an informal ad hoc activity that the testers begin to think about only when the coding is complete. It is carried out in accordance with a formal Test Plan document, which is prepared during the design and coding phases of the project. The Test Plan document specifies the tests to be performed, the test data to be used, and the design of any test driver or scripting software needed to perform the tests.

The complete Test Plan might involve **module tests** performed by the coder of each individual module before the module is submitted for integration into the system, **integration tests** performed by the group that is integrating the modules into the system, and finally the **QA test set** performed by the quality assurance group on the completely integrated system.

We focus our attention on the final QA test set, but we first note an important aspect of module testing—testing the individual SQL statements within the module. These tests might include **code checks**, in which a colleague examines each SQL statement to verify that it satisfies its English-language specifications, as well as more conventional tests in which the SQL statements are executed against actual or test databases.

The design of an appropriate QA test set for a commercial transaction processing system is a significant endeavor. The test set might include tests designed using two different approaches.

**Black box tests** are designed from the Specification Document, without looking at the Design Document or the code. These tests assume that the system is just a “black box,” and they do not look inside. The goal is to verify that the system meets its specifications. Thus, there must be at least one test for every specification in the Specification Document (including error situations). Some specifications might have several tests. For example, to test that the number of students registered for a course does not exceed the specified maximum, the designer might include tests in which a registration transaction is executed when the number of previously registered students is both one less than the maximum and exactly the maximum. The test should also include cases in which the number is far from the maximum. (For example, what happens if the maximum is specified as 0 or 1?) Since it is impractical to include tests where the number of previously registered students and the maximum number of students both range over all possible integers (a completely exhaustive set of tests), the designer must use her experience to develop a test set that is representative of situations that might occur, meets the appropriate boundary conditions, and can be performed in the allotted testing time.

Because the user interface is specified in the Specification Document, the black box tests must test the user interface.

**Glass box tests** are designed using the Design Document and the code. They are called “glass box” because they look inside the system. The goal is to verify that the detailed coding is correct. Thus, glass box tests should visit every line of code, visit every branch of the code, check the boundary conditions of every loop, invoke every event, execute every integrity check, and exercise all aspects of every algorithm. For example, the code to check whether a student has all the prerequisites for a course might contain a while loop, and the test designer might include tests to verify that the exit condition of that loop is correct. Note that the existence of this while loop and its exit condition is not evident from the Specification Document, which is why we need glass box tests in addition to black box tests. However, we also need black box tests since the designers might have misunderstood some portion of the Specification Document, and any test set based only on the Design Document would not find such errors. For example, a glass box test might show that the exit condition of a while loop corresponds to the Design Document, but the Design Document might be an incorrect interpretation of the specification.

The Test Plan might also include **stress tests** in which a (possibly simulated) realistic load is placed on the system to verify that it meets its specifications for transaction throughput and response time. Such tests might uncover situations in which a large number of deadlocks occur or in which the database design needs tuning for other reasons to increase throughput or decrease response time.

If the application is built on a system that guarantees the ACID properties, we do not need additional tests to verify that the concurrent execution of transactions works correctly, assuming that we have already tested that each transaction works correctly when executed in isolation. However, if the application is being executed at some level of isolation less than **SERIALIZABLE** (see Section 8.2.3), additional tests might be necessary to ensure correctness under concurrent execution.

Often neglected in a Test Plan is testing of the User Manual (and other deliverable documentation) to ensure that it corresponds to the specifications and to the delivered system.

The Test Plan document contains a script of all tests that are to be performed and the correct result of each. Since the Test Plan contains a large number of tests and must be executed many times during the testing and maintenance phases (after fixing some error or adding some new feature), it is highly desirable to employ a test driver or scripting mechanism to automate the Test Plan execution. If such a mechanism is used, the Test Plan document contains the appropriate inputs to the test driver. If a test driver is to be implemented as part of the project, its design must also be included.

The Test Plan document also contains a description of the testing protocol that will be used by the testers when performing the tests (or a reference to the company's standard testing protocol document). That protocol should include an **Error Report Form**, which must be filled out when an error is found. The Error Report Form should include the tester's name, the date, the error description, and, most important, a detailed description of how to recreate the situation in which the error occurs. The Error Report Form is passed on to the person responsible for fixing the error, who fills in information about when and how the error was fixed and which version of the code contains the fix. The entire protocol must ensure that all errors that are found are eventually fixed and that some version of the code contains all of the fixes.

To design a Test Plan that is comprehensive (in the sense just described) and yet manageable takes a considerable amount of skill and experience. The actual size of the test set and the scope of the testing effort for any particular application are often marketing as well as technical decisions, dependent on a number of sometimes conflicting factors. For example,

- How critical is the correctness of the system? Are people's lives at stake?
- How important is time to market of the system? Is a competitive product about to be released or is there an upcoming trade show at which the system must be exhibited?

In some critical applications, half of the entire time allocated to the project is devoted to testing. Sometimes issues of professional ethics arise when management applies pressure to release an inadequately tested product.

In applications in which the system is being implemented by one group and delivered to another group, which will operate and maintain it, the test set (together with any drivers or scripting mechanisms) is often one of the deliverables, along with the code and the documentation.

Some customers might insist that a complete history of the testing process, including the dates and results of all tests, errors found, errors fixed, etc., be delivered with the software in order to demonstrate that the test plan had been successfully carried out. In some situations both the customer and the developers store this history permanently, together with the rest of the documentation for the system, in case they are needed in some future legal situation, which might involve alleged

defects in the system. The company might need to demonstrate to a court that the system was developed and adequately tested according to standard software engineering practice.

**Acceptance testing and beta testing.** In addition to the Test Plan, which is prepared and carried out by the system implementors, the customer often prepares and carries out an acceptance test before accepting the system (and perhaps before making the final payment for it). An acceptance test is usually composed of realistic inputs and an actual database (in contrast with the implementor's Test Plan, which often involves boundary case inputs and test databases) and is intended to increase customer confidence that the system will fulfill its purpose. A savvy customer will spend considerable effort to design an acceptance test that comprehensively exercises the system in real-world situations.

If the system is a product with many customers, a small set of these customers is often selected to perform **beta testing**. In this context, the testing performed by the system implementors is called **alpha testing**. When the alpha testing is completed, the **beta test version** is supplied to customers, who use it on their actual applications and report any errors to the implementors. Because the beta test version might still contain serious errors, the customer might be at some risk in using it (especially since implementation groups have been known to minimize the amount of alpha testing and rely on beta testing to find many of the errors in their system), but the customer gets the benefits of receiving an early version and might receive some other financial or technical incentives as well. The beta testing continues for some period of time, perhaps weeks or months, after which the initial release of the system is made to all customers.

Even after all of this testing, customers with critical applications often run a new system in parallel with their existing system for some period of time until they gain sufficient confidence that it can do its job reliably and correctly.

### 15.3 Project Planning

Project planning is another important part of software engineering. While the Specification Document is being prepared, the project manager makes an initial version of the Project Plan. To make such a plan, the manager divides the project into a set of **tasks**, estimates the time required to complete each task, and then assigns each task to a specific person (or group) together with targeted start and completion dates.

A task might involve design, coding, testing, or documentation, but it must have the property that its completion can be precisely defined—for example, “The coding of module 3 is complete,” not “Module 4 is 90% debugged.” Estimating the time it will take to complete a given task is quite difficult and requires understanding the complexity of the task and relating that complexity to the skill of the person assigned to carry it out.

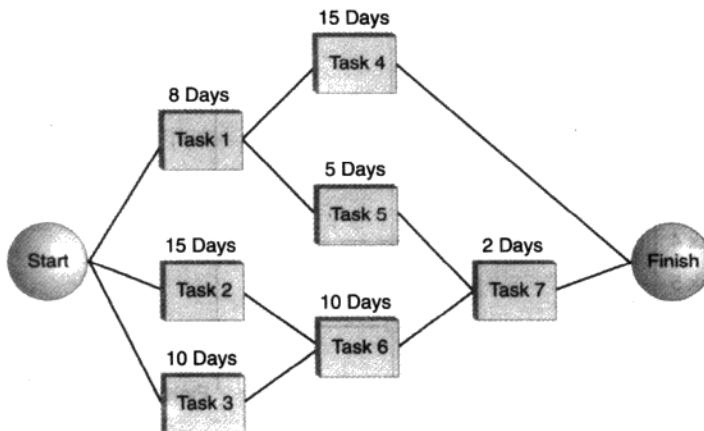


FIGURE 15.2 Dependency chart.

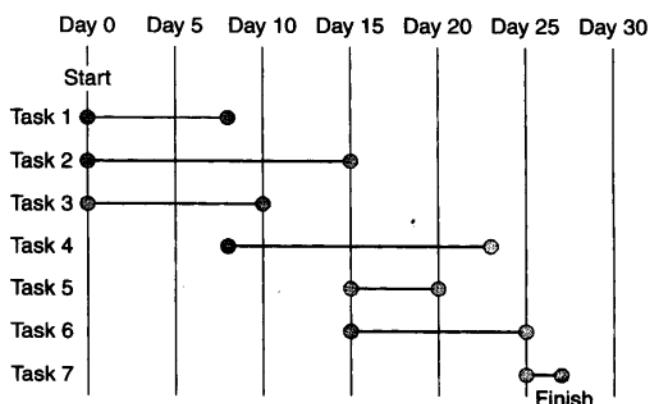
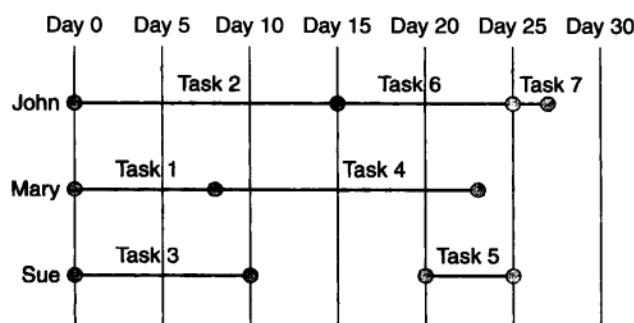
An essential aspect of task scheduling is **task dependency**: certain tasks cannot be started until certain others have been completed. For example, the testing of a module cannot begin until its coding is finished.

Given the dependencies and estimated duration of tasks, a **Dependency chart**, sometimes called a **PERT chart** (Program Evaluation and Review Technique), can be constructed, as shown in Figure 15.2. Activities are represented by rectangles with the duration written above each (sometimes maximal and minimal estimates of task duration are included) and dependencies represented by lines. In this way, the chart shows which tasks can be done in parallel and which must be done sequentially. The longest path through this chart from start to end is called the **critical path** and is an estimate of the minimum time required to complete the project.

Other charts used to document the Project Plan include the following:

- A **Gantt chart** (named after its developer, Henry Gantt)—a bar chart showing when tasks are scheduled to start and complete (see Figure 15.3)
- A **Staff Allocation chart**—a bar chart showing the assignments of individual staff members to specific tasks, together with the scheduled start and completion dates (see Figure 15.4)

As the project proceeds, the project manager schedules periodic (perhaps weekly) project meetings at which implementation team members report on the status and expected completion date of each assigned task, compared with its scheduled completion date in the Project Plan. The project manager should encourage an environment in which team members feel comfortable in honestly reporting when they are having trouble with their assigned task and might not be able to complete it on schedule. If necessary, the project manager then makes the appropriate decisions

**FIGURE 15.3** Gantt chart.**FIGURE 15.4** Staff allocation chart.

to ensure that the project completes on time. Tasks on the critical path must be monitored especially carefully. (Often the best people on the staff are assigned to them.) It is particularly important that minutes be kept of these meetings to document any decisions taken and assignments made.

When the project is completed, many project managers hold a windup meeting with the entire team to discuss what went right and wrong with the planning and other aspects of the project. The goal is to learn from any mistakes and to increase the software engineering skills of the manager and the team members.

Although project management software is available to automate the production of project planning charts, the preparation and monitoring of a Project Plan require a considerable amount of skill and experience. Indeed, the main reason many software projects fail or are late is the project manager's lack of skill in project planning and monitoring.

## 15.4 Coding

In most software projects, the time allocated for the coding phase is less than one-sixth of that allocated for the entire project. That fraction will become even smaller as reusable object libraries automate the production of code directly from the design.

A well-run IT department should have coding guidelines that every programmer follows. These guidelines vary from company to company, but there are many common rules. Some programming languages, such as Java, have their own coding styles, but this is usually only a small portion of what a typical set of coding standards mandates. One well-known example of such a set is known as the GNU Coding Guidelines [Stallman 2000]. Here we include some hints for producing professional-quality code:

- The two highest priorities in coding are *correctness* and *clarity*.<sup>1</sup> In addition to being correct, the code must be understandable to the large number of people (such as the quality assurance and maintenance groups) who will read it over its lifetime.
- All of the coders should use the same style of variable definitions, indentation, and the like. Good text editors and integrated development systems provide tools for automatically indenting programs according to certain rules. The program for the entire system should look as if it had been written by a single coder.
- Variables and procedures should have application-oriented names that make their use self-evident—for example, not S, or Stu, or even Student, but Student\_Name or Student\_ID\_Number.
- Comments should be used effectively:
  - The code for each module, procedure, and transaction should begin with a **preamble**, which is the same as its detailed description in the Design Document and can also include the author, date, and revision number. Some guidelines require that a **revision history** be included in the program file, which can include entries such as

---

```
1999-12-12: Mary Doe (md@company.com)
    foo.java (checkAll): added capability to check userids
1998-03-22: John Public (jp@company.com)
    foo.java (checkCredentials): fixed bug in while loop
```

---

However, keeping the revision history inside program files is becoming obsolete due to the development of sophisticated version control systems. A better approach is to keep the revision history for all files that are in the same directory in a separate file, often called ChangeLog. This allows

<sup>1</sup> The third “c,” *cleverness*, has the lowest priority.

the developers to see the changes made to all files in one place and in chronological order. This style is superior when there are dependencies between the code in different files in the same directory. The history of individual files can typically be obtained from the version control system, which is a better place to keep it than inside the program file.

- Comments within a module, procedure, and transaction should be application oriented. For example, the following comment is useless

---

```
/* Increment number_registered */
number_registered = number_registered+1;
```

---

because it is self-evident from the code. If this line of code were part of the registration transaction and needed to be documented at all, a better comment might be

---

```
/* Another student has registered */
```

---

Not every statement needs to be commented. Some programming style books suggest that all loop and conditional statements be commented. For example, comments for a loop and conditional statement, respectively, might be

---

```
/* Give all employees making ≤ $10,000 a 5% raise. */
```

---

or

---

```
/* If the customer has exceeded the credit limit,
** then abort transaction. */
```

---

- When the program needs to be changed because of bug fixes or enhancements, the appropriate comments should be updated along with the code.
- System-specific data structures and code should be clearly documented as such. For example, some vendors supply their own version of the embedded SQL CONNECT statement

---

```
EXEC SQL CONNECT student_database IDENTIFIED BY pml
    DBMS_PASSWORD = 'z9t.56';
/* ***System Specific: Ingres version of CONNECT */
```

---

Then, if the system must be ported to a different DBMS at some future time, the system-specific statements can be easily found with a text editor. If the CONNECT statement appears several times in the system, it can be encapsulated within a

procedure, which can be called at the appropriate points. Then if the system must be ported, only the body of that procedure need be changed.

**Prototype coding.** We have sharply differentiated the coding phase of the project from the requirements analysis and design phases. However, in practice, prototypes of parts of the system are often coded during the earlier phases in order to aid in decision making. For example, in a transaction processing system implementation, the following prototypes might be necessary:

■ *During the requirements analysis phase.*

- Prototype of the user interface to evaluate its clarity and usability and to incorporate customer feedback
- Prototype of the DBMS access code to compare the relative speeds of various design choices—for example, stored procedures compared with those stored within the application program. These experiments can also be used to validate the specification of the expected transaction throughput of the final system, or they might be used to evaluate (and perhaps reduce) the time required to perform these tasks in the coding and testing phases.

■ *During the design phase.*

- Prototypes of possible designs for a specific table to evaluate the effect of various indexing schemes on the time required to execute particular SELECT or UPDATE statements
- Prototypes of other parts of the design to evaluate and reduce the risk involved in certain design decisions

Sometimes this prototype code is later discarded. Sometimes it is used in the production version.

## 15.5 Incremental Development

Most large implementation projects take several years to complete. During that time, the goals and needs of the enterprise sponsoring the project might change considerably. Therefore, during the course of the project, the managers of the enterprise might request significant changes in the specifications of the system being built. An important issue is how the implementation team responds to such requests.

One approach is to continue building the system as originally specified. Unfortunately, even if the project is successfully completed, the resulting system might not completely satisfy the needs of the enterprise. Of course, implementation of the next version of the system, including some or all of the requested changes, can start as soon as the first version is completed and hopefully can be completed in a relatively short time.

Another approach is to try to keep up with the managers' requests, changing the specifications, design, and code many times during the project. Unfortunately, the project is often never successfully completed—the design is continuously being revised, the code is constantly being rewritten, and the project gets further and

further behind schedule, with more and more money required to keep it going, until finally it is canceled. According to the Standish Group Report [Standish 2000], 31% of all information system projects are canceled before they are completed. Successful use of this approach requires discipline (in limiting the number and scope of changes that are allowed) and skill (in negotiating with management on budget increases and schedule extensions).

Still another approach, called **incremental development**, involves building the system in stages. First a **core version** of the system is built based on a subset of the initial specifications, perhaps some subset of the use cases, which can be implemented in a short period of time. Then successive versions are built, each including more of the specified functionality and some of the changes that the managers have requested, based on both the changed needs of the enterprise and the experience gained from running the previous versions. After several such incremental versions have been built, the system includes most of the requirements. Of course, it never includes all the requirements since the managers tend to continually revise them.

The incremental approach is often better than the other two approaches. Risk is minimized because an operational system is available in a relatively short period of time and any enhancements to it can be based on working experience. A potential disadvantage is that, unless the initial design is carefully done, design decisions made for early versions might be inappropriate for the increased functionality of later versions. However, good object-oriented design minimizes this disadvantage since the internal design of objects can be changed and new methods can be added without affecting any existing code.

Unfortunately, some designers use the incremental approach as an excuse to do no design at all and just hack together each version on top of the previous one. Needless to say, that approach can lead to disaster.

## 15.6 The Project Management Plan

Many organizations require that all of their software project managers prepare a Project Management Plan at the start of every project. In addition, if the project is to be built for some external customer organization, many such customers require that all prospective vendor organizations prepare and submit a Project Management Plan as a part of their project proposals, and then they use that plan to evaluate and compare the capabilities of the vendors.

The readers of a Project Management Plan include the management of the organization performing the project, the management of the customer organization, and the individual members of the implementation team. One of the goals in preparing the plan is to convince both managements that the project will be managed in a professional way and is likely to be completed on time and within the allocated budget. Another goal is to inform the team members of what will be happening during the course of the project. Perhaps the most important goal is to allow the project manager to organize and formalize her ideas as to how the project will be managed and to communicate those ideas to all concerned parties.

The sections of a Project Management Plan might include

- A. **Title, author(s), date, version number.**
- B. **Project scope.** A brief description of what the system is intended to accomplish, which features are and are not included in the system, what other systems it must interface with, etc.
- C. **Deliverables.** Documentation, code, test results, testing software, etc.
- D. **Schedule and cost estimates.** Milestones, project planning charts, rationale for cost estimates, etc.
- E. **Personnel plan.** People (and their capabilities) assigned to the project, organization of teams to perform specified tasks, training plan if needed, etc.
- F. **Quality assurance plan.** Scheduled reviews, test plan (including plan for designing tests, performing tests, and fixing error found during testing), QA plan for documentation, configuration management plan, standards to be used for documentation, coding, testing, etc.
- G. **Project risk management plan.** Plan for identifying and dealing with risks involving project goals and schedules. For example, project managers might have identified a risk of a particular transaction not meeting its requirements for response time. The plan might be to assign a team to develop prototype code involving different transaction designs, database designs, indexing strategies, and assignments of space in the memory cache, and to perform experiments to evaluate the response times for each. Another risk might be not meeting the project schedule for the completion of a task involving interfacing two subsystems. The plan might be to assign a team to this task early in the project and to have that team consult with the developers of those subsystems.
- H. **System risk management plan.** Plan for identifying and dealing with risks involving the execution of the delivered system. For example, in a system to control an X-ray machine, one risk might be that the patient would receive an excessive dose of X-rays. Again, the plan might be to assign a team to address that risk. The team might isolate the code that calculates the required dose and controls the dose delivered by the X-ray system, and then take special care in the design, coding, and testing of that code. In particular, the test plan for that segment of the system might be subject to special review by the customer and the entire team, including the manager.

## 15.7 Design and Code for the Student Registration System

In Section 4.8, we discussed the design of the tables and some simple constraints suitable for the Student Registration System. In this section, we complete the design by providing details of the more complex constraints and part of the code for the registration transaction.

### 15.7.1 Completing the Database Design: Integrity Constraints

An important part of database design is listing the database integrity constraints and deciding how each will be checked: automatically by the DBMS (in a `CREATE TABLE`, `CREATE ASSERTION`, or `CREATE TRIGGER` statement) or in one or more of the transactions. In the initial database design in Section 4.8, we did not fully discuss this issue because we had not yet introduced some needed SQL constructs.

The following is a list of the database integrity constraints, showing where each is enforced: in the schema (shown in Figures 4.34, 4.35, and 15.5 on pages 116, 117, and 527) or in the individual transactions. They are the same as those given in the Requirements Document in Section 14.2 except that they have been expanded to specify the attributes and tables involved in each constraint.

- *Uniqueness of Ids.* Each Id in the STUDENT table, each Id in the FACULTY table, and each CrsCode in the COURSE table must be unique. This is enforced by the primary-key constraints in the corresponding tables, as described in Section 4.8.
- *If a student is listed as registered for a course in some semester/year, that course must be offered at that time.* If a tuple exists in the TRANSCRIPT table with a particular CrsCode, Semester, and Year, there must be a tuple in the CLASS table with that CrsCode, Semester, and Year. This is enforced by the registration transaction, `Register()`, in Section 15.7.3.
- *Enforcement of the enrollment limit.* The value of the Enrollment attribute for a tuple in the CLASS table cannot be larger than the value of the MaxEnrollment attribute for that same tuple. This is enforced by a CHECK constraint in the CLASS table, Section 4.8.
- *Enrollment consistency.* The value of the Enrollment attribute for a class in the CLASS table must be equal to the number of tuples in the TRANSCRIPT table corresponding to students who have registered in that class for that semester and year. This is enforced by the ENROLLMENTCONSISTENCY assertion in Figure 15.5.
- *An instructor cannot be assigned to two courses taught at the same time in the same semester.* Two tuples in the CLASS table cannot have the same ClassTime, Semester, Year, and InstructorId. This is enforced by a UNIQUE constraint in the CLASS table in Section 4.8.
- *Two courses cannot be taught in the same room at the same time in the same semester.* Two tuples in the CLASS table cannot have the same ClassroomId, Semester, Year, and ClassTime. This is enforced by a UNIQUE constraint in the CLASS table in Section 4.8.
- *Students enrolled in a course must have completed all the prerequisites for it with a grade of C or higher.* For each tuple,  $t_i$ , in the TRANSCRIPT table with attributes StudId, CrsCode, Semester, and Year, if there are any tuples,  $t_r$ , in the REQUIRES table with that same CrsCode value and the value of EnforcedSince preceding the semester designated in  $t_i$ , then for each such  $t_r$ , there must be a tuple,  $t_n$ , in the TRANSCRIPT table with the same value of StudId, a CrsCode value equal to the value of the PrereqCrsCode attribute of  $t_r$ , an earlier value of Semester and

**FIGURE 15.5** Some constraints for the Student Registration System.

```

CREATE ASSERTION ROOMADEQUACY
  CHECK ( NOT EXISTS (SELECT *
    FROM CLASS C, CLASSROOM R
    WHERE C.MaxEnrollment > R.Seats
      AND C.ClassroomId = R.ClassroomId ) )

CREATE ASSERTION ENROLLMENTCONSISTENCY
  CHECK (
    NOT EXISTS (SELECT *
      FROM CLASS C
      WHERE C.Year = EXTRACT(YEAR FROM CURRENT_DATE)
        -- current_semester() is a user-defined function
        AND C.Semester = current_semester()
        AND C.Enrollment <>
          (SELECT COUNT( * )
            FROM TRANSCRIPT T
            WHERE T.CrsCode = C.CrsCode
              AND T.Year = C.Year
              AND T.Semester = C.Semester)

CREATE TRIGGER CANTCHANGEGRADETOI
  AFTER UPDATE OF Grade ON TRANSCRIPT
  REFERENCING OLD AS O
  NEW AS N
  FOR EACH ROW
  WHEN (O.Grade IN ('A','B','C','D','F') AND N.Grade = 'I')
  ROLLBACK

```

---

Year, and a Grade of at least C. This is enforced by a call to the `checkPrerequisites()` method in the registration transaction in Section 15.7.3.

- A student cannot be registered for different courses taught at the same hour. No pair of tuples in the TRANSCRIPT table with the same StudId can have values of CrsCode, SectionNo, Semester, and Year attributes, such that the tuples with the corresponding values of CrsCode, SectionNo, Semester, and Year attributes in the CLASS table have the same ClassTime. This constraint is not shown.
- A student cannot be registered for more than 20 credits in any given semester. Let  $S$  be a set of all tuples in TRANSCRIPT having the same values in attributes StudId, Semester, and Year. Then the sum of the values of the CreditHours attribute in the COURSE table corresponding to the CrsCode values in tuples in  $S$  must be less than or equal to 20. This is enforced by a call to the method `checkRegisteredCredits()` in the registration transaction in Section 15.7.3.

- *The room assigned to a course must have at least as many seats as the maximum allowed enrollment for the course.* If *cl* and *cr* are tuples in tables CLASS and CLASSROOM, respectively, with the same value of ClassroomId, then the value of MaxEnrollment in *cl* is less than or equal to the value of Seats in *cr*. This is enforced by the ROOMADEQUACY assertion, in Figure 15.5.
- *A valid letter grade cannot be changed to an Incomplete.* Once a value of A, B, C, D, or F has been assigned to a Grade attribute in a tuple in the TRANSCRIPT table, it cannot be changed later to an I. This is enforced by the CANTCHANGEGRADETOI trigger in Figure 15.5.

Figure 15.5 completes the database design by defining the assertions and trigger portion of the schema.

### 15.7.2 Design of the Registration Transaction

We now present a design for the registration transaction in the format given in part G of Section 15.1.3. When a student wants to register for a particular course, she starts an application, which presents a GUI with a choice of courses offered in the next semester. When a particular course is selected, the GUI creates an instance object of the Java class ClassTable on page 531 using the constructor `ClassTable()`. Then, when the student presses the registration button on the GUI, the registration transaction (whose code is in the body of the `Register()` method of that class) is invoked on the new object.

1. *Transaction name.*

---

```
public int Register(Connection con, String sid)
```

---

This transaction is implemented as a method of class `ClassTable`. It is invoked on concrete instances of that class. The variables `courseId` and `sectionNo` are set by the class constructor, `ClassTable()`, to correspond to a particular class offering.

2. *Description.* This transaction registers a student in a course, after making a number of checks on the validity of the registration.
3. *Arguments.*
  - a. `Connection con`—the identifier of the database connection
  - b. `String sid`—the Id of the student registering
4. *Return values.*
  - a. If registration is successful, returns the constant `OK`.
  - b. If any of the checks described in *Actions*, (item 9, below) fails, returns a string corresponding to the nature of the failure.
  - c. If any database operation fails, returns the user-defined constant `FAIL`.

5. *Called from.* Not relevant here because we do not give the complete design with the names of the other classes and methods.

6. *Calls.*

- a. `checkCourseOffering()`
- b. `checkCourseTaken()`
- c. `checkTimeConflict()`
- d. `checkRegisteredCredits()`
- e. `checkPrerequisites()`
- f. `addRegisterInfo()`

The first five of these procedures perform the checks described under *Actions*, (item 9, below). If all the checks succeed, the last procedure updates the database to complete the registration.

7. *Preconditions.*

- a. The student whose Id is contained in the argument `sid` has been authenticated.
- b. The database has been opened, and the connection `con` has been created.
- c. The JDBC method `setAutoCommit(false)` has been executed on the connection.

8. *Isolation level.* `SERIALIZABLE`: Set using the call to  
`setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE)`

9. *Actions.*

a. *Textual description.*

- i. The transaction checks that
  - a. The course is offered the following semester.
  - b. The student is not already registered for the course, is not currently enrolled in the course, and has not completed the course with a grade of C or better.
  - c. The student is not already registered for another course scheduled at the same time.
  - d. The total number of credits taken by the student the following semester will not exceed 20.
  - e. The student has completed all of the prerequisites for the course with a grade of C or better (or is currently enrolled in some prerequisites).
- ii. If the student satisfies all of the checks, the transaction completes the registration by adding a new tuple for that student and class to the `TRANSCRIPT` table and incrementing the `Enrollment` attribute for that class in the `CLASS` table. It then commits and returns with status `OK`.

b. *Tables accessed.*

- i. For the checks: `STUDENT`, `COURSE`, `REQUIRES`, `CLASS`, `TRANSCRIPT`
- ii. For the updates: `TRANSCRIPT`, `CLASS`

c. *Error situations.*

i. **Validity checks**

If any of the checks described in *Actions* (item 9) fails, the transaction aborts and returns with a status that identifies the failure. For example, if the student has already taken the course with a grade of C or better, the method `checkCourseTaken()` returns the user-defined constant `CourseTaken` as a status, and the `Register()` transaction then returns with that status. Likewise, if a scheduling time conflict has been detected, the method `checkTimeConflict()` returns `TimeConflict`, and the transaction then returns with that status. The calling method is responsible for producing an appropriate error message.

ii. **Automatic constraint checks performed by the system**

The number of students registered must not exceed the maximum allowable enrollment, `MaxEnrollment`, for the course. (This is enforced by an assertion.) If this check fails, the transaction is aborted by the DBMS, and the calling procedure is responsible for producing an appropriate message.

iii. **Other anomalous situations**

If any database operation fails, the transaction aborts and returns `FAIL`. The calling procedure is responsible for producing an appropriate error message.

### **15.7.3 Partial Code for the Registration Transaction**

The example that follows shows part of the Java program for the registration transaction. The program defines class `ClassTable` and, in particular, its key method `Register()`, which specifies the course registration transaction. In addition, the example provides the code for one consistency checking method, `checkCourseTaken()`, but omits the others, which are similar.

The method has been structured to make it easy to code and understand. First the required checks are performed to see if the requested registration is allowed. Each check is made by a separate procedure. If all of the tests succeed, the tables are updated by the procedure `addRegisterInfo()`. If the updates succeed, the transaction commits. The bulk of the database actions performed by the check and update procedures can be implemented as stored procedures on the database server.

As the design specifies, all error and success messages are produced not by the registration transaction but by the GUI program that calls the registration transaction, based on the value the transaction returns. This design has advantages because it allows the system to be implemented in a two- or three-tiered architecture consisting of a presentation server, which contains programs that manage the displays on the screen, and an application server, which contains the application programs that do the actual work. Using a three-tiered architecture for the Student Registration System, the GUI program executes on the presentation server, the `register` method executes on the application server, and the procedures listed in item G.6 (page 513) of the Design Document are stored procedures executed at the database server.

```
public class ClassTable
{
    private String courseId;           // The course Id of the class
    private String sectionNo;          // The section number of the class
    .
    .
    // General return codes
    final public static int FAIL = -1;
    final public static int OK = 0;
    // Return codes for the various consistency checks
    final public static int CourseNotOffered = 1;
    final public static int CourseTaken = 2;
    final public static int TimeConflict = 3;
    final public static int TooManyCredits = 4;
    final public static int PrerequisiteFailure = 5;

    // The class constructor
    public ClassTable(String courseId, String sectionNo)
    {
        this.courseId = courseId;
        this.sectionNo = sectionNo;
    }

    // The registration transaction
    public int Register(Connection con, String sid)
    {
        int status = OK;           // return code of check*Status() methods
        int addResult = OK;         // return code of addRegisterInfo()

        try {
            // Make all the required consistency checks
            if ((status = checkCourseOffering(con,sid)) != OK) {
                con.rollback();           // Course not offered
                return status;
            } else if ((status = checkCourseTaken(con,sid)) != OK) {
                con.rollback();           // Course already taken
                return status;
            } else if ((status = checkTimeConflict(con,sid)) != OK) {
                con.rollback();           // Time conflict found
                return status;
            } else if ((status = checkRegisteredCredits(con,sid)) != OK) {
                con.rollback();           // Too many credits
                return status;
        }
    }
}
```

## CHAPTER 15 Design, Coding, and Testing

```
    } else if ((status = checkPrerequisites(con,sid)) != OK) {
        con.rollback();           // Lacks prerequisites
        return status;
    }
    // Consistency checks OK. Update tables now
    if ((addResult = addRegisterInfo(con,sid)) != OK) {
        // Failed to update tables—rollback
        con.rollback();
        return FAIL;
    }
    // Registration succeeded
    con.commit();
    return OK;
} catch (SQLException sqle) {
    // Catches exceptions raised during execution of commit or rollback
    return FAIL;
} // try-catch
}
// Register()
```

---

The following program is an implementation of one of the checks performed in `Register()`—a check for whether the student has already taken the course and received a satisfactory grade.

---

```
// Another method of class ClassTable
private int checkCourseTaken (Connection con, String sid)
{
    // Construct the SQL command. Observe the use of single quotes
    // and spaces to produce a valid SQL statement
    // Also note: courseId is a variable defined in class ClassTable
    String SQLStatement = "select CrsCode from Transcript"
        + " where StudId ='" + sid
        + "' and CrsCode ='" + courseId
        + "' and Grade in ('A','B','C',NULL)";
    Statement stmt;
    try {
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(SQLStatement);
        // If the result set is non-empty, course has been taken
        if (rs.next()) {
            // course has been taken
            stmt.close();
            return CourseTaken;
        }
    }
```

```
// course has not been taken
stmt.close();
return OK;
} catch(SQLException sqle) {
    // catches exceptions raised during execution of SELECT
    return FAIL;
} // try-catch
} // end of checkCourseTaken()

// Other methods of class ClassTable are defined here
:
}

} // end of class definition for ClassTable
```

## BIBLIOGRAPHIC NOTES

Many of the issues in this chapter are discussed in greater detail in standard software engineering texts such as [Summerville 2000; Pressman 2002; Schach 1999]. The particular issues involved in modeling and designing databases and transaction processing applications are discussed in [Blaha and Premerlani 1998].

## EXERCISES

- 15.1 Prepare a Design Document and Test Plan for a simple calculator.
- 15.2 Explain why
  - a. Black box testing cannot usually test all aspects of the specifications.
  - b. Glass box testing cannot usually test all execution paths through the code.  
(This does not mean that glass box testing cannot visit all lines and visit all branches of the code.)
- 15.3 Explain why concurrent systems (such as operating systems) are difficult to test. Explain why transaction processing applications, even though they are concurrent, do not have these same difficulties.
- 15.4 Explain the advantages of incremental system development from the viewpoint of
  - a. The managers of the enterprise sponsoring the project
  - b. The project manager
  - c. The implementation team
- 15.5 In the design of the registration transaction given in Section 15.7.2, some of the required checks are performed in the schema and some in the transaction program.
  - a. Which of the checks performed in the program can be performed in the schema?
  - b. For each such check, change the schema to perform that check.

- 15.6** Rewrite the program for the registration transaction in Section 15.7.3
- Using stored procedures for the procedures that perform the registration checks
  - Using one stored procedure that performs all of the checks that are performed by individual procedures in the figure
  - In C and embedded SQL
  - In C and ODBC
- 15.7** Evaluate the coding style used in the program for the registration transaction in Section 15.7.3.
- 15.8** Prepare a test plan for the registration transaction given in Section 15.7.2.
- 15.9** For the deregistration transaction in the Student Registration System
- Prepare a design.
  - Write a program.
  - Prepare a test plan.



## PART FIVE

---

# Advanced Topics in Databases

In this part of the book we will discuss some of the more advanced topics: object-oriented databases, semistructured data, and XML databases.

Object databases are beginning to find their way into the mainstream both independently and as extensions to existing relational products. In Chapter 16, we will study the principles underlying the object data model and the corresponding SQL extensions.

XML databases represent an emerging field that is expected to become important once the underlying standards and tools are developed. In Chapter 17, we will discuss some of these emerging standards and their applications.



# 16

## Introduction to Object Databases

In this chapter, we introduce the concept of a *database object* and define the *object data model*. As an application of these ideas, we present the recent object-oriented extensions to SQL, which appeared in the SQL:1999 and SQL:2003 standards. The full version of this book [Kifer et al. 2004] introduces other related standards, such as ODMG (from Object Database Management Group) and CORBA (from the Object Management Group).

### 16.1 Shortcomings of the Relational Data Model

Relational DBMSs swept the database market in the 1980s because of the simplicity of their underlying relational model and because tables turned out to be just the right representation for much of the data used in business applications. Encouraged by this success, attempts were made to use relational databases in other application domains for which the relational model was not specifically designed—for example, computer-aided design (CAD) and geographical data. It soon became obvious that relational databases are not appropriate for such “nontraditional” applications. Even in their core application area, relational databases have certain shortcomings. In this section, we use a series of simple examples to illustrate some of the problems with the relational data model.

**Set-valued attributes.** Consider the following relational schema that describes people by their Social Security number, name, phone numbers, and children:

---

PERSON (SSN: String, Name: String, PhoneN: String, Child: String)

---

We assume that a person can have several phone numbers and several children, and that *Child* is a foreign key to the relation *PERSON*. Thus, the key of this schema

consists of the attributes SSN, PhoneN, and Child. Here is one possible relation instance of this schema:<sup>1</sup>

SSN	Name	PhoneN	Child
111-22-3333	Joe Public	516-123-4567	222-33-4444
111-22-3333	Joe Public	516-345-6789	222-33-4444
111-22-3333	Joe Public	516-123-4567	333-44-5555
111-22-3333	Joe Public	516-345-6789	333-44-5555
222-33-4444	Bob Public	212-987-6543	444-55-6666
222-33-4444	Bob Public	212-987-1111	555-66-7777
222-33-4444	Bob Public	212-987-6543	555-66-7777
222-33-4444	Bob Public	212-987-1111	444-55-6666

This schema is *not* in the third normal form because of the functional dependency

$$\text{SSN} \rightarrow \text{Name}$$

since SSN is not a key and Name is not one of the attributes in a key. Furthermore, it is easy to verify that, if we first decompose this relation into its projections onto SSN, Name, PhoneN and SSN, Name, Child and then join the projections, we get the original relation back. Thus, according to Section 6.9, this relation satisfies the following join dependency:

$$\text{PERSON} = (\text{SSN Name PhoneN}) \bowtie (\text{SSN Name Child})$$

In Section 6.9, we argued that relations that satisfy nontrivial join dependencies might contain a great deal of redundant information (in fact, much more than the amount of redundancy caused by the FDs). Since information redundancy is a cause of update anomalies, the relational design theory suggests that we should decompose the original relation into the following three:

PERSON	SSN	Name
	111-22-3333	Joe Public
	222-33-4444	Bob Public

PHONE	SSN	PhoneN
	111-22-3333	516-345-6789
	111-22-3333	516-123-4567
	222-33-4444	212-987-6543
	222-33-4444	212-135-7924

<sup>1</sup> For brevity, we omit some tuples needed to satisfy the foreign-key constraint.

CHILDOF	SSN	Child
	111-22-3333	222-33-4444
	111-22-3333	333-44-5555
	222-33-4444	444-55-6666
	222-33-4444	555-66-7777

While this decomposition certainly removes update anomalies, there are still difficulties. Consider the query *Get the phone numbers of all of Joe's grandchildren.* The SQL statement

---

```
SELECT G.PhoneN
FROM PERSON P, PERSON C, PERSON G
WHERE P.Name = 'Joe Public' AND
      P.Child = C.SSN AND
      C.Child = G.SSN
```

---

**16.1**

performs the query for the original schema, while the statement

---

```
SELECT N.PhoneN
FROM CHILDOF C, CHILDOF G,
      PERSON P, PHONE N
WHERE P.Name = 'Joe Public' AND
      P.SSN = C.SSN AND
      C.Child = G.SSN AND
      G.SSN = N.SSN
```

---

**16.2**

does the same for the decomposed schema. Both of these SQL expressions seem rather cumbersome implementations of the simple query we just stated in English.

One problem is that the redundancy in the original schema for PERSON is solely due to the inability of the relational data model to handle set-valued attributes in a natural way. A much more appropriate schema for the original table would be

---

```
PERSON(SSN: String, Name: String,
      PhoneN: {String}, Child: {String})
```

---

where the braces {} represent set-valued attributes. For instance, Child: {String} says that the value of the attribute Child in a tuple is a *set* of elements of type String. The rows in such a table might look as follows (note the set-valued components):

---

```
(111-22-3333, Joe Public,
 {516-123-4567, 516-345-6789}, {222-33-4444, 333-44-5555})
(222-33-4444, Bob Public,
 {212-987-1111, 212-987-6543}, {444-55-6666, 555-66-7777})
```

---

The second problem with the example above is the awkwardness with which SQL handles queries (16.1) and (16.2).

Suppose that the type of the attribute `Child` were `{PERSON}` rather than `{String}` and that SQL could treat the value of the `Child` attribute as a set of `PERSON` tuples (rather than just a set of strings that represent SSNs). It would then be possible to formulate the query much more concisely and naturally because the expression `P.Child.Child` can be given precise meaning: the set of all tuples corresponding to the children of the children of `P`. This would allow us to write the above query in the following elegant way:

---

```
SELECT P.Child.Child.PhoneN
FROM PERSON P
WHERE P.Name = 'Joe Public'
```

---

**16.3**

Expressions of the form `P.Child.Child.PhoneN` are called **path expressions**.

**IsA hierarchies.** Suppose that some but not all people in our database are students. Since a student is a person, we can represent this fact by drawing arrows in the corresponding E-R or UML diagrams. Since the relational model does not support the concept of IsA hierarchies, we would simulate it by factoring out the general information pertinent to all persons and have the schema for `STUDENT` contain only the information specific to students (see Section 4.5.3 for a discussion of the representation techniques for IsA):

---

```
STUDENT(SSN: String, Major: String)
```

---

Then we reason that, since a `STUDENT` is also a `PERSON`, every student has a `Name` attribute. Consider the query *Get the names of all computer science majors*, which we can try to write in SQL as follows:

---

```
SELECT S.Name
FROM STUDENT S
WHERE S.Major = 'CS'
```

---

Unfortunately, SQL-92 would reject the above query because the attribute `Name` is not explicitly included in the schema of `STUDENT`. However, if the system knew about the IsA relationship between students and persons, it could infer that `STUDENT` inherits `Name` from `PERSON`.

Although ISA hierarchies exist in the E-R and UML models, these models do not come with their own query languages. We are thus compelled to use the relational model and standard SQL, which forces us to write the following, more complex query:

---

```
SELECT P.Name
  FROM PERSON P, STUDENT S
 WHERE P.SSN = S.SSN AND S.Major = 'CS'
```

---

In essence, SQL-92 programmers must *explicitly* include an implementation of the ISA relationship with each query.

**Blobs.** The term “blob” means **binary large object**. Virtually all relational DBMSs allow relations to have attributes of type blob. For example, a database of movies can have this schema:

---

```
MOVIE (Name: String, Director: PERSON, Video: blob)
```

---

**16.4**

The attribute **Video** might hold a video stream, which can contain gigabytes of data. From the relational point of view, a video stream is a large, unstructured sequence of bits.

There are several problems with blobs. Consider the query

---

```
SELECT M.Director
  FROM MOVIE M
 WHERE M.Name = 'The Simpsons'
```

---

Some systems might drag the entire tuple containing the blob from disk into main memory to evaluate the WHERE clause. This is a huge overhead. Even when a DBMS is optimized to handle blobs, its options are limited. Suppose that we need only the frames in the range 20,000 to 50,000. We cannot obtain this information if we stay within the traditional relational model. To handle such a query, we need a special routine, `frameRange(from, to)`, perhaps implemented as a stored procedure, which, for a given video blob, returns frames in the specified range.

Would it make sense to add `frameRange()` as an operator to the relational data model? While this addition would enable anyone to play with video blobs, it would not help with blobs that store DNA sequences or VLSI chip designs. Thus, rather than burdening the data model with all kinds of specialized operations, a general mechanism is needed to let users define such operations separately for each type of blob.

**Objects.** The shortcomings of SQL, which we just discussed, led to the idea of databases that can store and retrieve objects. An object consists of a set of attributes

and a set of methods that can access those attributes, together with an associated inheritance hierarchy.

Attribute values can be instances of complex data types or other objects. For example, the attributes of a person object might include a complex data type representing the person's address and an object representing her spouse.

Since the value of an attribute can be an instance of an object, the operations defined for the object can be used in queries. Thus, a video object might have a method, `frameRange()`, which can be used as follows:

---

```
SELECT  M.frameRange(20000,50000)
FROM    MOVIE M
WHERE   M.Name = 'The Simpsons'
```

---

**Impedance mismatch in database languages.** It is impossible to write complete applications entirely in SQL, so database applications are typically written in a host language, such as C or Java, and they access databases by executing SQL queries embedded in a host program. Chapter 8 discussed a number of mechanisms for accessing databases from host languages.

One problem with this approach is that SQL is set oriented, meaning that its queries return sets of tuples. In contrast, C, Java, and other host languages do not understand relations and do not support high-level operations on them. Apart from this mismatch of types, there is a sharp difference between the declarative nature of SQL (which specifies *what* has to be done) and the procedural nature of host languages (in which the programmer must specify *how* things are to be done). This phenomenon has been dubbed the *impedance mismatch* between the data access language and the host language; the cursor mechanism (Section 8.2.4) was invented to serve as an adaptor between procedural host languages and SQL.

The problem of the impedance mismatch provided an important motivation for the development of the object-oriented data model, and one object database standard—the ODMG standard (developed by Object Database Management Group)—is designed to avoid this mismatch as much as possible. On the other hand, the impedance mismatch is inherent in the design of SQL, and its object-oriented *extensions* completely ignore this problem. We do not discuss ODMG further in this text, but it can be found in the full version of this book [Kifer et al. 2004].

### Object Databases versus Relational Databases

From the previous examples, we can begin to see the broad outlines of the object model and how it relates to the relational model.

- A relational database consists of relations, which are sets of tuples, while an object database consists of classes, which are sets of objects. Thus, a relational database might contain a relation, called PERSON, with tuples containing information about each person, whereas an object database might contain a class,

called PERSON, with objects containing information about each person. A particular relational database can be implemented within the object model by defining a class for each relation. The attributes of a particular class are the attributes of the corresponding relation, and each object instantiated from the class corresponds to a tuple.

- In a relational database, the components of a tuple must be primitive types (strings, integers, etc.); in an object database, the components of an object can, in addition, be complex types (sets, tuples, objects, etc.).
- Object databases have certain properties for which there is no analogy in relational databases:
  - Objects can be organized into an inheritance hierarchy, which allows objects of a lower type to inherit the attributes and methods from objects of a higher type. This helps reduce clutter in type specifications and leads to more concise queries.
  - Objects can have methods, which can be invoked from within queries. For instance, the specification of the class MOVIE mentioned earlier might contain a method, frameRange, with a declaration of the form

---

```
list(VIDEOFRAME)    frameRange(Integer, Integer);
```

---

which states that frameRange takes two integer arguments and returns a list of video frames. Such declarations are made using a special object definition language, which is similar to the data definition sublanguage of SQL.

- Method implementations are written in advance using a standard host language (e.g., C++ or Java) and stored on the server. In this respect, methods are similar to stored procedures in SQL databases (Section 8.2.5). However, stored procedures are not associated with any particular relation, while a stored method is an integral part of the respective class and is inherited along the object type hierarchy in a manner similar to that for methods in object-oriented programming languages.
- In some object database systems, the data manipulation language and the host language are the same.

## 16.2 The Conceptual Object Data Model

As in the case of the relational databases, we will first develop a conceptual view of a data model suitable for object databases. This model, the **Conceptual Object Data Model** (CODM), is derived from the work of the research team behind O<sub>2</sub> [Bancilhon et al. 1990]; it had significant influence on object-oriented database standards. In Section 16.2.4, we will introduce the object-relational extensions of SQL in terms of CODM.

In CODM, every object has a unique and immutable identity, called the **object Id (oid)**, which is independent of the actual value of the object. The oid is assigned

by the system when the object is created and does not change during the object's lifetime. Note the distinction between oids and the primary keys of relations. Like an oid, a primary key uniquely identifies the object. However, unlike an oid, the value of a primary key might change (a person might change her Social Security number). In addition, oids are normally hidden, while primary keys are visible and can be explicitly used in queries.

### 16.2.1 Objects and Values

An object that describes a person, Joe Public, might look as follows:

---

```
(#32, [ SSN: 111-22-3333,
      Name: Joe Public,
      PhoneN: {"516-123-4567", "516-345-6789"},           16.5
      Child: {"#445, #73}] )
```

---

The symbol #32 is the oid of the data object that describes a real-world Joe Public. The rest specifies the *value* part of the object. The oid identifies this object among other objects, and the value provides the actual information about Joe. Observe that the value of the Child attribute is a set of oids that (presumably) describe Joe's children.

Formally, an object is a pair of the form  $(oid, val)$ , where  $oid$  is an object Id and  $val$  is a value. The value part,  $val$ , can take one of the following forms:

- *Primitive value*. A member of an Integer, String, Float, or Boolean data type; example: "516-123-4567"  
Primitive values are not new to CODM—they also exist in the relational model.
- *Reference value*. An oid of an object; example: #445  
Reference values do not exist in the relational model, since it does not have a representation for complex objects.
- *Tuple value*. Of the form  $[A_1 : v_1, \dots, A_n : v_n]$ , where the  $A_1, \dots, A_n$  are distinct attribute names and the  $v_1, \dots, v_n$  are values; example: the entire value part (inside the brackets) of object #32 in (16.5)  
Tuple values exist in the relational model also. However, they can occur only at the top level, as rows of relations. In CODM, tuple values can appear at any level. For instance, they can occur as components of top-level rows.
- *Set value*. Of the form  $\{v_1, \dots, v_n\}$ , where the  $v_1, \dots, v_n$  are values; examples: {"516-123-4567", "516-345-6789"} or {"#445, #73}  
Set values do not exist in the relational model except in the sense that relations are sets.

Thus, in addition to the objects of the form (16.5), other (perhaps less obvious) examples of objects are (#38, "Joe Average")—because "Joe Average" is a primitive value, (#77, #534)—because #534 is a reference value, and (#47, {"#987, #34})—

because `{#987, #34}` is a set value. Reference, tuple, and set values are called **complex values** to distinguish them from primitive values.

Note that the oid part of an object cannot change (if it did, it would indicate a different object). In contrast, the value part of an object can be replaced by another value as a result of an update. For example, if one of Joe Public's phone numbers should change, the value of the `PhoneN` attribute is replaced by a new value, but the object retains the same oid and is considered to be the same object.

The oid of an object cannot change, but the value can.

### 16.2.2 Classes

In object-oriented systems, semantically similar objects are organized into **classes**. For instance, all objects representing persons are grouped into class `PERSON`.

Classes play the same role in CODM that relations play in relational databases. Whereas in SQL-92 a database is a set of relations and each relation is a set of tuples, in CODM a database is a set of classes and each class is a set of objects. Thus, in SQL-92 we might have a relation called `PERSON` with tuples containing information about each person, and in CODM we might have a class called `PERSON` with objects containing information about each person. Note that we can always convert a relational database into an object database by attaching a unique oid to each tuple.

Classes help organize objects into categories. A class has a **type**, which describes the common structure of all objects in the class (e.g., all objects in a class might be sets of tuples), and **method signatures**, which are declarations of the operations that can be applied to the class objects. We discuss these notions in more detail below. Only method signatures are part of CODM—method implementations are *not*. A method implementation is a procedure, written in a host language, that is stored on the database server. An ODBMS must provide a mechanism to invoke the appropriate implementation whenever the method is used in the program.

In the relational data model, two tables can be related to each other by means of an interrelational constraint (e.g., a foreign-key constraint). In the object data model, one additional relationship—the *IsA relationship*—enjoys a special status. Suppose that, in addition to the `PERSON` class, which groups together all objects representing persons, we have a `STUDENT` class, which groups together all objects representing students. Naturally, every student is a person, so the set of objects that constitute class `STUDENT` must be a subset of the set of objects that constitute class `PERSON`. This is an example of the **subclass relationship**, in which `STUDENT` is a subclass of `PERSON`. The subclass relationship is also called the *IsA relationship*.

The set of all objects assigned to a class is called the **extent** of the class. To adequately reflect our intuition about the subclass relationship, extents must satisfy the following property:

If  $C_1$  is a subclass of  $C_2$ , the extent of  $C_2$  contains the extent of  $C_1$ .

For example, since STUDENT is a subclass of PERSON, the set of all students is a subset of the set of all persons.

The query language and the data manipulation languages are aware of the subclass relationship. For example, if a query is supposed to return all PERSON objects that have a certain property and if some STUDENT object has that property, the query will return the STUDENT object in the query result since every student is also a person.

To summarize, a class has a type (which describes the class structure), method signatures (which are often considered part of the type), and an extent (which lists all objects that belong to the class). Thus

Class, type, and extent are related, but distinct, notions.

### 16.2.3 Types

An important requirement of any data model is that the data must be properly structured. Because of the simplicity of the structure of tuples in the relational model, typing is not a big issue in relational databases. It is more complex in object databases. Consider, for example, the object in (16.5). We can say that its type—let us call it PERSON—is represented by the following expression:

---

[SSN: String, Name: String, PhoneN: {String}, Child: {PERSON}] **16.6**

---

This type definition states that the attributes SSN and Name draw their values from the primitive domain String; the attribute PhoneN must have values that are sets of strings; and the values of the attribute Child are sets of PERSON objects.

Intuitively, the type of an object is just the collection of the types of its components. More precisely, complex types suitable for structuring objects can be defined as follows:

- *Basic types.* String, Float, Integer, and so forth
- *Reference types.* User-defined class names, such as PERSON and STUDENT
- *Tuple types.* Expressions of the form  $[A_1 : T_1, \dots, A_n : T_n]$ , where each  $A_i$  is a distinct attribute name and  $T_i$  is a type. The type given in (16.6) is an example of a tuple type.
- *Set types.* Expressions of the form  $\{T\}$ , where  $T$  is a type. For example,  $\{\text{String}\}$  is a set type.

Note that (16.6) describes a type in which complex structures are nested within other structures. For instance, the values of PhoneN are sets of primitive values, while the values of Child are sets of objects of type PERSON.

What does it mean for an object to conform to a type? Given the recursive structure of objects and the presence of the ISA hierarchy, the answer to this question is not straightforward. We develop this concept in the following paragraphs.

**Subtyping.** In addition to grouping objects structurally, the type system can tell which types have “more structure” than others. For instance, suppose the type of the PERSON objects is

---

```
[SSN: String, Name: String,
Address: [StNumber: Integer, StName: String]]
```

---

This is a tuple type, in which the first two components have a basic type and the third has a tuple type.

Consider now the objects of type STUDENT. Clearly, students have names, addresses, and everything else that PERSON objects have. However, students have additional attributes, so an appropriate type might be

---

```
[SSN: String, Name: String,
Address: [StNumber: Integer, StName: String],
Majors: {String}, Enrolled: {COURSE}]
```

---

Our intuition suggests that type STUDENT has more structure than type PERSON because (1) it has all the attributes of PERSON, (2) the values of these attributes have at least as much structure as the corresponding attributes in PERSON, and (3) STUDENT has attributes not present in PERSON. This intuition leads to the notions of **subtype** and **supertype**: Type  $T$  is a *subtype* of (supertype)  $T'$  if  $T \neq T'$  and one of the following conditions holds:

- $T$  and  $T'$  are reference types, and  $T$  is a subclass of  $T'$ .
- $T = [A_1 : T_1, \dots, A_n : T_n, A_{n+1} : T_{n+1}, \dots, A_m : T_m]$  and  $T' = [A_1 : T'_1, \dots, A_n : T'_n]$  are tuple types (note that  $T$  includes all attributes of  $T'$ , that is,  $m \geq n$ ), and either  $T_i = T'_i$  or  $T_i$  is a subtype of  $T'_i$ , for each  $i = 1, \dots, n$ .
- $T = \{T_0\}$  and  $T' = \{T'_0\}$  are set types, and  $T_0$  is a subtype of  $T'_0$ .

According to this definition, STUDENT is a subtype of PERSON because it contains all of the structure defined for PERSON and has additional attributes of its own. Note, however, that having additional attributes is not necessary for a type to be a subtype. For instance,

---

```
[SSN: String, Name: String,
Address: [StNumber: Integer, StName: String, POBox: String]]
```

16.7


---

is still a subtype of PERSON even though it does not have attributes beyond those defined for PERSON. Instead, the attribute Address in (16.7) has more structure than the same attribute in PERSON.

**Domain of a type.** The **domain** of a type is a set of all values that conform to that type. Intuitively the domain of a type,  $T$ , denoted  $\text{domain}(T)$ , is the appropriate combination of the domains of the components of  $T$ . More precisely,

- The domain of a basic type, such as `Integer` or `String`, is just what we would expect—the set of all integers or strings, respectively.
- The domain of a reference type,  $T$ , is the extent of  $T$ , that is, the set of all Ids of objects in class  $T$ . For instance, if  $T$  is `PERSON`, the domain is the set of all oids of `PERSON` objects.
- The domain of a tuple type,  $[A_1 : T_1, \dots, A_n : T_n]$ , is

$$\{[A_1 : w_1, \dots, A_n : w_n] \mid w_i \in \text{domain}(T_i)\}$$

that is, the set of all tuple values whose components conform to the corresponding types of attributes. Thus, the domain of type (16.6) is the set of all values of the form in (16.5).

- The domain of a set type,  $\{T\}$ , is

$$\{\{w_1, \dots, w_k\} \mid w_i \in \text{domain}(T)\}$$

That is, it consists of *finite* sets of values that conform to the given type  $T$ . For instance, the domain of type `{COURSE}` is the set whose members are finite sets of oids for `COURSE` objects.

*Brain Teaser:* Is it useful to allow infinite sets as elements of the domain for a set type?

It is easy to see that domains are defined in such a way that the domain of a subtype,  $S$ , is (in a sense) a subset of the domain of a supertype,  $S'$ . More precisely, for any given object,  $o$ , in  $S$ , either  $o$  is already in  $S'$  or we can use  $o$  to obtain another object,  $o'$ , in  $S'$ , by throwing out some components of  $o$  or of sub-objects included in  $o$ . For instance, a `PERSON` object can be obtained from a `STUDENT` object by throwing out the components corresponding to the `Majors` and `Enrolled` attributes. Similarly, a `PERSON` object can be obtained from an object of type (16.7) by throwing out the `POBox` component from the nested address.

*Brain Teaser:* What is the domain of the tuple type  $[ ]$ , which has no attributes?

**Database schema and instance.** In object databases, the **schema** contains the specification for each class of objects that can be stored in the database. For each class,  $C$ , it includes

- The *type* associated with  $C$ . This type determines the structure of each object of  $C$ .

- The *method signatures* of C. A **method signature** specifies the method name, the type and order for the allowed method arguments, and the type of the result produced by the method. For instance, the method `enroll()` in class COURSE might have the following signature:

---

```
Boolean enroll (STUDENT);
```

---

and the method `enrolled()` in class STUDENT might have the signature

---

```
{COURSE} enrolled ();
```

---

The signature of `enroll()` says that in order to enroll a student in a course, one must invoke the method `enroll()` on the COURSE-object and supply the STUDENT-object as a parameter. The method returns a Boolean value that indicates the outcome of the operation. The signature of `enrolled()` says that, to check the enrollment of a student, one can invoke the method `enrolled()` in the context of the STUDENT-object corresponding to that student, and the result will be a set of COURSE-objects corresponding to courses in which the student is enrolled.

- The *subclass-of* relationship, which identifies the superclasses of C
- The *integrity constraints*, such as key constraints, referential constraints, or more general assertions, which are similar to constraints in relational databases

An **instance** of the database is a set of objects for the classes specified in the schema. The objects must satisfy all of the constraints implied by the schema, which includes type constraints. Thus, the value of each object must belong to the domain of the type associated with the object's class. Each object must also have a unique oid.

**Brain Teaser:** What domains must an object belong to if it is a member of several classes?

This completes the definition of the conceptual object model. As you can see, most of the notions used in CODM are extensions of familiar concepts from the relational model, but they require considerably more care because of the richness of the underlying data model.

#### 16.2.4 Object-Relational Databases

This breed of DBMS arrived in the early 1990s, when a number of vendors began advocating object-relational DBMS (instead of full-blown object databases) as a safer migration path from relational DBMS. The main selling point was that such databases could be implemented as conservative extensions to the existing relational DBMSs. After long deliberation, the SQL:1999 working group finally adopted a

subset of the object-relational data model. SQL:2003 gave support for the full object-relational submodel of CODM.

An **object-relational database** consists of a set of top-level classes, which are populated by *tuple objects*. A **tuple object** is of the form  $(oid, val)$ , where  $oid$  is an object Id and  $val$  is a tuple value whose components can be *arbitrary values* (i.e., primitive values, sets, tuples, and references to other objects).

Since the top-level structure of the top-level classes is a tuple, these classes are called *relations*, which explains the term “object-relational.”

The main difference between the object-relational and CODM models is that in the former, the top-level structure of each object instance is always a tuple while in the latter, the top-level structure can be an arbitrary value. However, this restriction on object-relational DBMSs does not significantly decrease the ability to model real-world enterprises.

What differentiates object-relational and traditional relational models is that the tuple components must be primitive values in the relational model whereas they can be arbitrary values in the object-relational model. Thus, the relational model can be viewed as a subset of the object-relational model (and hence of CODM).

We discuss the object-relational model underlying SQL:1999/2003 in the next section.

## 16.3 Objects in SQL:1999 and SQL:2003

Object-oriented extensions in SQL have gone through many revisions. The final result is a reasonably clean version of the object-relational model. It was a difficult standardization process, given the requirement to preserve backward compatibility with SQL-92—a language *not* designed with objects in mind.

The goals of this backward compatibility were that SQL:1999/2003 could be used in any of the following ways:

- To work with standard SQL-92 relations
- To work with relations that are similar to those in SQL-92 except that attributes can have values of complex user-defined types (such as sets or tuples)

In this section, we survey the new object-relational extensions of SQL:1999 and 2003. SQL:1999 is described in [Gulutzan and Pelzer 1999]. SQL:2003 is available through the standards organizations, such as ISO (<http://www.iso.org/>).

An SQL:1999/2003 database consists of a set of relations. Each relation is either a set of tuples or a set of objects. An **SQL object** is a pair of the form  $(o, v)$ , where  $o$  is an oid and  $v$  is an SQL tuple value. An **SQL tuple value** has the form  $[A_1 : v_1, \dots, A_n : v_n]$ , where  $A_1, \dots, A_n$  are distinct attribute names and each  $v_i$  takes one of the following values (using the terms introduced in Section 16.2.1):

- *Primitive value.* Constants of the usual SQL primitive types, such as CHAR(18), INTEGER, DECIMAL, and BOOLEAN
- *Reference value.* Object Ids

- *Tuple value.* Of the form  $[A_1 : v_1, \dots, A_n : v_n]$ , where each  $A_i$  is a distinct attribute name and each  $v_i$  is a value
- *Collection value.* Created using the MULTISET construct. It is the only major object-oriented addition introduced by SQL:2003. (SQL:1999 also provides the ARRAY construct, but it is only of marginal interest and will not be discussed in this book.)

As expected of a data model in the object-relational mold, the top-level value of every object in SQL:1999/2003 is a tuple. Tuples and sets can be nested, however.

### 16.3.1 Row Types

The simplest way to construct a tuple type is with the **ROW type constructor**. For instance, we can define the relation PERSON as follows:

---

```
CREATE TABLE PERSON (
    Name CHAR(20),
    Address ROW(Number INTEGER, Street CHAR(20), ZIP CHAR(5)) )
```

---

We reference the components of a row type using the usual mechanism of path expressions.

---

```
SELECT P.Name
FROM PERSON P
WHERE P.Address.ZIP = '11794'
```

---

A table with row types can be populated with the help of the **ROW value constructor** as follows:

---

```
INSERT INTO PERSON(Name, Address)
VALUES ('John Doe', ROW(666, 'Hollow Rd.', '66666'))
```

---

Updating tables that have attributes of type **ROW** is also straightforward.

---

```
UPDATE PERSON
SET Address.ZIP = '12345'
WHERE Address.ZIP = '66666'
```

---

When John Doe moves, we can change the entire address as follows:

---

```
UPDATE PERSON
SET Address = ROW(21, 'Main St.', '12345')
WHERE Address = ROW(666, 'Hollow Rd.', '66666')
    AND Name = 'John Doe'
```

---

### 16.3.2 User-Defined Types

Recall from Section 16.2.3 that a type (in CODM) is a set of rules for structuring data. The set of objects that conform to these rules is the type's domain. A class consists of a schema (which includes the type and method signatures) and an extent—a subset of the domain of the type. When we add method bodies to the signatures associated with a type, we get an **abstract data type**. In SQL:1999/2003, abstract data types are called **user-defined types** (or UDT). The following are examples of UDT definitions:

---

```

CREATE TYPE PERSONTYPE AS (
    Name CHAR(20),
    Address ROW(Number INTEGER, Street CHAR(20), ZIP CHAR(5)));
CREATE TYPE STUDENTTYPE UNDER PERSONTYPE AS (
    Id INTEGER,
    Status CHAR(2)
METHOD award_degree() RETURNS BOOLEAN;
CREATE METHOD award_degree() FOR STUDENTTYPE
LANGUAGE C
EXTERNAL NAME 'file:/home/admin/award_degree';

```

---

The first CREATE TYPE statement is syntactically similar to the earlier definition of table PERSON, except that now we define a type rather than a table. This type does not have any explicitly defined methods, but we will soon see that the DBMS automatically creates a number of methods for us.

The second statement is more interesting. It defines STUDENTTYPE as a subtype of PERSONTYPE, which is indicated with the clause UNDER. As such, it inherits the attributes of PERSONTYPE. In addition, STUDENTTYPE is defined to have attributes of its own plus a method, award\_degree(). The type definition includes only the signature of the method. The actual definition is done using the CREATE METHOD statement (which is associated with STUDENTTYPE through the FOR clause). The statement says that the method body is written in the C language (so that the DBMS knows how to link with this procedure) and tells where its executable can be found. If we specified LANGUAGE SQL instead, we could have defined the method code inside an attached BEGIN-END block using SQL/PSM, the language of stored procedures (see Chapter 8).

User-defined types can appear in two main contexts. First, they can be used to specify the domain of an attribute in a table, just like the primitive types of integers or character strings:

---

```

CREATE TABLE TRANSCRIPT (
    Student STUDENTTYPE, -- a previously defined UDT
    CrsCode CHAR(6),
    Semester CHAR(6),
    Grade CHAR(1)

```

---

Here we are using the CREATE TABLE statement with the only difference that some attributes (*Student*) have complex types (STUDENTTYPE).

Second, a UDT can be used to specify the type of an entire table. This is done through a new kind of CREATE TABLE statement, which, instead of enumerating the columns of a table, simply provides a UDT. This means that all rows of the table must have the structure specified by the UDT. For instance, we can define the following table based on the previously defined UDT STUDENTTYPE:

---

```
CREATE TABLE STUDENT OF STUDENTTYPE;
```

---

**16.9**

Tables constructed via the CREATE TABLE ... OF statement, as in (16.9), are called **typed tables**. The rows of a typed table are considered to be **objects**. Thus the rows of the table in (16.9) are objects, while the values of the *Student* attribute in (16.8) are *not*—even though the same UDT is used in both cases. These two distinct uses of UDTs are discussed next.

### 16.3.3 Objects

The only way to create an object in SQL is to insert a row into a typed table. In other words, every row in such a table is treated as an object with its own oid. The table itself is then viewed as a class (as defined in CODM), and its set of rows corresponds to the extent of the class.

It is instructive to compare (16.9) with the following declaration:

---

```
CREATE TABLE STUDENT1 (
    Name CHAR(20),
    Address ROW(Number INTEGER, Street CHAR(20), ZIP CHAR(5)),
    Id INTEGER,
    Status CHAR(2) )
```

---

Note that STUDENT1 contains exactly the same attributes as STUDENT—both names and types. However, STUDENT is a typed table whereas STUDENT1 is not, which means that SQL considers the tuples of STUDENT—but not the tuples of STUDENT1—to be objects. This disparity (one may even say inconsistency) between the two ways of constructing tables is solely due to the need to stay backward-compatible with SQL-92. Note also the difference in the use of STUDENTTYPE in (16.8) and (16.9). Instances of STUDENTTYPE in (16.8) are *not* objects, while they *are* objects in (16.9).

The next question concerns how we refer to an object. To understand the issue, let us come back to the TRANSCRIPT table in (16.8) and consider the attribute *Student*. Since the same student is likely to have taken several courses, he has several tuples in TRANSCRIPT. The trouble is that the declaration

---

```
Student STUDENTTYPE
```

---

means that the value of this attribute in a row of TRANSCRIPT is *not* a reference to a STUDENTTYPE object. Thus, information about every student (name, address, etc.) must be duplicated in each transcript record for that student. Clearly, this is the same redundancy we tried to eliminate in Chapter 6 using the relational normalization theory. SQL solves the problem by introducing the explicit **reference type**, denoted REF(STUDENTTYPE), which we will discuss further in Section 16.3.6. For now, we have to remember that the domain of a reference type is a set of oids. To reference an object in SQL, we need to obtain its oid, and so we have to look at the mechanism provided for this purpose.

The SQL:1999/2003 standard says that every typed table, such as (16.9), has a **self-referencing column**. For each tuple, this column holds the oids of that tuple (hence the name “self-referencing”). The oid is generated automatically when the tuple is created. However, to gain access to the oids stored in the self-referencing column, we have to give the column a name explicitly. The declaration of STUDENTTYPE above does not name the self-referencing column, thus there is no way to refer to the oids of the objects in that table.

Here is a way to take care of the self-referencing column:

---

```
CREATE TABLE STUDENT2 OF STUDENTTYPE
REF IS stud_oid;
```

---

**16.10**

The REF IS clause gives an explicit name, `stud_oid`, to the self-referencing column. (Note that this column also exists in (16.9) but is unnamed and hence cannot be referenced.) For most purposes, `stud_oid` is an attribute like any other. In particular, we can use it in queries (in both SELECT and WHERE clauses), but we cannot change its value because oids are assigned by the system and are immutable.

The distinction between objects and their references, as manifested by the reference types and self-referencing columns, is one of the muddier aspects of the object-relational extensions of SQL. This complication exists thanks to the undue influence of the C and C++ languages and also because the object extensions were tacked onto SQL as an afterthought. Note that such a distinction does not exist in Java, which is a true object-oriented language, or in the ODMG standard, which is discussed in the full version of this book [Kifer et al. 2004].

### 16.3.4 Querying User-Defined Types

Querying UDTs does not present any new problems. We can simply use path expressions to descend into the objects and extract the needed information. For instance,

---

```
SELECT T.Student.Name, T.Grade
FROM TRANSCRIPT T
WHERE T.Student.Address.Street = 'Hollow Rd.'
```

---

**16.11**

queries the TRANSCRIPT relation and returns the names and grades of the students who live on Hollow Road. Note that T.Student returns complex values of type

`STUDENTTYPE`, and inheritance from `PERSONTYPE` allows us to access the `Name` and `Address` attributes defined for it.

Note also that although `STUDENT` and `STUDENT1` are defined differently, queries concerning students look identical in both cases. Thus,

---

```
SELECT S.Address.Street
FROM X S
WHERE S.Id = '111111111'
```

---

returns the street name of a student with Id 111111111 regardless of whether `X` is `STUDENT` or `STUDENT1`.

### 16.3.5 Updating User-Defined Types

Having discussed the data definition aspects of UDTs, we turn to the issue of populating relations based on these UDTs. We have already seen in Section 16.3.1 how to insert tuples into the `PERSON` table. By analogy, we can use the same method to insert tuples into the tables `STUDENT` and `STUDENT2`. The fact that these relations contain objects (and the extra self-referencing attribute) does not matter because the oids are generated by the system. We have to worry about only the actual attributes. We might try a similar `INSERT` statement to populate the relation `TRANSCRIPT`:

---

```
INSERT INTO TRANSCRIPT(Student, Course, Semester, Grade)
VALUES (????, 'CS308', '2000', 'A')
```

---

But what should appear as the first component of the `VALUES` clause? There are two answers to this question. One is discussed later in this section, and the other in Section 16.3.6.

Insertion is further complicated by the fact that a UDT is considered to be **encapsulated**, that is, its components can be accessed only through the methods provided by the type. Although we did not define any methods for `STUDENTTYPE`, the DBMS did it for us. Namely, for each attribute the system provides an **observer method**, which can be used to query the attribute value, and a **mutator method**, which is used to change that value. Both the observer and the mutator have the same name as the attribute. In the case of `STUDENTTYPE`, the system provides the following observer methods:

- **Id:**  $() \rightarrow \text{INTEGER}$ . This method returns an integer and, like all observers, it does not take any arguments.<sup>2</sup>
- **Name** and **Status**: These methods have the types  $() \rightarrow \text{CHAR}(20)$  and  $() \rightarrow \text{CHAR}(2)$ , respectively.
- **Address:**  $() \rightarrow \text{ROW}(\text{INTEGER}, \text{CHAR}(20), \text{CHAR}(5))$ .

<sup>2</sup>The notation  $()$  indicates that the method takes no arguments.

Looking back at query (16.11) we can now say that it uses the observer methods `Name` and `Address`. On the other hand, the `Grade` attribute of the table `TRANSCRIPT` used in that same query is not part of a UDT, so it does not use an observer method. However, the difference is conceptual and not syntactic—syntactically we reference `Grade` in the same way we do `Name`.

The mutator methods are called in the context of a `STUDENTTYPE` object and return this same object. For instance, the mutator for an attribute, say, `Id`, takes a value for `Id` and returns the original object with the changed value of the `Id` attribute.

- `Id: INTEGER → STUDENTTYPE`. This method takes an integer and replaces the value of `Id` of the object with that integer. In other words, this mutator method changes the student `Id` and returns the modified object.
- `Name: CHAR(20) → STUDENTTYPE`. This method takes a string and replaces the value of the `Name` attribute in the student object. It returns the updated student object. The mutator for `Status` is similar.
- `Address: ROW(INTEGER, CHAR(20), CHAR(5)) → STUDENTTYPE`. This method takes a row that represents an address and replaces the student address with it.

*Brain Teaser:* Why does a mutator, such as `Id()`, return the entire object rather than just the new value for the `Id` attribute?

Note that SQL does not have the `public` and `private` specifiers of C++ and Java to control access to methods. Instead, access is controlled through the `EXECUTE` privilege and the usual `GRANT/REVOKE` mechanism introduced in Section 3.3.

We are now ready for our first insertion into a UDT:

---

```
INSERT INTO TRANSCRIPT(Student, Course, Semester, Grade)
VALUES (NEW StudentType()
        .Id(666666666)
        .Status('G5')
        .Name('Vlad Dracula')
        .Address(ROW(666,'Transylvania Ave.', '66666')),  
        'HIS666',
        'F1462',
        'D')
```

---

**16.12**

Two things should be noted here. A blank student object in the first component of the inserted tuple is created by a call to `StudentType()`—a default constructor that the DBMS creates for every UDT. Then the mutator methods are invoked one by one on the newly created object to fill it in with data.<sup>3</sup>

<sup>3</sup> Note that the syntax above is just an indented and more readable form of `NEW StudentType().Id(...).Status(...).Name(...).Address(...)`.

If the student's address, name, and grade are to be changed, we can use the following update statement:

---

```
UPDATE TRANSCRIPT
SET Student = Student
    .Address(ROW(21,'Main St.', '12345'))
    .Name('John Smith'),
Grade = 'A'
WHERE Student.Id = 666666666
    AND CrsCode = 'HIS666' AND Semester = 'F1462'
```

---

To change the value of the student object, we use the mutator methods for STUDENTTYPE, which are generated for us by the DBMS. First, we apply the `Address()` mutator to change the address and then the `Name()` mutator. In contrast, since the type of the `Grade` attribute is primitive, the grade is changed by a direct assignment.

You have certainly noticed that inserting new tuples into relations that involve UDTs is rather cumbersome. However, the ability to associate methods with complex data types can simplify this to some extent. Namely, we can define a special constructor method that takes only scalar values. In this way, a complex object can be created in one call to the constructor.

We illustrate the idea using the language of SQL stored procedures. First, we need to add the following declaration to our earlier definition of STUDENTTYPE:

---

```
ALTER TYPE StudentType
ADD METHOD StudentConstr(name CHAR(20), id INTEGER,
                           streetNumber INTEGER,
                           streetName CHAR(20),
                           zip CHAR(5), status CHAR(2))
RETURNS STUDENTTYPE;
```

---

Then we define the body of the method as follows:

---

```
CREATE METHOD StudentConstr(name CHAR(20), id INTEGER,
                           streetNumber INTEGER,
                           streetName CHAR(20),
                           zip CHAR(5), status CHAR(2))
FOR STUDENTTYPE
RETURNS STUDENTTYPE
LANGUAGE SQL
BEGIN
    RETURN NEW STUDENTTYPE()
        .Name(name)
        .Id(id)
```

```

    .Status(status)
    .Address(ROW(streetNumber,streetName,zip));
END;

```

With this new constructor, the insertion of a new tuple into the TRANSCRIPT relation corresponding to (16.12) becomes less of a chore:

---

```

INSERT INTO TRANSCRIPT(Student, Course, Semester, Grade)
VALUES (StudentConstr('Vlad Dracula', 666666666, 666,
                      'Transylvania Ave.', '66666', 'G5'),
        'HIS666',
        'F1462',
        'D')

```

---

### 16.3.6 Reference Types

In schema (16.8) for the TRANSCRIPT relation, the attribute Student has the type STUDENTTYPE. As explained in Section 16.3.3, this prevents sharing of student objects because every student object is physically stored inside the corresponding transcript tuple. To enable object sharing, SQL uses **reference data types**. A reference is an oid, and the domain of a type of the form REF(SomeUDT) consists of all of the oids of objects of type *SomeUDT*. With this feature, we can rewrite our definition of TRANSCRIPT in (16.8) as follows:

---

```

CREATE TABLE TRANSCRIPT1 (
    Student REF(STUDENTTYPE) SCOPE STUDENT2,
    CrsCode CHAR(6),
    Semester CHAR(6),
    Grade CHAR(1) )

```

16.13

---

The type of the Student attribute needs more explanation. First, the type REF(STUDENTTYPE) means that the value of Student must be an oid of an object of type STUDENTTYPE. However, we can create many different tables and associate them with STUDENTTYPE. Each such table can contain all kinds of students. We might not want Student to refer to just *any* student. Instead, we want some kind of referential integrity that ensures that this attribute refers to students described by a *particular* table. The clause SCOPE achieves just that by requiring that the value of Student be not just any oid of type STUDENTTYPE but one that belongs to an existing object in the table STUDENT2. To be consistent, the scope, such as the STUDENT2 relation, must have the type mentioned in REF. In our example, STUDENT2 is of type STUDENTTYPE, which is consistent with the type REF(STUDENTTYPE) of the attribute Student.

Stay tuned: the use of STUDENT2 rather than STUDENT in (16.10) is not accidental. We will come back to discuss this issue later in this section.

**Querying reference types.** “Misfeatures” often come on the heels of new features. Here is how query (16.11) looks when applied to the table TRANSCRIPT1, defined in (16.13):

---

```
SELECT T.Student->Name, T.Grade
FROM TRANSCRIPT1 T
WHERE T.Student->Address.Street = 'Hollow Rd.'
```

---

Recall that T.Student returns the Id of an object of type STUDENTTYPE. Observe the use of the symbol  $\rightarrow$  to refer to the attributes of that type. This means that the syntax for accessing the attributes of an object depends on whether the object is given by its oid or its value. The unfortunate distinction between references by  $.$  and  $\rightarrow$  is the disease that SQL contracted from C and C++. Such a distinction is not necessary in an object-oriented language, and it does not exist in Java or in ODMG databases.

The rule for deciding whether to use  $.$  or  $\rightarrow$  is the same as in C and C++. If an attribute has a reference type, then  $\rightarrow$  is used in path expressions; if it has an object type, then  $.$  is used. In our case, T.Student has a reference type, REF(STUDENTTYPE), so we use T.Student  $\rightarrow$  Address to access the attributes of the student object. In contrast, the earlier query (16.11) uses T.Student.Address because there T.Student has the type STUDENTTYPE, which is not a reference type.

**Creating tuples that contain reference types.** The next important question is how to populate the table TRANSCRIPT1. In Section 16.3.5, we saw examples of tuple insertion into complex types. However, in those cases we did not deal with object references. In order to insert a tuple into TRANSCRIPT1, we must find a way to access oids of student objects and assign them to the attribute Student. This is where the self-referencing column, introduced in Section 16.3.3, comes in handy. Recall that table STUDENT2 defined in that section has a self-referencing column, called stud\_oid.<sup>4</sup> Recall also that the table STUDENT in (16.9) has the same type as STUDENT2, except that its self-referencing column is unnamed. Thus, we cannot obtain oids of the tuples in STUDENT and assign them as values of the Student attribute in the table TRANSCRIPT1. Because of this handicap, we have to use STUDENT2 instead of STUDENT in the definition of TRANSCRIPT1.

Assuming that the Id attribute in STUDENT2 is a key, we can now insert a student into TRANSCRIPT1 as follows:

---

```
INSERT INTO TRANSCRIPT1(Student, Course, Semester, Grade)
SELECT S.stud_oid, 'HIS666', 'F1462', 'D'
FROM STUDENT2 S
WHERE S.Id = '666666666'
```

---

<sup>4</sup> Note that the self-referencing attribute stud\_oid has nothing to do with the attribute Id. The latter is a regular attribute of STUDENTTYPE whose value is set explicitly by the programmer.

Observe that we use the **SELECT** statement to retrieve the oid of the desired student object (`S.stud_oid`). This oid becomes the value for the **Student** attribute. The values for the attributes **Course**, **Semester**, and **Grade** are simply tacked on to the target list of the **SELECT** statement.

### 16.3.7 Inheritance

Recall that **STUDENTTYPE** was defined in Section 16.3.2 as being **UNDER** (i.e., as a subtype) of **PERSONTYPE**. This means that although the attributes of **PERSONTYPE**, **Name** and **Address**, are not explicitly defined in **STUDENTTYPE**, they are nevertheless applicable to the rows of the tables that have the type **STUDENTTYPE**.

Let **STUDENT** be a table of type **STUDENTTYPE** and **PERSON** a table of type **PERSONTYPE**. Suppose we insert a tuple into the **STUDENT** table as follows:

---

```
INSERT INTO STUDENT(Name, Address, Id, Status)
VALUES ('John Jones', ROW(123,'Main St.'), 111222333, 'G2')
```

---

Will this tuple automatically show up in the relation **PERSON**? The answer is *no*. The reason is actually quite logical: one can define several tables using the same UDT **PERSONTYPE**. Should the above **STUDENT** tuple show up in all such tables or only in some? The designers of SQL decided that it should show up in only some tables—those that the user explicitly marked as **supertables** (by analogy with superclasses) of the table **STUDENT**. Supertables are specified with the same keyword, **UNDER**, but this time it applies to tables rather than types:

---

```
CREATE TABLE STUDENT OF STUDENTTYPE UNDER PERSON
```

---

With this declaration, **PERSON** becomes a supertype of the table **STUDENT**, and all tuples inserted into **STUDENT** will automatically show up in **PERSON** (with the attributes that do not belong to **PERSONTYPE** removed).

To summarize, in order for a table, *T<sub>1</sub>*, to be a subtable of another table, *T<sub>2</sub>*, the following must hold:

1. The UDT of *T<sub>1</sub>* must be a subtype of (defined as being **UNDER**) the UDT of *T<sub>2</sub>*; and
2. The table *T<sub>1</sub>* must be defined as being **UNDER** the table *T<sub>2</sub>*.

**Brain Teaser:** What will the value of the attributes **Name** and **Address** be if we insert a tuple into **STUDENT** using only the proper attributes of the **STUDENTTYPE** data type: `INSERT INTO STUDENT(Id, Status) VALUES (111222333, 'G2')`?

### 16.3.8 Collection Types

SQL:2003 introduced the MULTISET collection type, thereby making the SQL data model fully object-relational. A **multiset** collection type is like a set except that the same element can occur in the collection more than once. This is in accordance with the default SQL strategy of retaining duplicate tuples, which makes query results into multisets (see Section 5.2). To illustrate the new collection type, we will add a new set-valued attribute, `Enrolled`, to `STUDENTTYPE`:

---

```
CREATE TYPE STUDENTTYPE UNDER PERSONTYPE AS (
    Id INTEGER,
    Status CHAR(2),
    Enrolled REF(COURSETYPE) MULTISET
)
```

---

In this example, the value of the attribute `Enrolled` must be a (multi)set of oids of tuples of the type `COURSETYPE`. We assume that `COURSETYPE` is a new data type that has attributes `CrsCode`, `Name`, and `Description`.

With multiset collection types, a new kind of query becomes possible in which multiset-valued path expressions can act as table expressions in the `FROM` clause. This is analogous to `SELECT` statements in the `FROM` clause, which is permitted in SQL-92. We illustrate the use of multisets with a query that returns all tuples of the form  $(i, n)$ , where the student with `Id i` took the course with name `n`.

---

```
SELECT S.Id, C.Name
  FROM STUDENT S, COURSE C
 WHERE C.CrsCode IN
      ( SELECT E->TmpCrsCode
        FROM UNNEST(S.Enrolled) AS TMPCOURSE(TmpCrsCode) E)
```

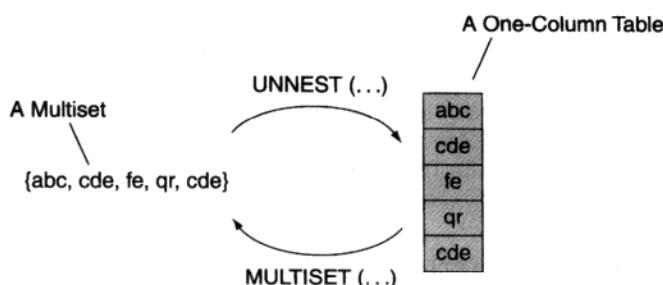
---

We assume that the relation `STUDENT` is of type `STUDENTTYPE` and `COURSES` of type `COURSETYPE`. The `WHERE` clause tests each course object for whether student `S` is enrolled in it. The condition uses a nested `SELECT` statement to produce the set of course codes of the courses taken by student `S`.

The new feature here appears in the `FROM` clause of the nested `SELECT` statement. The range of the variable `E` is the set of references to courses specified by a path expression, `S.Enrolled`. It lists the courses in which `S` is enrolled. This set is not a table, however, since the elements of this set are object IDs, not tuples. The `UNNEST` function converts multisets into one-column tables. This conversion is shown in Figure 16.1.

The `UNNEST` function can also specify a temporary relation name and its attributes, which can be used to refer to the table produced by unnesting. In our example, this relation is `TMPCOURSES(TmpCrsCode)`.

**FIGURE 16.1** Conversion between multisets and one-column tables.



Note that since E ranges over object references, we must access the attributes of the individual objects using the  $\rightarrow$  operator.

SQL:2003 also supports conversion in the opposite direction, from one-column tables to multisets. This is done with the help of the **MULTISET** function. In the following example, we use this function to add a new tuple to the **STUDENT** table. To create such a tuple, we need to construct a multiset of type **COURSETYPE** and assign it to the attribute **Enrolled**. Let us assume that transcripts reside in the following relation:

---

```
CREATE TABLE TRANSCRIPTI (
    Student REF(STUDENTTYPE),
    Course REF(COURSETYPE),
    Semester CHAR(6),
    Grade CHAR(1)
)
```

---

We can now insert a new record into **STUDENT** as follows:

---

```
INSERT INTO STUDENT(Id, Status, Enrolled)
VALUES(123987564, 'G2', MULTISET(SELECT T.Course
                                FROM TRANSCRIPTI T
                                WHERE TRANSCRIPTI->Id=123987564)
)
```

---

The last component of the inserted tuple corresponds to the attribute **Enrolled** and thus must be a multiset of references to objects of type **COURSETYPE**. The nested query returns a one-column table that includes the desired references; the **MULTISET** function converts this column into a multiset.

SQL:2003 introduces a number of additional functions and predicates to facilitate working with multisets:

- **SET(*multiset*)**—function. Eliminates duplicates from *multiset*
- **CARDINALITY(*multiset*)**—function. Returns the cardinality of *multiset*

- *multiset1 INTERSECT multiset2, multiset1 EXCEPT multiset2, etc.*—functions. Return the results of the corresponding set-theoretic operations on multisets
- *multiset IS [NOT] A SET*—predicate. Tests if *multiset* is (or is not) a set
- *multiset1 [NOT] SUBMULTISET OF multiset2*—predicate. Tests if *multiset1* is (or is not) a submultiset of *multiset2*
- *element [NOT] MEMBER OF multiset*—predicate. Tests if *element* is (or is not) a member of *multiset*

Here is a modification of a previous query that uses some of these functions and predicates:

---

```
SELECT S.Id, C.Name
  FROM STUDENT S, COURSE C
 WHERE C.CrsCode IN
   ( SELECT E -> Code
     FROM UNNEST(S.Enrolled) AS TEMPcourses(Code) E
    WHERE CARDINALITY(SET(S.Enrolled)) < 3
      AND S.Enrolled IS NOT A SET )
```

---

This query will return student-course name pairs for only those students who are enrolled in fewer than three distinct courses ( $\text{CARDINALITY}(\text{SET}(S.\text{Enrolled})) < 3$ ) and in at least one course, two or more times ( $S.\text{Enrolled}$  IS NOT A SET). (This situation is possible if a student is enrolled in different sections of the same course, such as Independent Study, which permit this kind of enrollment).

## BIBLIOGRAPHIC NOTES

The emergence of object-oriented databases was preceded by many developments—in particular, the growing popularity of object-oriented languages and the realization of the limitations of the relational data model. The idea of using an existing object-oriented language as a data manipulation language first appeared in [Copeland and Maier 1984]. Nested relations, which represent early attempts to enrich the relational data model, are discussed in [Makinouchi 1977; Arisawa et al. 1983; Roth and Korth 1987; Jaeschke and Schek 1982; Ozsoyoglu and Yuan 1985; Mok et al. 1996].

POSTGRES [Stonebreaker and Kemnitz 1991] was an early proposal for enriching relational databases with abstract data types. Now known as PostgreSQL, this system is a powerful open-source object-relational DBMS that is freely available at [PostgreSQL 2000]. The object database *O<sub>2</sub>*, which strongly influenced the ODMG data model and its query language, is described in [Bancilhon et al. 1990]. The latest version of the ODMG standard can be found in [Cattell and Barry 2000].

The conceptual object data model, presented in Section 16.2, and related issues, are discussed more fully in [Abiteboul et al. 1995]. Logical foundations of object-oriented database query languages have been developed in [Kifer et al. 1995].

The use of path expressions for querying object-like structures first appeared in the GEM system [Zaniolo 1983]. Path expressions were later incorporated in all major proposals for querying objects, including the various object-oriented extensions of SQL, such as XSQL, discussed in [Kifer et al. 1992].

The early databases that supported the object-relational data model were UniSQL, POSTGRES, and *O<sub>2</sub>*. Currently, most major relational database vendors (such as Oracle, Informix, and IBM) provide object-relational extensions to their products. Many ideas underlying the design of these systems found their way into the SQL:1999 and SQL:2003 standards. Further details on the SQL:1999 object-relational extensions can be found in [Gulutzan and Pelzer 1999], while at the time of this writing SQL:2003 is available only through the standards organizations, such as ISO (<http://www.iso.org/>).

Since SQL:1999/2003 object extensions are rather new, there are no products that fully conform to this standard. However, IBM's DB/2 and Oracle 9i come close in terms of syntax and supported features.

This chapter omitted discussion of the database design issues associated with object-oriented databases, which would correspond to the material developed in Chapters 4 and 6 for relational databases. For further discussion see [Biskup et al. 1996a; Biskup et al. 1996b; Gogola et al. 1993; Missaoui et al. 1995]. The approach to object-oriented database design that is growing in popularity is the **Unified Modeling Language (UML)** [Booch et al. 1999; Fowler and Scott 2003]. We described the basics of conceptual database design using UML (and applied it to the relational model) in Chapter 4.

While object-oriented E-R style modeling is currently well developed, the corresponding normalization theory has turned out to be much harder than in the relational case. Beginnings of such a theory can be found in [Weddell 1992; Ito and Weddell 1994; Biskup and Polle 2000a; Biskup and Polle 2000b].

## EXERCISES

- 16.1 Give examples from the Student Registration System where
  - a. It would be convenient to use a set-valued attribute
  - b. It would be convenient to use inheritance
- 16.2 Specify an appropriate set of UDTs for the Student Registration System.
- 16.3 Explain the difference between the object id in an object database and the primary key of a tuple in a relational database.
- 16.4 Explain the different senses in which the objects in an object database can be considered equal.
- 16.5 A relational database might have a table called ACCOUNTS with tuples for each account and might support stored procedures deposit() and withdraw(). An object database might have a class (a UDT) called ACCOUNTS with an object for each account and methods deposit() and withdraw(). Explain the advantages and disadvantages of each approach.

- 16.6 Consider the following type in CODM: [Name: STRING, Members: {PERSON}, Address: [Building: INTEGER, Room: INTEGER]]. Give three examples of subtype for this type. In one example, add more structure to the attribute **Members**; in another, add structure to the attribute **Address**; and in the third, add structure by introducing new attributes.
- 16.7 Give an example of an object that belongs to the domain of the type [Name: STRING, Children: {PERSON}, Cars: {[Make: STRING, Model: STRING, Year: STRING]}]. Consider a supertype [Name: STRING, Cars: {[Make: STRING, Model: STRING]}] of that type. Show how one can obtain an object of the second type from the object of the first type, which you constructed earlier.
- 16.8 Consider the following type, which describes projects: [Name: STRING, Members: {PERSON}, Address: [Building: INTEGER, Room: INTEGER]]. Use SQL:1999/2003 to specify the UDT corresponding to this type.
- 16.9 Use the UDT constructed in Exercise 16.8 to answer the following query: *List the names of all projects that have more than five members.*
- 16.10 Suppose that the ACCOUNTS class in the object database of the previous example has child classes SAVINGSACCOUNTS and CHECKINGACCOUNTS and that CHECKINGACCOUNTS has a child class ECONOMYCHECKINGACCOUNTS. Explain how the semantics of inheritance affects the retrieval of objects in each class. (For example, what classes need to be accessed to retrieve all checking account objects that satisfy a particular predicate?)
- 16.11 Use SQL:2003 (with the MULTISET construct, if necessary) to complete the schema partially defined in Section 16.3. Include the following UDTs: STUDENT, COURSE, PROFESSOR, TEACHING, and TRANSCRIPT. Your solution should follow the object-oriented design methodology. Repeating the statements from Chapters 3 and 4, which use SQL-92, is *not* acceptable.
- 16.12 Use the schema defined for the previous problem to answer the following queries:
- Find all students who have taken more than five classes in the Mathematics Department.
  - Represent grades as a UDT, called GRADETYPE, with a method, **value()**, that returns the grade's numeric value.
  - Write a method that, for each student, computes the average grade. This method requires the **value()** method that you constructed for the previous problem.
- 16.13 Use SQL:2003 and its MULTISET construct to represent a bank database with UDTs for accounts, customers, and transactions.
- 16.14 Use SQL:1999/2003 and the schema constructed for the previous exercise to answer the following queries:
- Find all accounts of customers living at the postal ZIP code 12345.
  - Find the set of all customers satisfying the property that for each the total value of his or her accounts is at least \$1,000,000.
- 16.15 E-R diagrams can be used in designing the class definitions of object databases. Design SQL:1999/2003 definitions for the E-R diagrams in Figure 4.1, Figure 4.6, and Figure 4.36.



# 17

## Introduction to XML and Web Data

The Web opens a new frontier in information technology and presents new challenges to the existing database framework. Unlike traditional databases, data sources on the Web do not typically conform to any well-known structure, such as a relation or object schema. Thus, traditional database storage and manipulation techniques are inadequate to deal with such data sources. This creates a need to extend existing database technologies to support new Web-based applications in electronic information delivery and exchange.

### 17.1 Semistructured Data

At first sight, the information on the Web bears no resemblance to the information stored in traditional databases. However, certain of its characteristics make it possible to apply many of the techniques developed in databases and information retrieval. First note that much of the Web data is presented in a somewhat structured form. For example, Figure 17.1 shows a student list as a tree encoded in Hypertext Markup Language (HTML), in which different data elements are set out using HTML tags.

To the human eye—albeit not quite so to the machine—the information on this HTML page appears to be a completely structured list of students, which, as shown in Figure 17.2, can be represented using the conceptual object data model (CODM) of Section 16.2. The actual object appears at the top of the figure. (We represent this object as an oid-value pair, as in Chapter 16.) The schema corresponding to the student list appears at the bottom of the figure.

How did we get from Figure 17.1, where the structure is implicit and intermixed with the data, to Figure 17.2, where the structure is represented separately from the data? Fortunately, the designer of the Web page was conscious of the need to make the structure easily understandable to a human and so made the data **self-describing** by including the names of the attributes (e.g., Name) along with the values (e.g., John Doe) within the data fields (e.g., Name : John Doe). In contrast, the object contains only the values and the schema contains only the attributes and their types. The label PERSONLIST has been added as the name of the type described by the schema.

Suppose now that the same information is delivered over the Web to a machine (rather than a human) for processing. Unlike the human reader, the machine is less

**FIGURE 17.1** A student list in HTML.

```
<html>
  <head><Title>Student List</Title></head>
  <body>
    <h1>ListName: Students</h1>
    <b>Contents:</b>
    <dl>
      <dt>Name: John Doe
        <dd>Id: 111111111
        <dd>Address:
          <ul>
            <li>Number: 123
            <li>Street: Main St
          </ul>
      <dt>Name: Joe Public
        <dd>Id: 666666666
        <dd>Address:
          <ul>
            <li>Number: 666
            <li>Street: Hollow Rd
          </ul>
    </dl>
  </body>
</html>
```

---

**FIGURE 17.2** Student list in object form.**Object:**

```
(#12345, ["Students",
  { ["John Doe", "111111111", [123,"Main St"] ],
    ["Joe Public", "666666666", [666,"Hollow Rd"] ] }
])
```

**Schema:**

```
PERSONLIST [ ListName: STRING,
  Contents: {
    [ Name: STRING,
      Id: STRING,
      Address:[Number: INTEGER, Street: STRING] ] }
]
```

---

likely to make an intelligent guess about the intended structure of the data received, since it cannot distinguish attributes from values in Figure 17.1. Furthermore, the schema might not even be well defined, as some students on the list might have additional attributes, such as a phone number, or some addresses might have a variable structure (e.g., post office box instead of street address). Therefore, to facilitate machine-to-machine exchange of information, it is advantageous to agree on a format that makes the data self-describing by distinguishing the attribute names from values within the data.

In sum, Web data *created for machine consumption* is likely to have the following characteristics:

- It is *object-like*; that is, it can be represented as a collection of objects of the form described by the conceptual data model introduced in Section 16.2.
- It is *schemaless*; that is, it is not guaranteed to conform to any type structure, unlike the objects discussed in Section 16.2.
- It is *self-describing*.

Data with the above characteristics has been dubbed **semistructured**. The "self-describing" property may be somewhat misleading, since it can imply that the meaning of the data is carried along with the data itself. In reality, semistructured data carries only the names of the attributes and has a lower degree of organization than the data in databases. In particular, since the schema is absent, there is no guarantee that all objects have the same attributes and that the same attribute in different objects has the same meaning.

In view of our observations, Figure 17.2 is not a completely adequate representation of the original data depicted in Figure 17.1 because neither the object notation nor the schema notation of CODM was designed for self-describing data representation. However, an appropriate notation can be developed by combining elements from both the object and the schema notation of CODM. With the new notation, our student list can be represented as schemaless but self-describing as follows:

---

```
(#12345,
  [ListName:"Students",
   Contents:{ [Name:"John Doe",
              Id: "111111111",
              Address:[Number:123, Street:"Main St"] ],
              [Name:"Joe Public",
               Id: "666666666",
               Address:[Number:666, Street:"Hollow Rd"] ] }
  ] )
```

---

17.1

Like the specification in Figure 17.2 (and unlike that in Figure 17.1), this syntax for self-describing objects is precise, machine-understandable, and conforms to the best of database practices. However, this is not the format chosen for data exchange on

the Web. The winner is called the **extensible markup language (XML)**—a standard adopted in 1998 by the World Wide Web Consortium (W3C).

Since its introduction, XML has been steadily gaining momentum and is on the way to becoming the main format for the information intended for both human and machine consumption. Section 17.2 introduces the various components of the language and provides examples of its use.

Although at its core, XML data is schemaless, schema-compliant data is always more useful. In particular, the needs of electronic data exchange require stricter enforcement of the formats for transmission than that provided by semistructured data. To help, XML has *optional* mechanisms for specifying document structure. We discuss two such mechanisms: the **document type definition language (DTD)**, which is part of the XML standard itself, and the **XML Schema**, which is a more recent specification built on top of XML. In the last part of the chapter, we introduce two query languages for XML: a lightweight language called **XPath** and an extension of SQL, called **SQL/XML**, which is designed to provide interoperability between the relational world and the world of XML. The full version of this book [Kifer et al. 2004] includes a study of two other important XML query languages: **XSLT** [XSLT 1999] and **XQuery** [XQuery 2004].

## 17.2 Overview of XML

XML is not a solution to all of the world's problems. It is not a revolutionary or even a new idea. Why, then, is it causing a revolution? In a nutshell, XML is a human- and machine-readable data format that can be easily parsed by an application and thus considerably simplifies data exchange. Formats for data exchange were proposed in the past, but either they were nonopen, proprietary standards or they did not have enough momentum. XML happened to be in the right place at the right time. People saw what the Web and open standards were doing for communication, education, publishing, and commerce, and they recognized the need to simplify data exchange among software agents. It also helped that a trusted standards body, the W3C, was in place and not affiliated with any particular industry group or government. For the first time a simple, open, and widely accepted data standard emerged, and this gave a boost to a wide range of Web applications.

XML is an HTML-like language with an arbitrary number of user-defined tags and no *a priori* tag semantics. To better understand what this means, consider HTML, a document format in which various pieces of text are marked with tags that affect the rendering of that text by a Web browser. An important point is that the number of tags in HTML is *fixed* by the HTML definition and each tag has its own well-defined semantics. The browser displays an HTML document by implementing the semantics of each tag. For instance, any text between the tags `<table>` and `</table>` is supposed to be rendered by the browser as a table, and the tag `<p>` tells the browser to start a new paragraph. In contrast, the repertoire of tags in XML is not set in advance, and the user is free to introduce new tag names. Furthermore, there is no set semantics for any XML tag.

The lack of semantics in XML might seem like a step backward. How does the receiver of an XML document know what to do with the documents it receives? The answer is that each category of applications will supply its own semantic layer on top of XML. Browser rendering is just one type of application. A browser renders an XML document using a **stylesheet**—a transformation that converts the XML document into an HTML document (which the browser already knows how to present). In this way, a stylesheet supplies a “visualization semantics” to XML documents. Most XML documents, however, are not intended for visual display. Instead, they are exchanged by applications and are processed without human intervention (for example, invoices, payments, purchase orders). As with browsers, the application infers the semantics by interpreting XML tags appropriate to the application domain. For example, a retail application might interpret the tag `<price>` to be the price of a product. At this time, whole industries are developing semantic layers for representing information in application domains such as catalogs, commerce, engineering, and other fields. All these efforts have the same common need: the ability to define schema. We discuss the structuring mechanisms available in XML in Sections 17.2.4 and 17.3, although it should be noted that, despite the schema, XML data remains semistructured. Compliance with the schema remains optional, and applications are free to ignore part or all of it.

For concreteness, consider the document in Figure 17.3, which is one possible XML representation of the student list from Figure 17.1. The first line is a mandatory statement that tells the program receiving the document (any such program is called **XML processor**) that it is dealing with XML version 1.0. The rest is structured like an HTML document except for the following important points:

- The document contains a large assortment of tags chosen by the document author. In contrast, the only valid tags in HTML are those sanctioned by the official specification of the language; other tags are ignored by the browser.
- Every opening tag *must* have a matching closing tag, and the tags must be properly nested (i.e., sequences such as `<a><b></a></b>` are not allowed). In contrast, some HTML tags are not required to be closed (e.g., `<p>`), and browsers are forgiving even when closing tags are missing.
- The document has a **root element**—the element that contains all other elements. In Figure 17.3, the root element is `PersonList`.

Any properly nested piece of text of the form `<sometag>...</sometag>` is called an **XML element**, and *sometag* is the **name** of that element. The text between the opening and closing tag is called the **content** of the element. Elements directly nested within other elements are called **children**. For instance, in our example `Name`, `Id`, and `Address` are children of `Person`, which is a child of `Contents`, which is a child of the top-level element, `PersonList`. Conversely, `PersonList` is said to be the **parent** of the elements `Contents` and `Title`, and `Contents` is a parent of `Person`.

XML also defines the **ancestor/descendant** relationships among elements, which are important for querying XML documents and will be revisited in Section 17.4. These relationships have their natural meaning: an ancestor is a parent,

**FIGURE 17.3** XML representation of the student list.

```
<?xml version="1.0" ?>
<PersonList Type="Student" Date="2000-12-12">
    <Title Value="Student List"/>
    <Contents>
        <Person>
            <Name>John Doe</Name>
            <Id>1111111111</Id>
            <Address>
                <Number>123</Number>
                <Street>Main St</Street>
            </Address>
        </Person>
        <Person>
            <Name>Joe Public</Name>
            <Id>6666666666</Id>
            <Address>
                <Number>666</Number>
                <Street>Hollow Rd</Street>
            </Address>
        </Person>
    </Contents>
</PersonList>
```

a grandparent, and so on, and a descendant is a child, a grandchild, and so on. For instance, `PersonList` is an ancestor of `Person` and `Address`, and `Address` is a descendent of `PersonList`.

An opening tag can have **attributes**. In the tag `<PersonList Type="Student">` of Figure 17.3, `Type` is the name of an attribute that belongs to the element `PersonList`, and `Student` is the attribute value. Unlike HTML, all attribute values must be quoted, as shown in the figure, but text strings between tags are not. Also note the element `<Title Value="Student List"/>`, which contains an attribute. This element does not have a closing tag, but instead is enclosed in `<.../>` and is called an **empty element** because it has no content. In XML, this notation is a shorthand for the combination `<Title Value="Student List"> </Title>`.

Apart from elements and attributes, XML allows **processing instructions** and **comments**. A processing instruction is a statement of the form

---

```
<?my-command go bring coffee?>
```

---

and can contain pretty much anything the document author might want to communicate to the XML processor (in the hope that the processor knows what to do with this information). Processing instructions are used fairly rarely.

A comment takes the following form:

---

```
<!-- A comment -->
```

---

It is allowed to occur everywhere except inside the **markups**, that is, between the symbols < and >, which open or close tags. Perhaps surprisingly, a comment is an integral part of the document—the sender is *not* supposed to delete comments prior to transmission, and the receiver is permitted to look inside the comments and use what it finds. Although such treatment of comments goes against prevailing practice in programming and database languages, it is not unheard of in document processing. For instance, JavaScript programs are often placed as comments in HTML documents, and an HTML browser is not supposed to ignore them. Instead, it executes JavaScript programs found inside the comments, unless the JavaScript feature is turned off.

Another feature of XML that is worth a brief mention is the CDATA construct, which serves as a quotation mechanism. Suppose we use XML to write a structured guide to Web publishing. We might want to include the following text:

Web browsers attempt to correct publisher's errors, such as improperly nested tags. For instance, `<b><i>Attention!</b></i>` would be displayed properly by most browsers.

Because of the XML tags included in this text, its inclusion would result in an ill-formed document rejected by every XML-compliant processor. Fortunately, *any* text can be included inside `<![CDATA[...]]>` brackets. For instance, the following is correct XML:

---

```
<![CDATA[<b><i>Attention!</b></i>]]>
```

---

Finally, a document can have an optional **document type definition** (or DTD), which determines document structure. We discuss DTDs in Section 17.2.4.

### 17.2.1 XML Elements and Database Objects

Let us now evaluate the XML document of Figure 17.3 as a format for sending semistructured data over the Web. It is easy to see that the element names effectively serve as attribute names for the object (XML attribute names can serve the same purpose), so this document is essentially yet another, equivalent textual representation for the self-describing object depicted in (17.1).

**Conversion of XML elements into objects.** A moment's reflection should convince us that the nested tag structure of XML is well suited to represent tree-structured self-describing objects. Each element in an XML document can be viewed as an object. The tag names of the children elements then correspond to the object's attributes, and the child elements themselves are the attribute values. For instance,

the first Person element in Figure 17.3 can be partially mapped back to an object as follows (where #6543 is some object Id):

```
{#6543, [Name: "John Doe",
           Id: "111111111",
           Address: <Address>
                     <Number>123</Number>
                     <Street>Main St</Street>
                   </Address>
     ]  
}
```

The conversion process is recursive. Simple elements such as Name and Id are converted immediately by directly extracting their contents. The element Address is left unchanged because it has a complex internal structure, which can be broken further by applying the same conversion procedure recursively. This results in a creation of a new address object:

```
{#098686, [Number: "123",
             Street: "Main St" ]
 }
```

**Differences between XML elements and objects.** Despite the apparent close correspondence between XML elements and structured database objects, there are several fundamental differences. First, XML evolved from and was greatly influenced by SGML [SGML 1986], which is a *document* markup language rather than a *database* language. For instance, XML allows documents of the form

```
<Address>
  Sally lives on
  <Street>Main St</Street>
  house number
  <Number>123</Number>
  in the beautiful Anytown, USA.
</Address>
```

This mixture of text and child elements, allowed in XML, is a hindrance when it comes to automated data processing since the mixture complicates the document.

Second, XML elements are *ordered*, while the attributes of an object in a database are not. Thus, the following two objects are considered the same:

---

```
{#098686, [Number: "123",
    Street: "Main St" ]}
}
{#098686, [Street: "Main St",
    Number: "123" ]
}
```

---

whereas the following two XML documents are different:

---

```
<Address>
    <Number>123</Number>
    <Street>Main St</Street>
</Address>
```

---

```
<Address>
    <Street>Main St</Street>
    <Number>123</Number>
</Address>
```

---

Third, XML has only one primitive type, a string, and very weak facilities for specifying constraints. Fortunately, many of these weaknesses are addressed by the XML Schema specification in Section 17.3.

### 17.2.2 XML Attributes

We saw the use of XML attributes such as Type and Value in Figure 17.3. An element can have any number of user-defined attributes. However, considering the expressive power of XML elements illustrated earlier, we are left to wonder about the role of XML attributes as a tool for data representation. That is, are they useful in data representation, and do they offer anything beyond what elements can offer?

The answer is that XML attributes are sometimes convenient for representing data, but almost everything they can do can also be done with elements. Still, attributes are widely used in XML-based specifications, such as XML Schema, which we introduce in Section 17.3. We also use attributes extensively in the examples to illustrate the various features of XML and because this often leads to more concise representation.

In document processing, attributes are used to annotate pieces of text enclosed between a pair of tags with values that are *not* part of that text but are related to it. In the following dialog,

---

```
<Act Number="5">
    <Scene Number="1" Place="Mantua. A street.">
        :
        <Apothecary Voice="scared">
            Such mortal drugs I have; but Mantua's law
            Is death to any he that utters them.
        </Apothecary>
        <Romeo Voice="persistent">
            Art thou so bare and full of wretchedness,
            And fear'st to die?
```

```
        ...
    </Romeo>
    ...
</Scene>
</Act>
```

we use attributes to annotate the text with meta-information that is not part of the dialog per se but is still relevant. They are convenient to use here because they do not disrupt the dialog flow. In data processing, on the other hand, text flow is a minor concern since computers are unlikely to start appreciating this type of prose in the near future. The concern here is that XML attributes represent yet another, unnecessary dimension in data representation that database programmers have to worry about.

In addition, attribute values can only be strings, which severely limits their usefulness, while XML elements can have children elements, which makes them much more versatile.

Having made these unflattering remarks, we should mention some advantages of attributes. First, the order of attributes in an element does not matter. Thus, the documents

---

```
<thing price="2" color="yellow">foobar</thing>
```

---

and

---

---

```
<thing color="yellow" price="2">foobar</thing>
```

---

are considered the same—much as they are in databases. Second, an attribute can occur at most once (i.e., `<thing price="2" price="2">` is not allowed), while elements with the same tag can be repeated, as in Figure 17.3. This constraint can be handy in the right circumstances. Third, attributes can lead to more succinct representation. For instance, `<thing price="2" color="yellow"/>` is much shorter than `<thing><price>2</price><color>yellow</color></thing>`.

Useful features of an XML attribute are that it can be declared to have a unique value and it can also be used to enforce a limited kind of referential integrity. This cannot be done with elements alone in plain XML. (However, this and much more can be done with the help of XML Schema, discussed in Section 17.3.) After we discuss document type definitions (DTDs) in Section 17.2.4, we will see that an attribute can be declared to be of type ID, IDREF, or IDREFS.

An attribute of type ID must have a unique value throughout the document. This means that if attr1 and attr2 are of type ID, it is illegal for both `<elt1 attr1="abc">` and `<elt2 attr2="abc">` to occur in the same document (regardless of whether elt1 and elt2 are the same tag, or whether attr1 and attr2 are the same attribute). In a sense, ID is a poor cousin of a key in relational databases. An

attribute of type IDREF must refer to a valid Id declared in the same document. That is, its value must occur somewhere in the document as a value of another attribute of type ID. Thus, IDREF is a poor cousin of a *foreign key*.

To illustrate, we consider the report document in Figure 17.4. An attribute of type IDREFS represents a space-separated list of strings, which are references to valid Ids. In our document, we can declare the attribute StudId of the element Student and the attribute CrsCode of the element Course to be of type ID; the attribute CrsCode of the element CrsTaken to be of type IDREF; and the attribute Members of the element ClassRoster of type IDREFS. As a result, any compliant XML processor will verify that no student or course is declared twice and that referential integrity holds—that is, that a course referenced in a CrsTaken element does exist and that all students mentioned in the Members lists are also present in the document.

You might be wondering why we have changed the Ids of students in Figure 17.4 from purely numerical to Ids that start with a letter. The answer is that XML requires that the values of attributes of type ID (and thus of IDREF as well) start with a letter.

We can now define an important correctness requirement. An XML document is **well formed** if the following conditions hold:

- It has a root element.
- Every opening tag is followed by a matching closing tag, and the elements are properly nested inside each other.
- Any attribute can occur at most once in a given opening tag; its value must be provided, as discussed above; and this value must be quoted.

Note that the restrictions on ID, IDREF, and IDREFS are not part of the definition of “well formed” because these attribute types are specified using DTDs, which well-formedness completely ignores.

### 17.2.3 Namespaces

*Namespaces* were not part of the original XML specification and were added as an afterthought. However, they have become central to many important standards built on top of XML, so we consider them to be an integral XML feature for all practical purposes.

The driving force behind the introduction of namespaces was the realization that different communities will be building vocabularies of terms appropriate for the various domains (e.g., education, finance, electronics) and will use them as XML tags. In this situation, naming conflicts between different vocabularies are inevitable, and the integration of information obtained from different sources becomes very hard. For instance, the term *Name* might have different meanings and structure depending on whether we are talking about people or companies, as we see in these two document fragments:

---

```
<Name><First>John</First> <Last>Doe</Last></Name>
<Name>IBM</Name>
```

---

**FIGURE 17.4** A report document with cross-references.

```
<?xml version="1.0" ?>
<Report Date="2000-12-12">
  <Students>
    <Student StudId="s111111111">
      <Name><First>John</First><Last>Doe</Last></Name>
      <Status>U2</Status>
      <CrsTaken CrsCode="CS308" Semester="F1997"/>
      <CrsTaken CrsCode="MAT123" Semester="F1997"/>
    </Student>
    <Student StudId="s666666666">
      <Name><First>Joe</First><Last>Public</Last></Name>
      <Status>U3</Status>
      <CrsTaken CrsCode="CS308" Semester="F1994"/>
      <CrsTaken CrsCode="MAT123" Semester="F1997"/>
    </Student>
    <Student StudId="s987654321">
      <Name><First>Bart</First><Last>Simpson</Last></Name>
      <Status>U4</Status>
      <CrsTaken CrsCode="CS308" Semester="F1994"/>
    </Student>
  </Students>
  <Classes>
    <Class>
      <CrsCode>CS308</CrsCode><Semester>F1994</Semester>
      <ClassRoster Members="s666666666 s987654321"/>
    </Class>
    <Class>
      <CrsCode>CS308</CrsCode><Semester>F1997</Semester>
      <ClassRoster Members="s111111111"/>
    </Class>
    <Class>
      <CrsCode>MAT123</CrsCode><Semester>F1997</Semester>
      <ClassRoster Members="s111111111 s666666666"/>
    </Class>
  </Classes>
  <Courses>
    <Course CrsCode="CS308">
      <CrsName>Software Engineering</CrsName>
    </Course>
    <Course CrsCode="MAT123">
      <CrsName>Algebra</CrsName>
    </Course>
  </Courses>
</Report>
```

So it will become harder for an application to process documents that are built out of the vocabularies that contain conflicting tag names.

To overcome this problem, it has been decided that the name of every XML tag must have two parts: the **namespace** and the **local name**, with the general structure *namespace:local-name*. Local names have the same form as regular XML tags except that they cannot have a : in them. A namespace is represented by a string in the form of a **uniform resource identifier (URI)**, which can be an abstract identifier (a general string of characters serving as a unique identifier) or a **uniform resource locator (URL)** (a Web page address).

The overall idea seems simple enough: different authors use different namespace identifiers for different domains, and thus terminological clashes are avoided. The strategy generally followed since the introduction of namespaces is that authors choose as namespace identifiers the URLs that are under their control. For instance, if Joe Public authors a vocabulary for the school supplies marketed by Acme, Inc., he uses a namespace such as

---

<http://www.acmeinc.com/jp#supplies>

---

and for toys the namespace could be

---

<http://www.acmeinc.com/jp#toys>

---

Note that these URLs need not refer to actual documents.

**Namespace declarations.** The W3C recommendation<sup>1</sup> for incorporating namespaces into XML [Bray 1999] goes beyond a simple two-part naming schema—it also fixes a particular syntax for declaring namespaces, their use, and scoping rules. Here is an example:

---

```
<item xmlns="http://www.acmeinc.com/jp#supplies"
      xmlns:toy="http://www.acmeinc.com/jp#toys">
  <name>backpack</name>
  <feature>
    <toy:item>
      <toy:name>cyberpet</toy:name>
    </toy:item>
  </feature>
</item>
```

---

Namespaces are defined using the attribute `xmlns`, which is a reserved word. In fact, W3C has advised that all names starting with `xml` be considered as reserved

<sup>1</sup> The final documents produced by W3C are inconspicuously called “recommendations,” but in reality they are as good as standards.

for the W3C's use. In our example, we declare two namespaces in the scope of the element `item`. The first one is declared using the syntax `xmlns=` and is called the **default namespace**. Naturally, there can be only one default namespace declaration per opening tag (this follows not only because of the semantics but also because XML does not permit multiple occurrences of the same attribute within the same opening tag). The second namespace is defined with the `xmlns:toy=` declaration. The prefix `toy` serves as a shorthand for the full namespace string `http://www.acmeinc.com/jp#toys`. One can declare several prefixed namespaces as long as the prefixes are distinct.<sup>2</sup>

Tags belonging to the namespace `http://www.acmeinc.com/jp#toys` should be prefixed with `toy:`. In our example, they are the inner tags `toy:item` and `toy:name`. Tags without any prefix (the outer `item`, `name`, and `feature`) are assumed to belong to the default namespace.

Namespace declarations have scope, which can be nested like a program block. To illustrate, we consider the following example:

---

```
<item xmlns="http://www.acmeinc.com/jp#supplies"
      xmlns:toy="http://www.acmeinc.com/jp#toys">
  <name>backpack</name>
  <feature>
    <toy:item>
      <toy:name>cyberpet</toy:name>
    </toy:item>
  </feature>
  <item xmlns="http://www.acmeinc.com/jp#supplies2"
        xmlns:toy="http://www.acmeinc.com/jp#toys2">
    <name>notebook</name>
    <toy:name>sticker</toy:name>
  </item>
</item>
```

---

Here we added one more child element to the outermost `item` element. The child is also called `item`, but it has its own default namespace and a redeclared namespace prefix, `toy`. Thus, the outermost `item` tag belongs to the default namespace

---

<http://www.acmeinc.com/jp#supplies>

---

The inner unprefixed `item` tag and its unprefixed child tag, `name`, are both in the scope of the default namespace

<sup>2</sup> Nevertheless, two tags are assumed to belong to the same namespace, even if they have different prefixes, if and only if their prefixes refer to the same URI ("same" meaning that the URIs are equal as character strings).

---

<http://www.acmeinc.com/jp#supplies2>

---

Similarly, the tags `toy:item` and `toy:name` inside the `feature` element belong to the namespace

---

<http://www.acmeinc.com/jp#toys>

---

The occurrence of `toy:name` at the end of the document belongs to the namespace

---

<http://www.acmeinc.com/jp#toys2>

---

Observe that, just as the innermost declaration of the default namespace overshadows the outermost declaration, the innermost declaration of the prefix `toy` overshadows the outermost declaration for the same prefix. A namespace-aware XML processor is supposed to understand these subtleties and, in particular, to recognize that the two unprefixed occurrences of `item` are *different tags* since they belong to different namespaces. Similarly, the unprefixed occurrences of `name` are different tags, and so are the prefixed versions of `name`. An XML processor that is *unaware* of namespaces will still be able to parse the above document. However, it will think that all unprefixed versions of `item` and `name` are the same and that all occurrences of the prefixed tag `toy:name` are the same. It will just wonder why the name has that weird : inside.

Even though the idea of a namespace seems like motherhood and apple pie—who could possibly be against it—it has been one of the least understood recommendations coming out of W3C [Bourret 2000]. Everyone agrees that tag names should come in two parts, but people have been trying to read between the lines of the recommendation and find things that are not there. One of the most confusing issues is the use of URLs as namespace identifiers. In our everyday experience, a URL points to some Web resource, and if a URL is used for a namespace, one tends to assume that it is a real address that contains some kind of schema describing the corresponding set of names. In reality, the name of a namespace is just a string that happens to be a URL, and it can be a big disappointment when pointing the browser toward such a URL brings up an unattractive error message.

The idea behind namespaces is nothing more than a mechanism for disambiguating tag names. An XML processor that reads a document encoded with namespaces should “know” how to parse it—that is, how to find its schema (represented as a DTD or an XML Schema—the specification languages described later). The information on the schema location can be provided in a special attribute, or it can be part of the convention used in a particular enterprise or community. For example, the toy industry might agree that all toy-related documents should be parsed using the DTD at a particular URL. One convention taking hold right now is that certain vocabularies (such as those used in the XML Schema specification—see Section 17.3) be identified using certain “well-known” namespaces, which prescribe the document schema uniquely.

### 17.2.4 Document Type Definitions

There are fixed rules that an author must follow in order to create an HTML document that can be properly rendered by the browser. For instance, the `table` element cannot occur inside the `form` element. XML, on the other hand, is intended for a variety of application domains (e.g., retail, healthcare, education), and each has its own idea of a properly structured document. Therefore, XML includes a language for specifying the document structure.

A set of rules for structuring an XML document is called a **document type definition (DTD)**. A DTD can be specified as part of the document itself, or the document can give a URL where its DTD can be found. A document that conforms to its DTD is said to be **valid**. The XML specification does not require processors to check each document for conformance to its DTD because some applications might not care if the document is valid. In some cases, the processor does not check validity, instead relying on the guarantee of the sender for this (e.g., in electronic billing, where both sides use software guaranteed to produce valid documents). XML does not even require that the document have a DTD, but it does require that all documents be well formed. (The conditions for well-formedness—proper element nesting and restrictions on the attributes—have been discussed in Section 17.2.2.)

These two notions of correctness can lead to significant simplification and speedup for XML processors. An HTML browser usually tries to correct bugs in the HTML documents and to display as much of a buggy document as possible. In contrast, an XML processor is expected to simply reject documents that are not well formed. A processor that expects valid documents would reject invalid ones (those that do not comply with the DTD) even if they are well formed.

For those who are familiar with formal languages, a DTD is a *grammar* that specifies a valid XML document, based on the tags used in the document and their attributes. For instance, the following DTD is consistent with the document in Figure 17.3 on page 572:

---

```
<!DOCTYPE PersonList [
    <!ELEMENT PersonList (Title,Contents)>
    <!ELEMENT Title EMPTY>
    <!ELEMENT Contents (Person*)>
    <!ELEMENT Person (Name,Id,Address)>
    <!ELEMENT Name (#PCDATA)>
    <!ELEMENT Id (#PCDATA)>
    <!ELEMENT Address (Number,Street)>
    <!ELEMENT Number (#PCDATA)>
    <!ELEMENT Street (#PCDATA)>
    <!ATTLIST PersonList Type CDATA #IMPLIED
                           Date CDATA #IMPLIED>
    <!ATTLIST Title Value CDATA #REQUIRED>
]>
```

---

This example illustrates the most common DTD components: a **name** (`PersonList` in the example) and a set of **ELEMENT** and **ATTLIST** statements. The name of a DTD must coincide with the tag name of the root element of the document that conforms to that DTD. One **ELEMENT** statement exists for each allowed tag, including the root tag. Furthermore, for each tag that can have attributes, the **ATTLIST** statement specifies the allowed attributes and their type.

In our example, the first **ELEMENT** statement says that the element `PersonList` consists of a `Title` element followed by a `Contents` element. A `Title` element (the second **ELEMENT** statement) does not contain any elements (it is an empty element). The `*` in the definition of the `Contents` element indicates that there are zero or more elements of type `Person`. If we use `+` instead of `*`, it would mean that at least one `Person` element must be present. The elements `Name`, `Number`, and `Street` are declared to be of type `#PCDATA`, that is, a character string.<sup>3</sup>

Following the element list, a DTD contains the description of allowed element attributes. In our case, `PersonList` is permitted to have the attributes `Type` and `Date`, while `Title` can only have the attribute `Value`. Other elements are not allowed to have attributes. Moreover, both attributes of `PersonList` are *optional*, as specified by the keyword `#IMPLIED`, while the `Value` attribute of `Title` is mandatory. All three attributes have the type `CData`, which is, again, a character string. (Note that different syntax is used to declare character string types for elements and attributes.)

Observe that our document in Figure 17.3 is valid with respect to the above DTD, but if, for example, we delete some `Address` elements from it, it will become invalid because the DTD says that each person must have an address. On the other hand, if the DTD has

---

```
<!ELEMENT Person (Name,Id,Address?)>
```

---

the address field becomes optional since `?` indicates zero or one occurrences of the `Address` element.

It is also possible to state that the order of elements in a person's description does not matter, using the connective `|`, which represents alternatives:

---

```
<!ELEMENT Person
  ((Name,Id,Address) | (Name,Address,Id) | (Id,Address,Name)
   | (Id,Name,Address) | (Address,Id,Name) | (Address,Name,Id))>
```

---

You can see that it becomes rather awkward, however.

DTDs allow the author to specify several types for an attribute. We have seen `CData`. The other frequently used types are `ID`, `IDREF`, and `IDREFS`, mentioned on page 576 in connection with the report document in Figure 17.4. We pointed out that a document of this type needs a mechanism for enforcing referential integrity—much as in the database examples of Chapter 3.

<sup>3</sup> `PCDATA` stands for *parsed character data*.

FIGURE 17.5 A DTD for the report document in Figure 17.4.

```

<!DOCTYPE Report [
  <!ELEMENT Report (Students,Classes,Courses)>
  <!ELEMENT Students (Student*)>
  <!ELEMENT Classes (Class*)>
  <!ELEMENT Courses (Course*)>
  <!ELEMENT Student (Name,Status,CrsTaken*)>
  <!ELEMENT Name (First,Last)>
  <!ELEMENT First (#PCDATA)>
  :
  :
  <!ELEMENT CrsTaken EMPTY>
  <!ELEMENT Class (CrsCode,Semester,ClassRoster)>
  <!ELEMENT Course (CrsName)>
  :
  :
  <!ELEMENT ClassRoster EMPTY>
  <!ATTLIST Report Date CDATA #IMPLIED>
  <!ATTLIST Student StudId ID #REQUIRED>
  <!ATTLIST Course CrsCode ID #REQUIRED>
  <!ATTLIST CrsTaken CrsCode IDREF #REQUIRED
    Semester CDATA #REQUIRED>
  <!ATTLIST ClassRoster Members IDREFS #IMPLIED>
]>

```

---

Specifically, we want to make sure that the values of the attributes `StudId` in `Student` and `CrsCode` in `Course` are distinct throughout the document, that the attribute `CrsCode` in `CrsTaken` represents a reference to a course mentioned in this document (that there is a `Course` element with a matching value in its `CrsCode` attribute), and that the members in a list indicated by `Members` in `ClassRoster` refer to student records mentioned in the document (for each such member there is a `Student` element with the matching value of its `StudId` attribute). This can be enforced with the DTD shown in Figure 17.5, in which we omit some easily reconstructible parts.

A compliant XML processor that insists on document validity is obliged by this DTD to make sure that no two `Student` elements have the same value in their `StudId` attribute (similarly for `Course` elements). This is because `StudId` is declared to have the type `ID`. In fact, no pair of attributes of type `ID` (with the same or different names) can have the same value in a valid XML document.

Referential integrity is maintained using the `IDREF` and `IDREFS` declarations. Because the attribute `CrsCode` of the element `CrsTaken` is declared as `IDREF`, referential integrity for course codes is preserved. The attribute `Members` in `ClassRoster`

is declared as IDREFS, which represents *lists* of values of type IDREF. This secures the integrity of references to student IDs.

There are also constraints in the document that beg to be noticed, but they cannot be enforced using DTDs. We discuss these issues in the next section.

### 17.2.5 Inadequacy of DTDs as a Data Definition Language

XML was conceived as a simplified, streamlined version of SGML [SGML 1986], which was standardized years before the work on XML began. SGML was created for specifying documents that can be exchanged and automatically processed by software agents, and this was the original goal of XML as well. DTDs and the rationale behind their use were borrowed from SGML. Their technical underpinnings come from the theory of formal languages, and general-purpose parsers that can validate any document against any DTD are well known. Such validation has important implications for document-processing software. For instance, if an XML processor can expect that the documents it receives have been validated and will conform to the DTD Report shown in Figure 17.5, it does not need to take care of special cases and exceptions, such as the possibility that a student might have taken a nonexistent course or that a street address is missing.

During the development of XML, new ideas started to emerge. In particular, XML introduced the possibility of treating Web documents as data sources that can be queried (similarly to database relations) and that can be related to each other through semantically meaningful links (similar to foreign-key constraints). It was at this point that XML began to outgrow its SGML heritage. One of the first enhancements, which came too late to be included in XML 1.0, was namespaces, discussed earlier. A much more significant enhancement is the development of the XML Schema specification (Section 17.3), which is designed to rectify many of the limitations of DTD as a data definition language. These limitations include the following:

- DTDs are not designed with namespaces in mind. A DTD views `xmllns` as just another attribute with no special meaning. It is not hard to extend them to include namespaces, but there is a problem of backward compatibility and, in view of other limitations of DTDs, such enhancement is probably a futile exercise.
- DTDs use syntax that is quite different from that of XML documents. While this is not a fatal drawback, it is not the most elegant feature of XML 1.0 either.
- DTDs have a very limited repertoire of basic types (essentially just glorified strings).
- DTDs provide only limited means for expressing data-consistency constraints. They do not have keys (except for the very limited ID type), and the mechanism for specifying referential integrity is very weak. The only way to reference something is through the IDREF and IDREFS attributes, and even these are based on only one primitive type, a string. In particular, it is not possible to type the references. One cannot require that the attribute `CrsCode` of the element

**CrsTaken** in the report document of Figure 17.4 reference only **Course** elements. Thus, it is possible for John Doe to have a child element

---

```
<CrsTaken CrsCode="s666666666" Semester="F1999"/>
```

---

which refers to the student Id of Joe Public instead of to a course, and no XML 1.0 compliant processor can detect this problem. DTDs have ways of enforcing referential integrity for attributes but no corresponding feature for elements. For example, the contents of the element **Class** include the elements **CrsCode** and **Semester** (not to be confused with similarly named attributes of the tag **CrsTaken**). Clearly, we want the content of the element **CrsCode** to refer to a valid course and match a value of the attribute **CrsCode** in some **Course** element. Furthermore, for each pair of values of the attributes in the element **CrsTaken**, there must be a corresponding pair of values of **CrsCode/Semester** tags in some **Class** element. These constraints cannot be enforced using DTDs.

- XML data is ordered; database data is unordered (e.g., the order of tuples does not matter). Also, the order of the attributes in a database relation or an object does not matter; the order of elements in XML matters. We already saw that DTDs allow us to specify alternatives, and through them we can state that the order of elements is immaterial (as in the earlier example of the **Name**, **Address**, and **Id** children of the element **Person**). However, this becomes extremely awkward as the number of attributes grows. For instance, to state that the order among  $N$  children elements is immaterial, a DTD must specify  $N!$  alternatives.
- Element definitions are global to the entire document. If a DTD specifies that, for example, **Name** consists of children elements **First** and **Last**, then it is not possible to have a *differently structured* **Name** element anywhere else in the document. This happens because a DTD can have only one **ELEMENT** clause per element name. There is no way to localize it with respect to a parent element so that different definitions would apply to different occurrences of **Name**, depending on where it is nested.

## 17.3 XML Schema

XML Schema, a data definition language for XML documents, became a recommendation of W3C in 2001. It was developed in response to the aforesaid limitations of the DTD mechanism and has the following main features:

- It uses the same syntax as that used for ordinary XML documents.
- It is integrated with the namespace mechanism. In particular, different schemas can be imported from different namespaces and integrated into one schema.
- It provides a number of built-in types, such as string, integer, and time—similar to SQL.
- It provides the means to define complex types based on simpler ones.

- It allows the same element name to be defined as having different types depending on where the element is nested.
- It supports key and referential integrity constraints.
- It provides a better mechanism for specifying documents where the order of element types does not matter.

The downside is that XML Schema is an order of magnitude more complex than the DTDs, and the DTDs are still widely used for simpler kinds of XML processing, those where the advanced features just described are not required.

An XML document that conforms to a given schema is said to be **schema valid** and is called an **instance** document of the schema. As with DTDs, the XML Schema specification does not require an XML processor to actually use the document schema. It is free to ignore the schema or to use a different one. For instance, the XML processor might want to consider only the documents that satisfy stricter integrity constraints than those given in the schema, or it might decide to enforce only part of the schema. This liberal attitude should be contrasted with databases, where *all* data must comply with the schema. In this sense, XML data as a whole should be considered semistructured (Section 17.1) despite the fact that a schema might partially describe it.

### 17.3.1 XML Schema and Namespaces

An XML Schema document (like a DTD) describes the structure of other (instance) XML documents. It begins with a declaration of the namespaces to be used in the schema, three of which are particularly important:

- <http://www.w3.org/2001/XMLSchema>—The namespace that identifies names of tags and attributes used *in the schema*. These names are not related to, nor do they appear in, any particular instance document. Instead, they are used in schema documents to describe structural properties of instance documents. Hence, this namespace is part of schema documents but is not used in instance documents. Among the names associated with this namespace are **schema**, **attribute**, and **element**.
- <http://www.w3.org/2001/XMLSchema-instance>—Another namespace used in conjunction with <http://www.w3.org/2001/XMLSchema>. It identifies a small number of special names, which are also defined in the XML Schema specification but are used in the instance documents rather than in their schema (whence the name **XMLSchema-instance**). One such name, **schemaLocation**, specifies the location of the schema for the document. Another defines the special **null** value when it appears in a document. We will discuss these features in due time. This namespace is part of the specification of instance documents since it defines tags used in those documents.
- The **target namespace**—Identifies the set of names *defined* by a particular schema document, in other words, the user-defined names that are to be used in the instance documents of that particular schema. For instance, in the schema

document for Figure 17.4, the names `CrsTaken`, `Student`, `Status`, and so forth, would be associated with the target namespace. (We will soon start developing the various parts of that schema.) The target namespace is declared using the attribute `targetNameSpace` of the opening tag of the `schema` element—the root tag of every schema document.

The integration with namespaces is one of the important items missing in DTDs: a DTD can define any number of tags, but there is no way to associate those tags with a namespace.

We now begin to develop a schema for the report document of Figure 17.4. Our first example simply declares the namespaces to be used in the schema we are creating.

---

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://xyz.edu/Admin">

    <!-- Nothing here yet -->
</schema>
```

---

The first namespace declared in this example makes the standard `XMLSchema` namespace the default. This is handy because in creating the schema, we are likely to use many special tags defined by the XML Schema specification, and making `XMLSchema` the default namespace will obviate the need for namespace prefixes for them. If, however, we want a different namespace to be the default, we can use

---

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

---

By convention, `xsd` is the prefix for names in the standard `XMLSchema` namespace. In this case, we have to use `xsd` whenever a name associated with the XML Schema's namespace is used:

---

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              xsd:targetNamespace="http://xyz.edu/Admin">

    <!-- Nothing here yet -->
</xsd:schema>
```

---

The first attribute here says that `xsd` is the prefix for names associated with the `XMLSchema` namespace. The second attribute says that the new tags and attributes defined by the above schema document are considered to be part of the `http://xyz.edu/Admin` namespace. Note that, since `targetNamespace` is a name defined by the XML Schema specification, its use is prefixed with `xsd`.

Suppose now that we have filled in all the blanks in the above schema. How does the fact that we now have a schema for the instance document in Figure 17.4 change this document? We need to add three things to the instance: the declaration of the

**FIGURE 17.6** Schema and an instance document.

```
<!-- An XML schema document; located at http://xyz.edu/Admin.xsd -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
         targetNameSpace="http://xyz.edu/Admin">

    <!-- Nothing here yet -->
</schema>

<!-- An instance document conforming to the above schema;
     it uses the target namespace defined in that schema -->
<?xml version="1.0" ?>
<Report xmlns="http://xyz.edu/Admin"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xyz.edu/Admin
                             http://xyz.edu/Admin.xsd">

    <!-- Same contents as in the report document of Figure 17.4 -->
</Report>
```

---

default namespace it uses (in our case, `http://xyz.edu/Admin`), the location of its schema, and the `XMLSchema-instance` namespace. The latter is needed because the attribute `schemaLocation`, which specifies the schema location, occurs in instance documents and is part of the `XMLSchema-instance` namespace. To better understand the relationship among the schema, the actual instance document, and the various namespaces, we show the report document and its schema together in Figure 17.6.

Note in the figure that the default namespace in the instance document is `http://xyz.edu/Admin`—the namespace defined in the `targetNameSpace` attribute of the schema document.<sup>4</sup> There need not be anything at this URL because a namespace is just an identifier that is used to disambiguate the names of document tags and attributes. This namespace is chosen as a default in order to minimize the number of namespace prefixes that need to be used in the document. Because the document in Figure 17.6 is supposed to have the same contents as in the report in Figure 17.4, most of the tag and attribute names belong to this default namespace.

The attribute `xsi:schemaLocation` is part of the XML Schema specification and belongs to the namespace

---

`http://www.w3.org/2001/XMLSchema-instance`

---

The value of the attribute is a namespace-URL pair, and it says that the schema for the namespace `http://xyz.edu/Admin` can be found in an XML Schema document

<sup>4</sup> Most namespaces and document locations used in the examples have been changed to protect the innocent. However, the `XMLSchema` and `XMLSchema-instance` namespaces are real.

at the URL `http://xyz.edu/Admin.xsd`. However, as mentioned earlier, XML processors are not bound by these hints. They can choose to ignore the schema or to use a different one.

Before plunging into the specifics of defining the actual schema, we mention one other important detail, the `include` statement. It is easy to see from Figure 17.4 that our report has three distinct components: a student list, a class list, and a course list. Since these components have very different structures, it is reasonable to assume that they might well occur separately in other contexts and that they might have their own schemas. Given this, it is unreasonable for us to copy those schemas over in order to create the schema for the report document. Instead, we can use the `include` statement in the schema document as follows:

---

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://xyz.edu/Admin">

    <include schemaLocation="http://xyz.edu/StudentTypes.xsd"/>
    <include schemaLocation="http://xyz.edu/ClassTypes.xsd"/>
    <include schemaLocation="http://xyz.edu/CourseTypes.xsd"/>

    <!-- Nothing here yet -->
</schema>
```

---

The effect of the `include` statement is to include the schemas at the specified address in the given document. This technique allows for greater flexibility and modularity of XML schemas. Included schemas must have the same target namespace as the including schema since the `include` statement effectively integrates them into the including schema document. Observe one possibly confusing detail in the above example. We have used the attribute `schemaLocation` without prefixing it with `xsi`, and, unlike the previous example, we did not include the `XMLSchemas-instance` namespace. This discrepancy has a rational explanation. The `schemaLocation` attribute of the tag `include` belongs to the standard `XMLSchemas` namespace (like the `include` tag itself); that is, this attribute is different from the similarly named attribute in the report document above. Since, unlike the report document, our schema does not use any names from the `XMLSchemas-instance` namespace, this namespace was not declared.

### 17.3.2 Simple Types

**Primitive types.** The dearth of primitive types was one of the criticisms leveled against DTDs. The XML Schema specification rectifies the problem by adding many useful primitive types, such as `decimal`, `integer`, `float`, `boolean`, and `date`, in addition to `string`, `ID`, `IDREF`, and `IDREFS`. More important, it provides type constructors, such as `list` and `union`, and a mechanism to derive new primitive types

from the basic ones. This mechanism is similar to the CREATE DOMAIN statement of SQL (see Section 3.3.6).

**Deriving simple types using the list and union constructors.** As in DTDs, IDREFS is one of the primitive types in the XML Schema specification. However, it can also be derived using the list constructor.<sup>5</sup>

---

```
<simpleType name="myIdrefs">
    <list itemType="IDREF"/>
</simpleType>
```

---

Here the `name` attribute is used to give a name, `myIdrefs`, to the newly defined type. This and any other name introduced by the `name` attribute in a schema document belongs to the target namespace of that document.

The `union` type can be useful when there is a need for two or more ways to enter data. For instance, in the United States a telephone number can be 7 or 10 digits long, which can be expressed as follows:

---

```
<simpleType name="phoneNumber">
    <union memberTypes="phone7digits phone10digits"/>
</simpleType>
```

---

We will see the definitions of the types `phone7digits` and `phone10digits` shortly.

**Deriving simple types by restriction.** A more interesting way of deriving new types is via the `restriction` mechanism, which allows us to constrain a basic type using one or more constraints from a fixed repertoire defined by the XML Schema specification. This is how we are going to define the type `phone7digits`:

---

```
<simpleType name="phone7digits">
    <restriction base="integer">
        <minInclusive value="1000000"/>
        <maxInclusive value="9999999"/>
    </restriction>
</simpleType>
```

---

The 10-digit number type is defined similarly. In the definition of `phone7digits`, we used the tags `maxInclusive` and `minInclusive` to define the range of acceptable numbers. XML Schema provides a large number of built-in constraints to play with, such as `minInclusive/maxInclusive` [XMLSchema 2000a, XMLSchema 2000b].

<sup>5</sup> Unless stated otherwise, all examples of XML schemas assume the standard <http://www.w3.org/2000/10/XMLSchema> namespace as a default.

Here we mention just a few of the more interesting ones. Suppose that, in addition, we let the user specify phone numbers in the XXX-YYYY format. This can be done in several ways, one being

---

```
<simpleType name="phone7digitsAndDash">
    <restriction base="string">
        <pattern value="[0-9]{3}-[0-9]{4}" />
    </restriction>
</simpleType>
```

---

Here we use the pattern tag to restrict the set of all strings to those that match the given pattern. The language for constructing patterns is similar to that used in the Perl programming language, but the basics should be familiar to anyone with a working knowledge of text editors such as Vi or Emacs. In the above example, [0-9] means “any digit between 0 and 9” and {3} is a pattern modifier that says that only a sequence of exactly three digits is allowed.

Other ways to derive simple types from the basic string type include the following:

- <length value="7"/>—Restricts the domain to strings of length 7.
- <minLength value="7"/>—Restricts the domain to strings of length *at least* 7.
- <maxLength value="14"/>—Restricts the domain to strings of length *at most* 14.
- <enumeration value="ABC"/>—Specifies one value in an enumerated set.

The above constraints are not limited to strings, and enumeration is applicable to virtually any base type. Here is an example:

---

```
<simpleType name="emergencyNumbers">
    <restriction base="integer">
        <enumeration value="911"/>
        <enumeration value="333"/>
        <enumeration value="5431234"/>
    </restriction>
</simpleType>
```

---

**Simple types for the report document.** We now define some simple types for our report document of Figure 17.4 on page 578. We will later attach these types to the appropriate attributes in the document schema. For easy reference, we summarize all student-related types in Figure 17.9 on page 605 and all course-related types in Figure 17.10 on page 606.

---

```
<simpleType name="studentId">
    <restriction base="ID">
        <pattern value="s[0-9]{9}" />
```

```
</restriction>
</simpleType>
<simpleType name="studentRef">
    <restriction base="IDREF"
        <pattern value="s[0-9]{9}" />
    </restriction>
</simpleType>
<simpleType name="studentIds">
    <list itemType="adm:studentRef" />
</simpleType>
<simpleType name="courseCode">
    <restriction base="ID">
        <pattern value="[A-Z]{3}[0-9]{3}" />
    </restriction>
</simpleType>
<simpleType name="courseRef">
    <restriction base="IDREF">
        <pattern value="[A-Z]{3}[0-9]{3}" />
    </restriction>
</simpleType>
```

The first type, `studentId`, defines student IDs as digit strings of length 9; it will be used to specify the domain of values for `studId` in the report. The second defines the type of *references* to student IDs, the third defines *lists* of references to student IDs, the fourth defines course codes as strings of three uppercase letters followed by three digits, and the fifth is the type for course references. Note also that we have used ID and IDREF as base types. They have the same semantics as they do in DTDs, so uniqueness and referential integrity are guaranteed. The definition of the type `studentIds` uses a previously defined type `studentRef`. Note that the reference to that type is tagged with a namespace prefix `adm`, which is here assumed to be associated with the target namespace of the schema document. The need for this prefix will be explained shortly.

Observe that we are already doing better than in the case of the Report DTD shown in Figure 17.5 on page 584. It is impossible for a DTD to say that the attribute `Members` returns a list of references to students rather than to courses or to impose a similar restriction on the attribute `CrsCode` of the tag `CrsTaken`. In contrast, the above simple types prevent such meaningless references because the type `courseRef` is disjoint from the domain of `studentId` and the domain of `studentRef` is disjoint from that of `courseCode`.

**Type declarations for simple elements and attributes.** So far, we have been talking about types without attaching them to elements and attributes. Here are some simple cases of type declaration for tags in our report document, which will later become part of the schema document for this report.

---

```
<element name="CrsName" type="string"/>
<element name="Status" type="adm:studentStatus"/>
```

---

**17.2**

The first declaration states that the element `CrsName` has a simple content of type `string`. The last declaration is fancier: it associates the `Status` tag with a derived type, `studentStatus`, defined as an enumeration of strings `U1`, `U2`, `U3`, `U4`, `G1`, `G2`, `G3`, `G4`, and `G5`, which represent the various status codes for undergraduate and graduate students.

---

```
<simpleType name="studentStatus">
    <restriction base="string">
        <enumeration value="U1"/>
        <enumeration value="U2"/>
        :
        <enumeration value="G5"/>
    </restriction>
</simpleType>
```

---

A subtle but very important point in this example is the prefix `adm` attached to `studentStatus` in (17.2)—a consequence of the namespace consideration. To understand this better, let us consider the context in which the above statements appear:

---

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:adm="http://xyz.edu/Admin"
    targetNamespace="http://xyz.edu/Admin">
    :
    <element name="CrsName" type="string"/>
    <!-- reference to StudentStatus -->
    <element name="Status" type="adm:studentStatus"/>
    :
    <!-- definition of StudentStatus -->
    <simpleType name="studentStatus">
        :
    </simpleType>
    :
</schema>
```

---

In a schema document the default is typically the standard XMLSchema namespace. This enables us to use frequently occurring symbols, such as `element`, `simpleType`, `name`, and `type`, without a prefix. In addition, a schema document defines a number of types (e.g., `studentStatus`), elements (e.g., `Status`), and attributes (see Section 17.3.3) that belong to a target namespace (`http://xyz.edu/Admin` in our case). When we define a new element or type, we use it without a prefix (for example, `name="Status"` and `name="studentStatus"`) because these names are newly defined and hence cannot be part of the default namespace. They are automatically placed in the target namespace. However, how do we refer to the names defined within the same schema (for example, our reference to `studentStatus` in the `type` attribute)? If we do not use any prefix, the XML processor is supposed to assume that the name belongs to the default namespace. This is precisely what happens with the `string` type of the element `CrsName`. Since `string` is not prefixed, it is assumed to be taken out of the standard XMLSchema namespace, which is correct. In contrast, using `studentStatus` without a prefix causes the XML processor to assume that this symbol also comes from the default namespace, which is an error since XML Schema does not define `studentStatus`. Therefore, we need to define a namespace prefix for the target namespace and use it with every reference to a component of the target schema. The purpose of the second occurrence of the `xmlns` attribute of the `schema` element in the above example is thus to associate the prefix `adm` to the target namespace. From now on, we assume that the target namespace has the prefix `adm`, and we will use it with defined types without mention.

Next, consider how one specifies the types of some attributes in our document:

---

```
<attribute name="Date" type="date"/>
<attribute name="StudId" type="adm:studentId"/>
<attribute name="Members" type="adm:studentIds"/>
<attribute name="CrsCode" type="adm:courseCode"/>
```

---

Notice that these declarations do not associate attributes with elements, so they are not very meaningful at this point. We cannot make the association here because elements that have attributes are considered to have *complex types* (even if they have empty content, such as `CrsTaken`), so we need to familiarize ourselves with such types first.

### 17.3.3 Complex Types

**Basic example.** So far we have seen how to define *simple types*—the only types allowed in attributes and the types of elements that do not have attributes or children. The fragment of a schema in Figure 17.7 defines a complex type suitable for the `Student` element in the report document.

This example contains two type declarations and many new features. First, the tag `complexType` is used instead of `simpleType` to warn the XML processor of things to come. Second, the `sequence` tag is used to specify that the elements `Name`, `Status`, and `CrsTaken` must occur in the given order. Third, the `CrsTaken` element (whose

**FIGURE 17.7** Definition of the complex type `studentType`.

```

<complexType name="studentType">
  <sequence>
    <element name="Name" type="adm:personNameType"/>
    <element name="Status" type="adm:studentStatus"/>
    <element name="CrsTaken" type="adm:courseTakenType"
      minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="StudId" type="adm:studentId"/>
</complexType>
<complexType name="personNameType">
  <sequence>
    <element name="First" type="string"/>
    <element name="Last" type="string"/>
  </sequence>
</complexType>

```

---

type will be defined shortly) is said to occur zero, one, or more times. In general, we can specify any number as a value of `minOccurs` and `maxOccurs`. Doing the same with DTDs is possible but extremely awkward since one must use alternatives (specified using `|`), which leads to unwieldy schemas. For other elements, we did not specify `minOccurs` and `maxOccurs` because they both default to 1 (which we want anyway). Finally, the attribute declaration at the end of the complex type definition associates `StudId` with the type `studentId` (Figure 17.9 on page 605), and, because it occurs in the scope of the definition of `studentType`, it means that every element of type `studentType` must have this attribute (and no other).

The second type declaration in Figure 17.7 supplies the type for the `Name` element used in the definition of `studentType`. This declaration does not introduce new features.

Associating a complex type with an element is no different from associating a simple type with an element. The following statement associates the `Student` element with the complex type `studentType`:

---

```
<element name="Student" type="adm:studentType"/>
```

---

**Special cases.** The simple picture just described is complicated by two special cases: how do we define the type of an element that has both a simple content (just text with no children elements) *and* attributes, and how can we define the type of an element that has attributes but *no* content (defined as `EMPTY` in the DTD)? We have seen the first kind of element in the dialog between Romeo and Apothecary on page 575; the second kind is represented by the element `CrsTaken` of Figure 17.4, page 578.

Defining the first type of element is a little awkward, and we skip this topic since it rarely occurs in data representation using XML.<sup>6</sup> On the other hand, defining the type for elements such as CrsTaken is easy:

---

```
<complexType name="courseTakenType">
    <attribute name="CrsCode" type="adm:courseRef"/>
    <attribute name="Semester" type="string"/>
</complexType>
```

---

**Combining elements into groups.** The example of studentType in Figure 17.7 shows how to combine elements into an ordered group using sequence. Tags such as sequence, which describe how elements can be combined into groups, are called **compositors**; they are required when an element has **complex content**, that is, when the element has at least one child element. XML Schema defines several compositors; one provides a way to combine elements into *unordered* sets. Note that the lack of a practical way to specify unordered collections of elements was one of the criticisms of DTDs in Section 17.2.5.

Suppose that we want to allow the street name, number, and the city name to appear in any order in an address. We can specify this using the compositor all:

---

```
<complexType name="addressType">
    <all>
        <element name="StreetName" type="string"/>
        <element name="StreetNumber" type="string"/>
        <element name="City" type="string"/>
    </all>
</complexType>
```

---

Unfortunately, there are a number of restrictions on all that make it hard to use in many cases. First, all must appear directly below complexType, so the following is illegal:

---

```
<complexType name="studentType2">
    <sequence>
        <all>
            <element name="Name" type="adm:personNameType"/>
            <element name="Status" type="adm:studentStatus"/>
        </all>
        <element name="CrsTaken" type="adm:courseTakenType"
            minOccurs="0" maxOccurs="unbounded"/>
```

---

<sup>6</sup> Defining an element whose content is just text (no children elements) and which has attributes is done with the help of the simpleContent tag defined by XML Schema; see, for example, the XML Schema Primer [XMLSchema 2000a].

---

```

</sequence>
<attribute name="StudId" type="adm:studentId"/>
</complexType>

```

---

Second, no element within it can be repeated. In other words, `maxOccurs` must be 1 for every child of `all`, so the following is also not allowed:

---

```

<complexType name="studentType3">
    <all>
        <element name="Name" type="adm:personNameType"/>
        <element name="Status" type="adm:studentStatus"/>
        <element name="CrsTaken" type="adm:courseTakenType"
            minOccurs="0" maxOccurs="unbounded"/>
    </all>
    <attribute name="StudId" type="adm:studentId"/>
</complexType>

```

---

The third grouping construct of XML Schema is the `choice` compositor, which plays the same role for complex types as `union` does for simple types. For instance, in the following example,

---

```

<complexType name="addressType">
    <sequence>
        <choice>
            <element name="POBox" type="string"/>
            <sequence>
                <element name="Name" type="string"/>
                <element name="Number" type="string"/>
            </sequence>
        </choice>
        <element name="City" type="string"/>
    </sequence>
</complexType>

```

---

`choice` lets us substitute the post office box for the street address. That is, a valid address must have precisely one of the two possibilities: a post office box or a street address.

Note that a content descriptor, such as a compositor, is required even if the type contains only one child element. For instance,

---

```

<complexType name="foo">
    <element name="bar" type="integer"/>
</complexType>

```

---

is illegal, but

---

```
<complexType name="foo">
    <sequence>
        <element name="bar" type="integer"/>
    </sequence>
</complexType>
```

---

is correct.

**Local element names.** In DTDs, all element declarations are global because only one ELEMENT statement per element name is allowed. Thus, it is not possible to define a valid report document (with respect to *any* DTD) where both Student and Course have a Name child element with different types. Indeed, this is the case in Figure 17.4 on page 578, where a course name is a string while a student name has complex type personNameType. Since the types are different, we could not use Name as the element name for both. Instead, we had to use CrsName to identify course names. For the same reason, DTDs do not let us use the element name Course instead of the name CrsCode for the child element of Class: the Course child inside the element Courses has a different structure than the CrsCode element inside Class. Thus, if we replace the tag name CrsCode with Course, the DTD must have two different ELEMENT statements for Course, which is impossible.

The XML Schema specification corrects this problem by providing local scope to element declarations. This is done as in any programming language. A declaration of an element type is considered local to the nearest containing <complexType ... > ... </complexType> block. In the report document, this local scoping allows us to rename the CrsName tag to Name and define the following schema:

---

```
<complexType name="studentType">
    <sequence>
        <element name="Name" type="adm:personNameType"/>
        <element name="Status" type="adm:studentStatus"/>
        <element name="CrsTaken" type="adm:courseTakenType"
            minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="StudId" type="adm:studentId"/>
</complexType>
<complexType name="courseType">
    <sequence>
        <element name="Name" type="string"/>
    </sequence>
    <attribute name="CrsCode" type="adm:courseCode"/>
</complexType>
```

---

Here both studentType and courseType include a child element, Name. In the first case, this element has a complex type personNameType, which includes two ele-

ments: *First* and *Last*. In the second case, it has a simple type, `string`. However, unlike in a DTD, the two declarations have a different scope and thus their definitions do not clash.

**Importing schemas.** In Section 17.3.1, we illustrated the use of the `include` instruction for constructing a schema out of separate components that reside in different files. This facility supports modular construction of complex XML Schemas by small teams of collaborating programmers. Therefore, it requires that the target namespace of an included schema be the same as the target namespace of the containing schema.

At the same time, the designers of the XML Schema specification understood that the true potential of the Web can be realized only if people can pull together schemas constructed by different groups or organizations. This is the goal of the `import` statement. As with the `include` statement, the `schemaLocation` attribute is provided, but it is optional for `import`. The only required attribute is `namespace` because it is possible to import a schema whose target namespace is different from the target namespace of the importing schema. In the absence of `schemaLocation`, the XML processor is supposed to find the schema on its own, possibly deriving it from the namespace using some convention. Even when `schemaLocation` is provided, the processor is allowed to ignore it or to use a different schema. The only thing that the processor must not ignore is the `namespace`. Thus, the `namespace` attribute of the `import` statement determines the target namespace of the imported schema.

In the following example, we use `import` instead of `include`:

---

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNameSpace="http://xyz.edu/Admin"
        xmlns:reg = "http://xyz.edu/Registrar"
        xmlns:crs = "http://xyz.edu/Courses">
    <import namespace="http://xyz.edu/Registrar"
            schemaLocation="http://xyz.edu/Registrar/StudentTypes.xsd"/>
    <import namespace="http://xyz.edu/Courses"/>
    :
</schema>
```

---

Here we assume that the schemas describing students and courses use different target namespaces and that the report processing software knows where to find the schema that describes courses. Therefore, the `schemaLocation` attribute is not provided in the second `import` statement (but it is in the first). The first `import` statement imports a schema with target namespace `http://xyz.edu/Registrar`. This namespace is assigned the prefix `reg` so that any part of the imported schema, for example, `x`, could be referred to as `reg:x`.

**Deriving new complex types by extension and restriction.** In some cases, the user might need to modify parts of the included or imported schema. This is easy with inclusion because all documents are assumed to be under the author's control. With importing, however, the control is usually with an external entity and the importer might not be allowed to copy the schema, or this might not be desirable. For example, in many cases, the importer just wants to have a "view" of the original schema so that the importer's schema would change along with that original.

XML Schema provides two mechanisms for modifying imported schema: **extension** and **restriction**. Both are special cases of the notion of *subtype* defined in Section 16.2.3. Extending a schema means adding new elements or attributes to it. Restricting a schema means tightening its definition in order to exclude some instance documents.

Suppose that `foo.edu` decides to follow the example of `xyz.edu` and "XMLize" their registration system. Overall they like the schema of `xyz.edu` but want to add a short course syllabus to every course record. Because `xyz.edu` is constantly improving its XML student registration tools, `foo.edu` decides that it can take advantage of these improvements by importing and *extending* the schema rather than copying it over. Specifically, `foo.edu` wants to extend the type `courseType` (Figure 17.10, page 606) with an additional element, `syllabus`. To this end, they create the following schema document:

---

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xyzCrs="http://xyz.edu/Admin"
        xmlns:fooAdm="http://foo.edu/Admin"
        targetNamespace="http://foo.edu/Admin">
    <!-- fooAdm is the prefix to be used for references
        to the target namespace within this schema -->
    <import namespace="http://xyz.edu/Admin"/>

    <complexType name="courseType">
        <complexContent>
            <extension base="xyzCrs:courseType">
                <element name="syllabus" type="string"/>
            </extension>
        </complexContent>
    </complexType>
    <!-- Now define a Course element for the target namespace
        and associate it with the derived type -->
    <element name="Course" type="fooAdm:courseType"/>
    :
</schema>
```

---

Notice that the target namespace is now `http://foo.edu/Admin`—that of the client university—and we associate the prefix `fooAdm` with it. The document uses the import statement to obtain the schema of `xyz.edu` to use as a basis for constructing a new schema. We assume that the important schema has the namespace `http://xyz.edu/Admin`. Since the new schema refers to the names defined in `xyz.edu`'s namespace (such as `courseType`), we need to associate a prefix with the imported namespace of `xyz.edu`. We choose `xyzCrs`.

After defining the namespaces, we define a new type, `courseType`, using a similar type in the imported schema. The newly defined type is not prefixed because we want it to belong to the target namespace. However, the base type imported from `xyz.edu` is prefixed and is referred to as `xyzCrs:courseType`. To signal the XML processor that a complex type is to be defined by modifying another type, the XML Schema specification requires the `<complexContent> ... </complexContent>` tag pair. Inside this pair, either an `extension` or a `restriction` clause is specified. We use `extension` in the above example, which means that the specified element, `syllabus`, is to be added to the contents of the type `xyzCrs:courseType` to form the new type `courseType` (in the target namespace `http://foo.edu/Admin`).

`foo.edu` might need to make other changes to the schema. For instance, they might generally like the type `studentType` defined in the namespace `http://xyz.edu/Admin` (Figure 17.9, page 605), but not that it allows students to take any number of courses (because of `maxOccurs="unbounded"`). Thus, `foo.edu` decides to limit this number to 63 by *restricting* the original schema:

---

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xyzCrs="http://xyz.edu/Admin"
    xmlns:fooAdm="http://foo.edu/Admin">
    targetNamespace="http://foo.edu/Admin">

    <import namespace="http://xyz.edu/Admin"/>
    .

    <complexType name="studentType">
        <complexContent>
            <restriction base="xyzCrs:studentType">
                <sequence>
                    <element name="Name" type="xyzCrs:personNameType"/>
                    <element name="Status" type="xyzCrs:studentStatus"/>
                    <element name="CrsTaken" type="xyzCrs:courseTakenType"
                        minOccurs="0" maxOccurs="63"/>
                </sequence>
                <attribute name="StudId" type="xyzCrs:studentId"/>
            </restriction>
        </complexContent>
    </complexType>
```

```
<!-- Now define a Student element for the target namespace  
and associate it with the derived type -->  
<element name="Student" type="fooAdm:studentType"/>  
:  
  
</schema>
```

Analogously to the type extension mechanism, we use the tag **restriction** inside the **complexContent** block. The important difference, however, is that, when restricting a complex type, we must repeat all the element and attribute declarations from the base type. At the same time, we can impose restrictions on the components of the base type, for instance, by replacing **maxOccurs="unbounded"** with the more restrictive **maxOccurs="63"**. Thus, a restriction of a complex base type has exactly the same overall structure as the base type except that some elements and attributes comprising the restriction may be subsets of the corresponding ranges of the base type.

#### 17.3.4 Putting It Together

We have now defined a large number of simple and complex types, and we are ready to put them together to form a complete schema, which can describe document instances such as the report in Figure 17.4 on page 578. Such a schema requires a number of type definitions and at least one declaration of a **global element**. One of these global elements typically serves as the root element of the document instances described by the schema (e.g., the **Report** element in Figure 17.8). The others can be elements that are used in the definition of the type of the root. We will discuss global elements more fully on page 607. In our example, a single global declaration associates the root element, called **Report**, with its type, **adm:reportType**. This complex type contains declarations of other elements and attributes together with their types. Starting with the root element, then, we can descend into its type and find all of its elements and attributes. Repeating this recursively, we can find the elements and attributes at any depth in the document structure. A complete schema for our example (whose parts are defined elsewhere and inserted using the **include** statement) is shown in Figure 17.8.

We omit the definition of the lower-level types **classOfferings** and **courseCatalog**, which are defined similarly to **studentList**. Like **studentList**, these types are defined in terms of the types shown in Figures 17.9 and 17.10, which are the targets of the **include** statement in Figure 17.8.

As before, we must be careful about the namespaces, so we define **adm** as a prefix for the target namespace and use it in all references to the names defined in this schema (except in the statements that define these names using the attribute **name**). Recall that the including and included schemas are required to have the same namespace, so one prefix, **adm**, suffices to refer both to the names defined by the including schema (e.g., **adm:courseCatalog**) and the names defined in the included schemas (e.g., **adm:studentType**).

**FIGURE 17.8** A complete schema.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:adm="http://xyz.edu/Admin"
  targetNamespace="http://xyz.edu/Admin">

  <!-- The following schemas are shown in Figures 17.9, 17.10, and 17.11 -->
  <include schemaLocation="http://xyz.edu/StudentTypes.xsd"/>
  <include schemaLocation="http://xyz.edu/ClassTypes.xsd"/>
  <include schemaLocation="http://xyz.edu/CourseTypes.xsd"/>

  <element name="Report" type="adm:reportType"/>

  <complexType name="reportType">
    <sequence>
      <element name="Students" type="adm:studentList"/>
      <element name="Classes" type="adm:classOfferings"/>
      <element name="Courses" type="adm:courseCatalog"/>
    </sequence>
  </complexType>
  <complexType name="studentList">
    <sequence>
      <element name="Student" type="adm:studentType"
        minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>

  <!-- Plus the definition of classOfferings, courseCatalog -->
  <!-- The definition of studentType is in the included schema
       http://xyz.edu/studentTypes.xsd -->
</schema>
```

---

**FIGURE 17.9** Student types at <http://xyz.edu/StudentTypes.xsd>.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:adm="http://xyz.edu/Admin"
        targetNameSpace="http://xyz.edu/Admin">

    <complexType name="studentType">
        <sequence>
            <element name="Name" type="adm:personNameType"/>
            <element name="Status" type="adm:studentStatus"/>
            <!-- courseTakenType is defined in Figure 17.10 -->
            <element name="CrsTaken" type="adm:courseTakenType"
                    minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
        <attribute name="StudId" type="adm:studentId"/>
    </complexType>
    <complexType name="personNameType">
        <sequence>
            <element name="First" type="string"/>
            <element name="Last" type="string"/>
        </sequence>
    </complexType>
    <simpleType name="studentStatus">
        <restriction base="string">
            <enumeration value="U1"/>
            <enumeration value="U2"/>
            :
            <enumeration value="G5"/>
        </restriction>
    </simpleType>

    <simpleType name="studentId">
        <restriction base="ID">
            <pattern value="[0-9]{9}" />
        </restriction>
    </simpleType>
    <simpleType name="studentIds">
        <list itemType="adm:studentRef"/>
    </simpleType>
    <simpleType name="studentRef">
        <restriction base="IDREF">
            <pattern value="[0-9]{9}" />
        </restriction>
    </simpleType>
</schema>
```

**FIGURE 17.10** Course types at <http://xyz.edu/CourseTypes.xsd>.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:adm="http://xyz.edu/Admin"
        targetNameSpace="http://xyz.edu/Admin">

    <complexType name="courseTakenType">
        <attribute name="CrsCode" type="adm:courseRef"/>
        <attribute name="Semester" type="string"/>
    </complexType>
    <complexType name="courseType">
        <sequence>
            <element name="Name" type="string"/>
        </sequence>
        <attribute name="CrsCode" type="adm:courseCode"/>
    </complexType>

    <simpleType name="courseCode">
        <restriction base="ID">
            <pattern value="[A-Z]{3}[0-9]{3}"/>
        </restriction>
    </simpleType>
    <simpleType name="courseRef">
        <restriction base="IDREF">
            <pattern value="[A-Z]{3}[0-9]{3}"/>
        </restriction>
    </simpleType>
</schema>
```

### 17.3.5 Shortcuts: Anonymous Types and Element References

We will now present two constructs that can help reduce the size and complexity of a schema document.

**Anonymous types.** All types defined so far were **named types** because each type definition had an associated name, and every element was explicitly associated with a named type. Naming is useful when we expect to share the same type among several definitions of elements or attributes. In many cases, however, a type might be one of a kind and not expected to be reused. For instance, in the above combined schema for the report document, the type `reportType` (as well as several other types such as `studentList` and `classOfferings`) is not shared. In this case, **anonymous types** can be a convenient shortcut.

Anonymous types are defined similarly to named types, except that the `name` attribute is not used and the type definition must be attached to the appropriate

element or attribute definition that uses it. These definitions with attached anonymous types are also slightly different from definitions of named types. First, they do not use the type attribute to introduce the anonymous type. Second, instead of the empty tags <element ... /> and <attribute ... />, they use tag pairs, and the definition of the anonymous type is physically enclosed by the opening and closing tag. Thus, we can change the definition of the element Report in our schema to use an anonymous type as follows:

---

```
<element name="Report">
  <complexType>
    <sequence>
      <element name="Students" type="adm:studentList"/>
      <element name="Classes" type="adm:classOfferings"/>
      <element name="Courses" type="adm:courseCatalog"/>
    </sequence>
  </complexType>
</element>
```

---

Similarly, we can change the definitions of the elements Students, Classes, and Courses to use anonymous types. In this case, the contents of the corresponding type definitions would be physically included in the above schema.

**Referencing global elements.** We conclude the discussion of the facilities for type definition in XML Schema with a mention of yet another shortcut, **global element referencing**, which is frequently used in schema definitions.

The overall scenario in which element referencing is used is as follows. First, a global element is defined as usual. A **global element** definition is one that appears as a direct child of the schema tag (not inside of any type definition). For instance, the element Report in Figure 17.8 is global. The ref attribute of an element tag allows us to include any global element in any type definition.

To illustrate, suppose that we want to use the element Comment, defined as

---

```
<element name="Comment" type="string"/>
```

---

17.3

in several parts of the schema in Figure 17.8 (for instance, both in reportType and studentList types). To do so, we would place (17.3) as a child of schema (for example, right after the definition of the element Report) and then place

---

```
<element ref="Comment"/>
```

---

17.4

in each place where a comment element is to appear. For instance, the following modification of our previous definition includes Comment as part of reportType:

---

```

<complexType name="reportType">
  <sequence>
    <element ref="Comment"/>
    <element name="Students" type="adm:studentList"/>
    <element name="Classes" type="adm:classOfferings"/>
    <element name="Courses" type="adm:courseCatalog"/>
  </sequence>
</complexType>

```

---

One can say that we have not achieved a great deal of savings through the use of the reference facility since (17.4) is not much shorter than the full definition (17.3). Nevertheless, if the `Comment` element needs to be inserted in many places, the referencing facility can provide a degree of consistency.

### 17.3.6 Integrity Constraints

In Section 17.3.2 we touched upon the issue of referential integrity in XML documents and showed how the XML Schema specification improves upon DTDs in this regard. Even in XML Schema, however, we still used the special types `ID` and `IDREF` that are inherited from DTDs. In XML Schema, the types `ID` and `IDREF` can be given to elements as well as attributes, which is already an improvement over DTDs. Still, these types are inadequate for representing integrity constraints. First, `ID` values must be globally unique. More importantly, the `ID` type cannot represent multiattribute keys. To illustrate, consider Figure 17.11, which shows a definition for the type `classType`.

If a student claims to have taken a course using a `CrsTaken` element (of the type declared in Figure 17.10), the corresponding course should have been offered in the

**FIGURE 17.11** <http://xyz.edu/ClassTypes.xsd>: type for the element `Class` in Figure 17.4, page 578.

```

<element name="Class" type="adm:classType"/>
<complexType name="classType">
  <sequence>
    <!-- courseCode type is defined in Figure 17.10 -->
    <element name="CrsCode" type="adm:courseCode"/>
    <element name="Semester" type="string"/>
    <element name="ClassRoster" type="adm:classListType"/>
  </sequence>
</complexType>
<complexType name="classListType">
  <!-- studentIds is defined in Figure 17.9 -->
  <attribute name="Members" type="adm:studentIds"/>
</complexType>

```

---

specified semester. Such offerings are described by **Class** elements in the instance document. Given an element such as

---

```
<CrsTaken CrsCode="CS308" Semester="F1997"/>
```

---

there must exist an element of the form

---

```
<Class>
  <CrsCode>CS308</CrsCode><Semester>F1997</Semester>
  :
</Class>
```

---

The problem is that neither **CrsCode** nor **Semester** alone uniquely determines the **Class** element, so the ID/IDREF mechanism is inapplicable.

**XML keys.** To address the above problems, the XML Schema specification allows general multiattribute keys and foreign-key constraints in a way that resembles SQL. There is a slight complication, however. SQL deals with flat relations, so to specify a key we simply list the attributes that belong to that key. Similarly, to specify a foreign-key constraint in SQL we simply specify a sequence of attributes in both the referencing and the referenced relation. In XML, we are dealing with complex structures, and the notion of a key is more involved. Indeed, a key might be composed of a sequence of values located at different depths inside an element.

Assuming that the frame of reference is the parent element of the **Class** elements, we can say that the key of the collection of **Class** elements is composed of values reachable using the pair of path expressions **Class/CrsCode** and **Class/Semester**. The idea of path expressions is familiar to us from Chapter 16, but in XML they take a more elaborate form. In fact, XML path expressions are part of another specification, called **XPath**, which we study in Section 17.4.1.

To see how complicated a key specification can be, let us expand the definition of **Class** in the schema by adding sections and splitting the season from the year in semester names. Then, in an instance document we might have the following class:

---

```
<Class>
  <CrsCode Section="2" Number="CS308"/>
  <Semester><Term>Fall</Term><Year>1997</Year></Semester>
  :
</Class>
```

---

Here the set of values that uniquely determines the class is scattered in different places (in attributes and in element content) and at different levels (in the attributes of the tag **CrsCode** and in the **Term** and **Year** children of the **Semester** element). The

path expressions needed to reach each of these components are specified in XPath as follows:

---

```
CrsCode/@Section  
CrsCode/@Number  
Semester/Term  
Semester/Year
```

---

All of these path expressions are relative to **Class** elements in the report document. The first selects the value of the attribute **Section** of the tag **CrsCode**, which must be a child of the current element (assumed to be **Class**). The second selects the value of the attribute **Number** of the **CrsCode** element. The third and fourth select the contents of the elements **Term** and **Year**, respectively, which must be children of the element **Semester**, which in turn must be a child of the current element.

XML Schema provides two ways to specify a key. One uses the tag **unique** and is similar to the **UNIQUE** constraint in SQL; it specifies *candidate keys*, in the terminology of Chapter 3. The other uses the tag **key** and corresponds to the **PRIMARY KEY** constraint in SQL. The only difference between **unique** and **key** is that keys cannot have *null values*. (In XML, the value of an element of the form **<footag></footag>** is an empty string and not necessarily a null.) For **footag** to have a null value (called a **nil** in XMLSchema) the following is used:

---

```
<footag xsi:nil="true"></footag>
```

---

Here **nil** is a symbol defined in the namespace

---

```
http://www.w3.org/2001/XMLSchema-instance
```

---

(and we assume that **xsi** is a prefix that has been defined to refer to that namespace).

Next is an example of a primary-key declaration for the report document. Declaring candidate keys is similar, except that the tag **unique** is used instead of the tag **key**. Referring to the type **classType** of Figure 17.11 (not the more elaborate type with section numbers that we just discussed), we want to specify that the pair of values of tags **CrsCode** and **Semester** uniquely identifies the **Class** element within the document. To show how this is done we elaborate on the earlier schema in Figure 17.8 on page 604.

---

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"  
        xmlns:adm="http://xyz.edu/Admin">  
    targetNameSpace="http://xyz.edu/Admin">  
  
    <include schemaLocation="http://xyz.edu/StudentTypes.xsd"/>  
    <include schemaLocation="http://xyz.edu/ClassTypes.xsd"/>  
    <include schemaLocation="http://xyz.edu/CourseTypes.xsd"/>
```

```
<element name="Report" type="adm:reportType"/>

<complexType name="reportType">
    <sequence>
        <element name="Students" type="adm:studentList"/>
        <element name="Classes">
            <!-- Replacing adm:classOfferings with anonymous type -->
            <complexType>
                <sequence>
                    <!-- adm:classType is defined in Figure 17.11 and
                        included with http://xyz.edu/ClassTypes.xsd -->
                    <element name="Class" type="adm:classType"
                            minOccurs="0" maxOccurs="unbounded"/>
                </sequence>
            </complexType>
        </element>
        <element name="Courses" type="adm:courseCatalog"/>
    </sequence>
</complexType>
:
</schema>
```

The above schema lists the relevant type definitions for our report document. The namespace declarations and the include statements have already been discussed. The type **reportType** is used for the root element, **Report**. It is a sequence of three elements: **Students**, **Classes**, and **Courses**. Their types were defined in Figures 17.9, 17.11, and 17.10 on pages 605, 608, and 606, respectively. The definition of **courseCatalog**, the type for the element **Courses**, is an easy exercise.

The most interesting feature here is the key declaration specified with a **key** tag; it is named **PrimaryKeyForClass** using the attribute **name**. Declaration of a key is always part of an element specification. However, observe that **PrimaryKeyForClass** appears in the definition of the element **Classes** rather than of **Class**, even though the key involves only the components of type **classType** and refers to **Class** elements only. This is intentional, to illustrate the point that XML key declarations are associated with collections of objects (which typically are sets of elements), and these objects can be identified using XPath expressions regardless of where the key

definition occurs. The `xpath` attribute of the `selector` tag specifies a path expression (relative to the element that contains the key declaration), which identifies the collection of objects to which the key declaration applies. In our case, the selecting path expression is `Class`; it is relative to the element `Classes` because the key declaration appears as a child of this element declaration. The collection identified by the selector is simply the set of all `Class` elements in the document.

Having identified the appropriate collection of objects, we use the subsequent `field` elements to specify the fields that constitute the key. As explained earlier, these fields can come from different places in an object and can be nested in complex ways. In our case, however, things are simple: the first field in the key is the contents of the child element `CrsCode` of the element `Class`, and the second field is the contents of the child element `Semester`. (Path expressions specified in the `xpath` attribute of the `field` clause are relative to the collection of the objects determined by the selector. This is why, for example, the first path expression is simply `CrsCode` rather than `Class/CrsCode`.) Note that, for a path expression to make sense as a specification of a field in a key, it must return precisely one value for each object to which it applies. For instance, the path expression `CrsCode` returns precisely one value for any given `Class` element, so the field specification

---

```
<selector xpath="Class"/>
<field xpath="CrsCode"/>
:
:
```

---

is allowed. In contrast, the path expression `CrsTaken/@CrsCode` within the scope of a `Student` element can return a set of courses taken by the student (refer to `studentType` defined in Figure 17.9 on page 605), so the field specification

---

```
<selector xpath="Student"/>
<field xpath="CrsTaken/@CrsCode"/>
:
:
```

---

is not allowed within the scope of the element `Students`.

**Foreign-key constraints in XML.** Next, we want to be able to state that every element `CrsTaken` in a student record refers to an actual class element in the same report. This is akin to a foreign-key constraint and is defined using the `keyref` element, as depicted in Figure 17.12.

A foreign-key constraint has a name, a reference identifier, a selector, and a list of fields. The name of a foreign key is of little importance. The reference is defined using the attribute `refer`, and its value must match the name of a key or unique constraint. In our case, it matches the key constraint, `PrimaryKeyForClass`, defined within the scope of the element `Classes`. In SQL, this corresponds to the `REFERENCES relation-name` part of a foreign-key constraint. Next comes the selector.

**FIGURE 17.12** Part of a schema with a key and a foreign-key constraint.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:adm="http://xyz.edu/Admin"
  targetNameSpace="http://xyz.edu/Admin">

  <complexType name="courseTakenType">
    <attribute name="CrsCode" type="adm:courseRef"/>
    <attribute name="Semester" type="string"/>
  </complexType>
  <complexType name="classType">
    <sequence>
      <element name="CrsCode" type="adm:courseCode"/>
      <element name="Semester" type="string"/>
      <element name="ClassRoster" type="adm:classListType"/>
    </sequence>
  </complexType>

  <complexType name="reportType">
    <sequence>
      <element name="Students" type="adm:studentList"/>
      <keyref name="NoEmptyClasses" refer="adm:PrimaryKeyForClass">
        <selector xpath="Student/CrsTaken"/>
        <field xpath="@CrsCode"/>
        <field xpath="@Semester"/>
      </keyref>
    </element>
      <element name="Classes" type="adm:classOfferings"/>
      <key name="PrimaryKeyForClass">
        <selector xpath="Class"/>
        <field xpath="CrsCode"/>
        <field xpath="Semester"/>
      </key>
    </element>
      <element name="Courses" type="adm:courseCatalog"/>
    </sequence>
  </complexType>
</schema>
```

As in the case of the key constraint, it identifies a **source collection** of elements through its `xpath` attribute. In our case, the collection in question consists of all `CrsTaken` elements. Each of these is supposed to reference the key of an object from the **target collection** specified in the key constraint `PrimaryKeyForClass`.

Finally, we have to specify the foreign key itself, that is, the fields inside the CrsTaken elements (the source collection) that actually reference the fields in the target collection (specified in the key constraint). We do this using the already familiar field tag. As before, this tag provides a path expression (relative to the selected collection of objects) that leads to a value. We want the attributes CrsCode and Semester of the source collection of CrsTaken elements to refer to the fields that constitute the key of the target collection of Class elements. In XPath, we use the path expressions @CrsCode and @Semester to identify these attributes (Figure 17.10). As with the fields that form a key, each path expression in a foreign key (@CrsCode and @Semester in our case) must yield a single value when applied to an object in the source collection.

In a similar way, we can specify other key and foreign-key constraints in the report document and replace the constraints previously specified using the ID and IDREF data types (see Exercises 17.5 and 17.7).

We should note that it is not clear how to specify IDREFS-style referential integrity with the help of the key and keyref tags. For instance, the attribute Members in ClassRoster (Figure 17.11) has the type studentIds, which is a list of values of type studentRef. Since studentRef is derived by restriction from the base type IDREF (see Figure 17.9 on page 605), studentIds can be seen as a specialized version of IDREFS. We can try to specify the desired referential integrity using something like this:

---

```
<keyref name="RosterToStudIdRef" refer="adm:studentKey">
  <selector xpath="Class"/>
  <field xpath="ClassRoster/@Members"/>
</keyref>
```

---

where studentKey is an appropriately defined key constraint for Student elements:

---

```
<key name="studentKey">
  <selector xpath="Student"/>
  <field xpath="@StudId"/>
</key>
```

---

The problem here is that the value of the attribute Members is a *list* while the value of the key attribute StudId in the Student tag is a *single* item. Unfortunately, it is not possible in XPath to create a reference from the individual components of a list data type (represented by the Members attribute) to other entities in the document (i.e., student Ids defined in the Student elements). The only solution is to use a representation where student Ids are not in a list but occur as individual elements (Exercise 17.9).

## 17.4 XML Query Languages

Why would you want to query an XML document? Will databases soon begin storing XML and speak it fluently?

Storing XML documents in a database specifically designed for this kind of data is one possibility—methods exist for efficient storage and retrieval of tree-structured objects, including XML documents [Deutsch et al. 1999; Zhao and Joseph 2000]—and major relational database vendors are beginning to offer an option for native storage for XML documents. Native storage is also supported by SQL/XML, a forthcoming standard for interoperability between SQL databases and XML (see Section 17.4.2). In this capacity, SQL/XML can be seen as an alternative way of storing objects in a database (the other alternative being the SQL object-relational extensions).

XML documents can also be stored by mapping them to the relational or object-oriented format. Utilities that perform such mapping automatically are widely available and are part of most major database products. Such databases can receive XML documents and convert them into relations or objects; they also provide tools for generating XML from the data already stored in the database. Once generated, an XML document is transmitted to another machine, which either presents it to the user or processes it automatically. To help with this task, the W3C has developed the **document object model (DOM)** for XML [DOM 2000], which standardizes the API by which a client application can access various parts of an XML document and thereby simplifies the task of writing such applications.

What does a query language have to do with all this? First, if XML is stored natively in a database, one needs a way to query it. The second possibility is even more intriguing. Imagine that you are preparing your next semester's schedule and need to find all courses offered in that semester between 3 PM and 7 PM. If the university database server lets you pose such a query, you are in good shape, but more likely it provides a fixed interface that supports only a limited set of queries. In this case, finding what you want might require a tedious process of filling out a series of forms and eyeballing the results, and you might also have to use low-tech instruments, such as pen and paper, to record the needed information.

An alternative is to ask the server for an XML document containing the list of all courses offered this semester and have a client application find the desired information. As mentioned, DOM simplifies this task considerably. Still, it provides only a low-level interface to XML. If your query requires joining information stored in different parts of the document or in separate documents, you might end up writing a fairly large program (and the semester will be over by the time you debug it). An analogy here is using nested loops and IF-statements to replace a complex SQL query. We will see how a powerful, high-level query language can simplify this task, enabling a new class of client applications capable of processing information in an intelligent and custom-tailored way.

In the remainder of Section 17.4, we discuss two query languages for XML: **XPath** [XPath 1999] and **SQL/XML**. The first is an official W3C recommendation, and the last is going to be part of the forthcoming SQL:2003 standard.

XPath is intended to be simple and efficient. It is based on the idea of path expressions, with which we became familiar in Chapter 16, and is designed so that queries are compact and can be incorporated into URLs. SQL/XML is an extension of SQL designed to provide interoperability between data stored in relational databases and XML documents.

### 17.4.1 XPath: A Lightweight XML Query Language

In an object-oriented language, such as SQL:1999, a path expression is a sequence of object attributes that provides the exact route to a data element nested deep within the object structure. The requirement to provide an exact route is not a problem when the schema of the database is known to the programmer and is not likely to change. However, when the schema is not known and the structure of data needs to be explored (which is often the case in Web applications), merely adopting path expressions from object-oriented languages is not enough.

XPath extends path expressions with query facilities by allowing the programmer to replace parts of the route to data elements with search conditions. By then examining the data, the XPath interpreter is supposed to find the missing parts of the route at run time. The idea of augmenting path expressions with queries is not new. It appeared in [Kifer and Lausen 1989; Kifer et al. 1992; Frohn et al. 1994] in the context of object-oriented databases and was further developed in works on semistructured data, such as [Buneman et al. 1996; Abiteboul et al. 1997; Deutsch et al. 1998; Abiteboul et al. 2000]. XPath was built on these ideas and became an important basis for many XML extensions.

**The XPath data model.** XPath views XML documents as trees and views elements, attributes, comments, and text as nodes of those trees. There is a special **root node** in the tree, which should not be confused with the root element of an XML document. This is illustrated in Figures 17.13 and 17.14. The XML instance document in Figure 17.13 (itself a fragment of the report document in Figure 17.4, page 578) is the basis for the XPath document tree in Figure 17.14.

Note that the root node of the XPath tree is different from the node that corresponds to `Students`, which is the root element of the document. The need for the special root node is apparent from Figure 17.14: it serves as a gathering point for all of the document components, including the comments that are allowed to occur outside the scope of the root element.

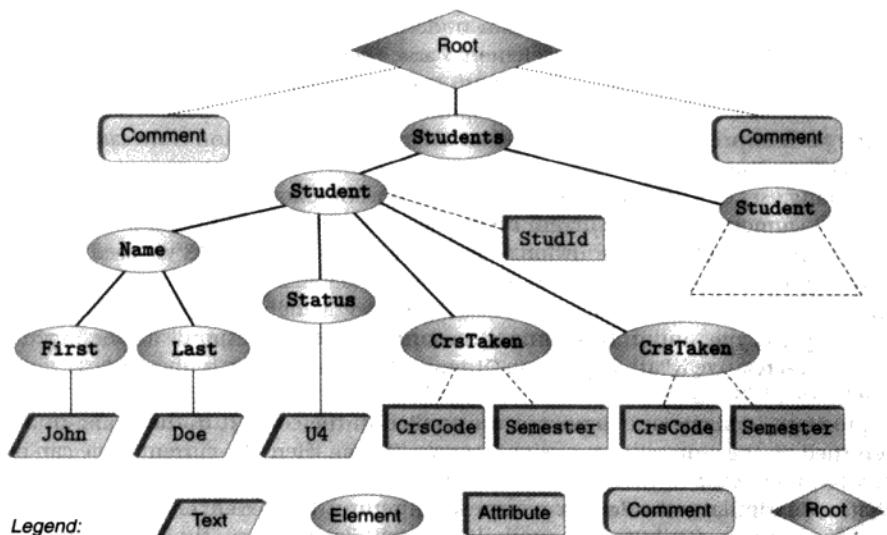
As usual in a tree, every node except the root node has a parent. A node,  $P$ , immediately above another node,  $C$ , is the **parent** of that node, and  $C$  is a **child** of  $P$ . However, the XPath specification has an important and sometimes confusing exception: an attribute is *not considered a child of its parent node*. That is, if  $C$  corresponds to an attribute of  $P$ , then  $P$  is a parent of  $C$ , but  $C$  is not a child of  $P$ . Because of the potential confusion due to the peculiar XPath terminology, we use the standard terminology for tree data structures and do regard attributes as children of their parents. To avoid ambiguity, we sometimes talk about **e-children**,

**FIGURE 17.13** Fragment of the report document in Figure 17.4, page 578.

```

<?xml version="1.0" ?>
<!-- Some comment -->
<Students>
    <Student StudId="s111111111">
        <Name><First>John</First><Last>Doe</Last></Name>
        <Status>U4</Status>
        <CrsTaken CrsCode="CS308" Semester="F1997"/>
        <CrsTaken CrsCode="MAT123" Semester="F1997"/>
    </Student>
    <Student StudId="s987654321">
        <Name><First>Bart</First><Last>Simpson</Last></Name>
        <Status>U4</Status>
        <CrsTaken CrsCode="CS308" Semester="F1994"/>
    </Student>
</Students>
<!-- Some other comment -->

```

**FIGURE 17.14** XPath document tree corresponding to document in Figure 17.13.

**a-children**, and **t-children** when we want to restrict attention to the particular type of children: elements, attributes, or text. For example, `Name` is an *e*-child of `Student`, `StudId` is an *a*-child of `Student`, and `John` is a *t*-child of `First`. When we want to include both element children and text children, we refer to **et-children**. Similarly, **ta-children** refers to text and attribute children, and so on.

Another peculiarity of the XPath data model is that text that occurs inside XML elements (e.g., `John`, `U4`) is represented by a node in the tree. However, text that represents attribute values (e.g., the value `s987654321` of the attribute `StudId`) is not deemed to be worthy of a tree node of its own.

The XPath data model provides operators for navigating the document and accessing its various components. These operators include accessing the root of the XPath tree, the parent of a node, its children, the contents of an element, the value of an attribute, and the like.

We saw some of these operators when we discussed constraints in XML Schemas. The basic syntax is that of the UNIX file naming schema: The symbol "/" represents the root of the XPath tree, "." represents the current node, and ".." represents the parent node of the current node. An XPath expression takes a document tree and returns a list of nodes in the tree. The path expression `/Students/Student/CrsTaken` is **absolute**; it returns the set of nodes that correspond to the elements `CrsTaken`, which are reachable from the root through a `Students` child and a `Student` grandchild. Our tree has three such `CrsTaken` nodes (one is not shown in the figure). If the current node corresponds to the element `Name`, then `First` and `./First` both refer to the same child element. If the current node corresponds to the element `First`, then `../Last` is the sibling node corresponding to the element `Last`. These are **relative path expressions** since their departure point for navigation is the *current node* rather than the root.

Many uses of XPath provide some notion of a context in which one of the nodes in the document tree is the **current node**. We have already seen this in the way XPath is used in XML Schema. For instance, in Figure 17.12 on page 613 we used relative path expressions in the definition of the primary key `PrimaryKeyForClass` and of the foreign key `NoEmptyClasses`. The primary key is defined as part of the XML type `reportType`, and in this context the current node corresponds to the `Report` element. A relative XPath expression `Classes/Class` is applied to that node to yield the set of all `Class` nodes that are grandchildren of `Report`. The two relative XPath expressions `CrsCode` and `Semester` are applied in the context of the `Class` nodes returned by the aforesaid expression `Classes/Class`. Here the current node can be any of these `Class` nodes, and the `CrsCode` expression returns the `CrsCode` child of that node; similarly the `Semester` expression returns the `Semester` child.

To access an attribute, the symbol "@" is used. For instance, the list of attribute nodes that correspond to `CrsCode` in the document of Figure 17.13 is obtained using the path expression `/Students/Student/CrsTaken/@CrsCode`. In our case, this list consists of three nodes because the `CrsCode` attribute occurs three times: with the values `CS308`, `MAT123`, and `CS308` (note the repetition due to the fact that the same value occurs more than once in different attribute nodes). Text nodes are accessed using the `text()` function. For instance, `/Students/Student/Name/First/text()`

represents the collection of nodes, each representing the text content of an element of type `First`. We have two such nodes in our document; one corresponds to John and the other to Bart. If you were wondering, the two comment nodes in Figure 17.13 can be selected using the expression `/comment()`.

**Advanced navigation in XPath.** The more advanced features of XPath navigation include facilities to select specific nodes of an XML document as well as facilities to jump through an indeterminate number of children. For instance, to select the second course taken by John Doe, we use the expression `/Students/Student[1]/CrsTaken[2]`. Here, `/Students/Student[1]` selects the first of the two `Student` nodes in the document tree. The expression `CrsTaken[2]` then selects the second `CrsTaken` *e-child* in that first `Student` node. Another example of selection of a particular node is `/Students/Student/CrsTaken[last()]`. This is similar to the above except that the prefix `/Students/Student` selects all nodes that correspond to `Student` and `CrsTaken[last()]` then chooses the last `CrsTaken` node under each selected `Student` node. In our case, the above expression returns

---

```
<CrsTaken CrsCode="MAT123" Semester="F1997"/>
<CrsTaken CrsCode="CS308" Semester="F1994"/>
```

---

At times, we might not know the exact structure of the document, or specifying the exact navigation path might be cumbersome, so XPath provides several wildcard facilities. One is the *descendant-or-self* operation, `//`, illustrated by the expression `//CrsTaken`, which is an absolute path expression that starts at the root and selects all `CrsTaken` elements in the entire tree. In our particular case, the effect is the same as that produced by `/Students/Student/CrsTaken`. However, if the document contains elements `CrsTaken` nested under different types of elements and at different depths, then selecting all such elements without a wildcard is hard and unwieldy. Similarly, `/Students//CrsTaken` selects all `CrsTaken` elements that are descendants of `Students` nodes regardless of the nesting level.

The descendant operation can be used in relative expressions as well. For instance, `.//CrsTaken` searches through all descendants (or self) of the current node to find the `CrsTaken` elements. Observe that `./CrsTaken` and `CrsTaken` are the same. However, `.//CrsTaken`, `CrsTaken`, and `//CrsTaken` are all different: the first expression returns all `CrsTaken` descendants at the current node (or the current node itself, if it is a `CrsTaken` element); the second expression returns only the `CrsTaken` children at the current node; and the third, all `CrsTaken` elements found anywhere in the document.

XPath also allows searching through all ancestors (parent, grandparent, etc.) of any given node, but we omit this wildcard.

The third wildcard, `*`, lets us collect all *e-children* of a node irrespective of type. For instance, `Student/*` selects all *e-children* of the `Student` children of the current node. (If the current node is the (only) `Students` node, the wildcard selects the two `Name` nodes, the two `Status` nodes, and the three `CrsTaken` nodes.) The

expression `/*/*` selects all *e*-children of the root and their *e*-descendants (since `//` is descendant-or-self, the set of nodes `/*/*` includes the set of children nodes of the root, `/*`).

The `*` wildcard can also be applied to attributes. For instance, `CrsTaken/@*` selects all attribute values of the `CrsTaken` nodes that sit below the current node. Note that `*` does not include the text nodes that could possibly exist among the children of the `Student` element. To select those, the expression `Student/text()` is used.

**XPath semantics.** The general form of an XPath query is

---

`locationStep1/locationStep2/... or /locationStep1/locationStep2/...`

---

where each location step is of the form `axis::nodeSelector [selectionCondition]`. The term `axis` refers to the **navigation axis**, which indicates the direction along which navigation is taking place in the corresponding location step. The available axes are `child` (i.e., go to a child node), `parent`, `descendant` (i.e., child, grandchild, etc.), `descendant-or-self`, etc. The node selector is either the name of the node (e.g., the name of an element or attribute; a selector for an unnamed node, such as `text()` or `comment()`; or a wildcard, such as `*` or `@*`). For instance, `child::Student` is a simple location step that directs navigation from the current node down to a child element named `Student`, and `descendant-or-self::@Semester` directs navigation from the current node to a `Semester` attribute in either the current node or in a descendant of that node. Because the full syntax of XPath is so verbose, most axes have convenient abbreviations, and this is what we have been using up until now (and will continue using). For instance, `child::Student` abbreviates as `Student`, the relative expression `descendant-or-self::@Semester` abbreviates as `./@Semester`, and the absolute expression `/descendant-or-self::@Semester` as `//@Semester`. The optional `selectionCondition` in a location step selects a subset of nodes reachable by the location step. We will see examples of such conditions shortly.

The value of a location step `axis::nodeSelector [selectionCondition]` on an instance document is the set of all nodes of the form `nodeSelector` that are reachable by the navigation `axis` and that satisfy `selectionCondition`. For instance, `./@Semester` specifies the set of attribute nodes called `Semester`, which are reachable from the current node by the axis `descendant-or-self`. In plain English: the set of all `Semester` attributes that appear in either the current node or in one of its descendants.

For a path expression `locationStep1/locationStep2/...`, the value on a source document is the set of all document nodes computed as follows: From the current node, find all nodes reachable by `locationStep1`. For each such node, *N*, find the set of all nodes that are reachable from *N* via `locationStep2`. Take the union of all node sets reachable by `locationStep2`. Apply `locationStep3` to each node in the

union, etc., until the last location step is reached. The value of the path expression is the set of nodes obtained at this last step.

**XPath queries.** We are particularly interested in the features of XPath that give it the ability to select nodes using a query facility. XPath queries can include selection conditions at any step in the navigation process. To give meaningful examples of XPath expressions with queries, we go back to our report document in Figure 17.4, page 578.

Here is a simple example of a path expression that selects all student nodes where the student has taken a course in fall 1994:

---

```
//Student[CrsTaken/@Semester = "F1994"]
```

---

Here we have a wildcard expression, `//Student`, that selects all `Student` nodes under the root node. The expression inside the square brackets is a **selection condition** that eliminates the nodes that do not satisfy the condition by selecting a `Student` node only if the path expression `CrsTaken/@Semester` can be applied at this node and if it returns a set that *includes* `F1994`.<sup>7</sup> To select elements based on the content of an element rather than of an attribute, we use the following expression:

---

```
//Student[Status = "U3" and starts-with(./Last, "P")
        and not(./Last = ./First)]
```

---

This example introduces several important features:

1. Selection conditions can be combined using `and`, `or`, and `not`.
2. To select an element based on the content of one of its children or descendants, we simply equate the appropriate path expression with another such expression or a constant. Strictly speaking, we should have written `Status/text() = "U3"` instead of `Status = "U3"`, but since `Status` does not have subelements, XPath allows us to be less pedantic in this case. This is possible because, in order to evaluate a comparison such as the one above, XPath converts every node returned by the path expression into its **string value** and then compares strings. For simple element nodes such as those returned by `//Student/Status` or `//Student//Last`, the string value is simply the text inside the element. For attribute nodes, such as those returned by `//Student/CrsTaken/@Semester` the string value is the value of the attribute. A full set of rules that defines string values for the various nodes in an XPath tree is given in [XPath 1999].
3. XPath has a rich repertoire of functions that greatly increase its expressive power. For the full list of these functions, we refer you to the XPath specification [XPath 1999].

<sup>7</sup> Note that if a `Student` node has several `CrsTaken` children, then the path expression `CrsTaken/@Semester` returns a *set* of nodes.

In the above example, we use the built-in predicate `starts-with()` to select only those students whose last names start with P. To summarize the above query, it selects all students who have the status U3, whose last names start with P, and whose last and first names are different. The other string manipulation functions allow us to check for containment, perform concatenation, determine length, and so forth. For instance, the following query can be used to search for students who have "van" as part of their name:

---

```
//Student[contains(concat(Name//text()), "van"))]
```

---

Here `Name//text()` returns the set of all text nodes that are descendants of the `Name` element, and the concatenation function makes one string out of those nodes—in this case the student's first and last names. Then we check if the result contains `van` as a substring.

Aggregate functions available in XPath include `sum()` and `count()`. For example, the following selects the students who have taken at least five courses:

---

```
//Student[count(CrsTaken) &gt;= 5]
```

---

In this expression, `CrsTaken` returns the set of all e-children of type `CrsTaken` for the current node (which must be a `Student` node). Thus, `Count(CrsTaken)` returns the number of these children, which is compared with "5". The obscure `&gt;=` contraption, stands for `>=`. This complication is due to the fact that the symbols `<` and `>` must be encoded as `&lt;` and `&gt;` because `<` and `>` are reserved for tag delimiters.

It should be noted that selection conditions can be applied at different levels and multiple times in a path expression. Thus, the following is legal:

---

```
//Student[Status="U4"]/CrsTaken[@CrsCode="CS305"]
```

---

This expression selects all the `CrsTaken` elements in the document that occur in `Student` elements with status `U4` and whose `CrsCode` attribute has the value `CS305`.

Multiple selection conditions can also be applied at the same level in a path expression, as shown in the following expression that selects all `Student` elements having the property that the student took (among other courses) `MAT123` in fall 1994:

---

```
//Student/CrsTaken[@CrsCode="MAT123"][@Semester="F1994"]
```

---

The same expression can be written as

---

```
//Student/CrsTaken[@CrsCode="MAT123" and @Semester="F1994"]
```

---

Note that this expression is different from

---

```
//Student [CrsTaken/@CrsCode="MAT123" and CrsTaken/Semester="F1994"]
```

---

which selects students who either took MAT123 or took a course in the fall of 1994. The reason for this difference in the interpretation is that nothing in the latter expression tells us that the two occurrences of the CrsTaken expression select the same element node. The “or” connective—for example, CrsTaken/@CrsCode="MAT123" or CrsTaken/@Semester="F1994"—is also allowed.

There is one other interesting form of selection condition, one where a path expression is used as a predicate rather than as an argument to a predicate. Suppose that Grade is an optional attribute of CrsTaken. Then

---

```
//Student [CrsTaken/@Grade]
```

---

selects all student elements that have a CrsTaken child element with an explicitly specified Grade attribute (regardless of its value). Likewise,

---

```
//Student [Name/First or CrsTaken]
```

---

selects all Student elements that have either the element First as a grandchild or the element CrsTaken as a child.

Finally, recall that SQL allows the use of algebraic query operators, such as UNION and EXCEPT. XPath, being a frugal language, allows only the union operator, which is denoted by the symbol |, as in the expression

---

```
//CrsTaken[@Semester="F1994"] | //Class [Semester="F1994"]
```

---

The set of nodes selected by this query is a union of elements of different types: the CrsTaken elements that pertain to the fall 1994 semester and the Class elements that describe fall 1994 course offerings. This illustrates how a path expression can return a set containing elements of different types.

### 17.4.2 SQL/XML

SQL is not called Intergalactic Dataspeak for nothing. When OQL-speaking aliens (see Chapter 16) descended on our galaxy, SQL was extended with object-relational constructs. SQL is now being extended with a new dialect, called SQL/XML. This extension can be viewed as yet another way to introduce object-relational data into SQL—see Section 16.3 for an earlier proposal, which became part of SQL:1999. The major vendors have implemented most of the proposed extensions. When complete, it is expected that SQL/XML will become part of the SQL:2003 standard.

SQL/XML addresses the following practical needs:

- To publish the contents of SQL tables and even the contents of entire databases as XML documents. This requires the development of conventions for translation of the primitive SQL data types, such as CHAR(4), into XML Schema data types and back.
- More generally, to create XML documents out of SQL query results. This requires the addition of primitives to allow the creation of XML elements by SQL queries.
- To store XML documents in relational databases and to query them effectively using SQL. Since SQL is unable to deal with tree-like structures directly, XPath is used for that purpose.

In the remainder of this section, we review the current state of SQL/XML. Keep in mind, however, that the work on this standard is still ongoing, that some details will definitely change by the time this text is published, and that vendor implementations (e.g., from Oracle, IBM, and Microsoft) may vary slightly from the SQL/XML proposal.

### Encoding Relations as XML Documents

This part of the SQL/XML specification defines the conventions for converting relations into XML documents (and relation schemas into XML schemas). The main purpose is to provide a standard way of exchanging relations on the Internet. The current proposal does not include a built-in function that would take a table and return an XML document. However, it provides more general functions that make it possible to create arbitrary XML documents using the SELECT clause. We discuss these functions in the next subsection, on page 627.

There are many ways to encode relational data in XML (see Exercise 17.1). SQL/XML does this as follows:

- The entire relation is enclosed in a pair of tags named after the relation.
- Each row is enclosed within the `row` tag pair.
- Each attribute value is enclosed within a pair of tags named after that attribute.

For instance, the PROFESSOR relation in Figure 3.5 on page 39 will be represented as

---

```
<Professor>
  <row>
    <Id>101202303</Id>
    <Name>John Smyth</Name><DeptId>CS</DeptId>
  </row>
  <row>
    <Id>783432188</Id>
    <Name>Adrian Jones</Name><DeptId>MGT</DeptId>
  </row>
  <row>
```

```

<Id>121232343</Id>
<Name>David Jones</Name><DeptId>EE</DeptId>
</row>
:
</Professor>

```

---

Suppose the type of the Id attribute in the PROFESSOR relation is INTEGER and the types for Name and DeptId are CHAR(50) and CHAR(3), respectively. Defining an XML Schema document corresponding to the relational schema of the PROFESSOR relation is not hard. The only question is the representation of SQL types in XML Schema. In our particular case, this representation is easy:

```

<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tnc="http://xyz.edu/Admin"
    targetNamespace="http://xyz.edu/Admin">
    <element name="Professor">
        <complexType>
            <sequence>
                <element name="row" minOccurs="0" maxOccurs="unbounded">
                    <complexType>
                        <sequence>
                            <element name="Id" type="integer"/>
                            <element name="Name" type="CHAR_50"/>
                            <element name="DeptId" type="CHAR_3"/>
                        </sequence>
                    </complexType>
                </element>
            </sequence>
        </complexType>
    </element>
</schema>

```

---

The types CHAR\_50 and CHAR\_3 are standard conventions within SQL/XML for the corresponding SQL CHAR(...) types. For example, CHAR\_50 is defined by restricting the base XML Schema type string as follows:

```

<simpleType name="CHAR_50">
    <restriction base="string">
        <length value="50"/>
    </restriction>
</simpleType>

```

---

The real problem is that SQL has a large number of built-in types, such as INTERVAL, TIMESTAMP, MULTISET, etc., as well as user-defined types, which are created using the CREATE DOMAIN statement. All of these need to be painstakingly defined in XML, and much of the XML/SQL specification deals with this issue. We omit the gory details in this text.

### Storing and Manipulating XML in Relational Databases

**The XML data type.** Although an XML document could be stored inside a table as an attribute of type string, doing so would make querying the document extremely inefficient. For example, XPath would have to scan the entire string and parse it before an expression could be evaluated. Hence, SQL/XML envisions support for *native storage* of an XML document as a hierarchical tree structure, which facilitates navigation within the document. For example, the structure would make it easy to locate the children of each node. Fortunately, efficient storage and indexing techniques exist to support native storage of XML documents [Deutsch et al. 1999; Zhao and Joseph 2000], and a new data type, XML, was added to SQL for this purpose. For instance, we might decide to store transcripts in the native XML format along with the student information, as follows:

---

```
CREATE TABLE STUDENTXML (
    Id      INTEGER,
    Details XML )
```

---

17.6

The Details attribute is supposed to contain XML documents of the form

---

```
<Student>
    <Name><First>John</First><Last>Doe</Last></Name>
    <Status>U2</Status>
    <CrsTaken CrsCode="CS308" Semester="F1997"/>
    <CrsTaken CrsCode="MAT123" Semester="F1997"/>
</Student>
```

---

17.7

However, since we indicated that the type of Details is XML, such a document will be stored not as a string, but in a special data structure, which supports efficient navigation and querying.

Using a CHECK constraint, we can even tell the DBMS to validate the above Student XML document against a suitable schema before allowing the document to be inserted:

---

```
CREATE TABLE STUDENTXML (
    Id      INTEGER,
    Details XML,
    CHECK(Details IS VALID INSTANCE OF 'http://xyz.edu/student.xsd'))
```

---

Here we have assumed that the schema is stored at the URL <http://xyz.edu/student.xsd>.

**The XMLELEMENT and XMLATTRIBUTES functions.** Since SQL is a relational language, an SQL query does not produce an XML document directly. Instead, a query result can contain tuples in which the value in a particular column is an XML document. We already know that XML documents can be stored as values of an attribute. What is new here is the ability to construct such documents on the fly from data stored in tables. The simplest way to do this is to use the XMLELEMENT function, which takes as parameters the name to be given to the element's tag and (optionally) the element's attributes and content. For instance, the following query produces a relation with a column that contains XML documents:

---

```
SELECT P.Id, XMLELEMENT(
    Name "Prof",
    XMLATTRIBUTES(P.DeptId AS "Dept"), -- The attribute(s)
    P.Name
) AS Info
-- The content
FROM PROFESSOR P
```

---

The parameter that provides the element's tag is identified by the keyword `Name`, which in this case is `Prof`. The `XMLATTRIBUTES` function produces the element's attributes. In this case the element has the single attribute `Dept`. The remaining parameters specify the element's content. In this case the element has no *e*-children and the content is simply the value of `P.Name` associated with `P.Id`. Using the `PROFESSOR` relation depicted in Figure 3.5, page 39, the query would produce the following tuples:

---

```
{ 101202303, <Prof Dept="CS">John Smyth</Prof> }
{ 783432188, <Prof Dept="MGT">Adrian Jones</Prof> }
:
:
```

---

The query maps each row of `PROFESSOR` into a row of the query result. The second component in each row is an XML element, and the column name is `Info`.

The `XMLELEMENT` constructs can be nested. This feature can be used to create arbitrarily complex XML elements. For instance, we could publish the `PROFESSOR` relation in the standard form defined by SQL/XML (see the document (17.5)) as follows:

---

```
SELECT XMLELEMENT(Name "Professor", -- The tag name
    XMLELEMENT(Name "Id", P.Id), -- Child elements
    XMLELEMENT(Name "Name", P.Name),
```

```

        XMLEMENT(Name "DeptId", P.DeptId)
    ) AS ProfElement
FROM PROFESSOR P

```

The result of this query is not a set of XML elements, but a table containing a single column with name ProfElement. Each row contains an XML element. The absence of an XMLATTRIBUTES parameter indicates that the element has no attributes. The remaining parameters of the outer XMLEMENT define the content. In this case each element has exactly three child elements.

**The XMLGEN function.** XMLGEN provides similar functionality to XMLEMENT in that it produces a table whose rows contain XML documents. It has, however, a more convenient format and is thus simpler to use. Its first argument is an XML template, which can contain *placeholder variables* of the form {\$foo}. The remaining arguments to XMLGEN are named expressions whose values are substituted for the placeholders with the same name. For instance, the above query can be rewritten with the help of XMLGEN as follows:

---

```

SELECT XMLGEN( '<Professor>
    <Id>{$I}</Id><Name>{$N}</Name>
    <DeptId>{$D}</DeptId>
    </Professor>',
    P.Id AS I,
    P.Name AS N,
    P.DeptId AS D,
    ) AS ProfElement
FROM PROFESSOR P

```

---

Placeholder variables can occur in the position of XML elements and attributes, which provides great flexibility in the way XML documents can be constructed from relational data.

The expressions to be substituted for the placeholders in XMLGEN are not limited to SQL variables, such as P.Id in the above example—they can be XML-generating expressions or even SELECT statements. For instance, we could rewrite the above query in the following way, where the expression to be substituted for \$I generates an XML element of the form <Id>...</Id>:

---

```

SELECT XMLGEN( '<Professor>
    {$I}<Name>{$N}</Name><DeptId>{$D}</DeptId>
    </Professor>',
    XMLEMENT(Name "Id", P.Id) AS I,
    P.Name AS N,
    P.DeptId AS D,
    ) AS ProfElement
FROM PROFESSOR P

```

---

**Grouping and XMLEGG.** In SQL/XML, SELECT statements can be embedded inside XML constructors (such as XMLELEMENT) that appear in the SELECT clause of a parent query. As a result, it is possible to group elements as children of another element. The following example illustrates this facility with a query that returns student transcripts grouped inside Student elements, as shown in (17.7). For brevity, we omit student name and status from the output.

---

```
SELECT XMLELEMENT( Name "Student",
                    XMLATTRIBUTES(S.Id AS "Id"),
                    ( SELECT
                        XMLELEMENT( Name "CrsTaken",
                                    XMLATTRIBUTES(T.CrsCode AS "CrsCode",
                                                T.Semester AS "Semester"))
                        FROM TRANSCRIPT T
                        WHERE S.Id = T.StudId ) )
FROM STUDENT S
```

---

In this example, we assume that information is stored inside the relations STUDENT and TRANSCRIPT, as in Figures 3.2 and 3.5 on pages 36 and 39. The statement produces a table with a single column and a row corresponding to each row of STUDENT. The content of each row is an XML element describing a particular student. The element is produced by the outer XMLELEMENT function with tag Student and attribute Id. The nested SELECT clause creates the content of the Student element, which in this case is a list of CrsTaken child elements corresponding to that student. Each of these e-children has two attributes defined by the XMLATTRIBUTES function, but no other content. The result of this query will thus be a set of XML elements that look like Student elements at the top of Figure 17.4 on page 578 (with student name and status information omitted).

Note that, strictly speaking, the nested SELECT clause produces not a list of CrsTaken elements (despite what we said earlier), but a set of 1-tuples, each containing a CrsTaken element. At the time of this writing it is unclear whether such a set of tuples will be converted into a list of XML elements automatically or if a special function will be provided for this purpose.

Alternatively, we could express the same query using aggregation with the help of the XMLEGG function of SQL/XML. To understand how it works, consider the following example, which reformulates the previous query using XMLEGG.

---

```
SELECT XMLELEMENT( Name "Student",
                    XMLATTRIBUTES(S.Id AS "Id"),
                    XMLEGG(
                        XMLELEMENT( Name "CrsTaken",
                                    XMLATTRIBUTES(T.CrsCode AS "CrsCode",
                                                T.Semester AS "Semester"))
                        ORDER BY T.CrsCode ) )
```

```
FROM STUDENT S, TRANSCRIPT T
WHERE S.Id = T.StudId
GROUP BY S.Id
```

**XMLELAGG** provides the content of a **Student** element created by the outer **XMLELEMENT** function. Each **Student** element corresponds to a particular group produced by the **GROUP BY** clause, and each row in that group (selected from the set of tuples in the join of **STUDENT** and **TRANSCRIPT**, which have the same student Id) describes a course taken by a particular student. **XMLELAGG** refers to that group through its use of the tuple variable **T**.

**XMLELAGG** takes as an argument an XML construct, such as the nested **XMLELEMENT** invocation in the example, in which an SQL variable is a parameter—**T** in this case. The invocation of **XMLELAGG** produces a *list* of elements—one for each legal value of the variable. Here the legal values are the rows of **TRANSCRIPT** that are joined with the particular row of **STUDENT** used to form the group. The list is nested within the **Student** element and is ordered by course codes using the optional **ORDER BY** clause of **XMLELAGG**.

For instance, if a student, Bart Simpson, with Id 987654321 took CS305 in the fall of 1995 and MGT123 in the fall of 1994, then when **S.Id** is bound to 987654321, the **GROUP BY** clause will produce a group of two bindings for the variables **S** and **T**: one where **S = (987654321, Bart Simpson, ...)** and **T = (987654321, CS305, F1995, C)**, and another where **S = (987654321, Bart Simpson, ...)** and **T = (987654321, MGT123, F1994, B)**. For each binding of **T**, the **XMLELAGG** operator will produce one **CrsTaken** element. Thus, for this particular group of tuples the above query will construct the following element:

```
<Student Id="987654321">
    <CrsTaken CrsCode="CS305" Semester="F1995"/>
    <CrsTaken CrsCode="MGT123" Semester="F1994"/>
</Student>
```

You may find the reference to the word “aggregate” in the function name **XMLELAGG** confusing. The aggregate here is the list—think of the above list of **CrsTaken** elements as a “sum” of the courses the student has taken.

### Querying XML Documents Stored inside Relations

Once XML documents are stored as instances of the new XML data type (see (17.6) and (17.7)), it becomes necessary to provide a mechanism to query them in order to extract their contents. This is done using a pair of new functions, **XMLEXTRACT** and **XMLEXISTS**, which are discussed below.

**The XMLEXTRACT function.** The **XMLEXTRACT** function applies an XPath expression to XML documents stored as XML data type. To illustrate this, we use the relation **STUDENTXML** defined in (17.6), which stores documents of the form (17.7). The

query returns the names and IDs of all students who have status U3 and who have taken MAT123.

---

```
SELECT S.Id, XMLEXTRACT(S.Details, '//Name')
FROM STUDENTXML S
WHERE XMLEXTRACT(S.Details, '//Status/text()') = 'U3' AND
XMLEXTRACT(S.Details, '//CrsTaken/@CrsCode') = 'MAT123'
```

---

The first parameter of `XMLEXTRACT` names an attribute of type `XML`, which contains a document. The second parameter is an XPath expression. `XMLEXTRACT` returns the result of an application of the expression to the document.

In this query, application of the path expression, `//Name`, to any document in `S.Details` yields a single `Name` element, so the result of the query is a set of regular tuples (modulo the fact that the second tuple component is an `XML` element). In general, however, a path expression can yield a set. Will the result still be legal? The answer is yes—in SQL:2003, which introduced the `MULTISET` type to allow the components of a tuple to be sets. The `MULTISET` data type is described in Section 16.3.8.

**The predicate `XMLEXISTS`.** The function `XMLEXTRACT` makes it possible to query XML documents that are stored inside SQL databases. When the set of nodes returned by the associated XPath expression is empty, `XMLEXTRACT` returns an empty document. Sometimes, however, it is necessary to just test if this set is empty and act based on the result. In SQL/XML, this purpose is served by the predicate `XMLEXISTS`. The following example, which lists all students who have at least one course in their transcript, illustrates the use of this feature.

---

```
SELECT S.Id, XMLEXTRACT(S.Details, '//Name')
FROM STUDENTXML S
WHERE XMLEXISTS(S.Details, '//CrsTaken')
```

---

17.8

### Modifying Data in SQL/XML

So far we have been focusing on schema and query-related issues pertaining to XML data stored in SQL databases. In this section we briefly review the support for updating such data. The material in this section is somewhat more tentative than the other parts of SQL/XML, and the details are more likely to change.

**Functions `XMLPARSE` and `XMLVALIDATE`.** We have seen some queries involving `STUDENTXML` (17.6), a relation that stores XML documents of the form (17.7) in the `Details` column. But how do you put XML documents into such a table in the first place?

Since an XML document can be viewed as a string of characters, we could, in principle, store it as a string data type. However, this misses an important point

behind SQL/XML—the XML data type. XML documents are stored in columns of type XML, not as strings but rather using special tree structures. To produce such a structure, SQL/XML provides a special function, XMLPARSE, that converts an XML document represented as a string into the tree structures appropriate for the XML data type prior to storing it. This makes it possible to insert tuples into a relation that contains XML documents, as shown in the following example.

---

```
INSERT INTO STUDENTXML(Id, Details)
VALUES ( 123987456,
        XMLPARSE(
          '<Student>
            <Name><First>John</First><Last>Doe</Last></Name>
            <Status>U2</Status>
            <CrsTaken CrsCode="CS310" Semester="F2003"/>
            <CrsTaken CrsCode="CS305" Semester="F2003"/>
          </Student>' )
        )
```

---

The XMLPARSE function parses documents (to put them into the format appropriate to the XML data type) and checks for correctness, but it is not supposed to validate them—this is reserved for the XMLVALIDATE function. At present there is only a preliminary proposal, which indicates that the function should work as in the following modification of the above example:

---

```
INSERT INTO STUDENTXML(Id, Details)
VALUES ( 123987456,
        XMLVALIDATE(XMLPARSE(
          '<Student>
            <Name><First>John</First><Last>Doe</Last></Name>
            <Status>U2</Status>
            <CrsTaken CrsCode="CS310" Semester="F2003"/>
            <CrsTaken CrsCode="CS305" Semester="F2003"/>
          </Student>' ) )
```

---

The result is the same as in the previous case, except that the XML document will be stored in the database only if it is validated against an appropriate schema document. If the XML document includes the mention of the corresponding schema document, the above scenario would be adequate. This is not the case, however, in our example. In the future, presumably there will be an option to provide the location of an XML Schema document to validate against.

It is also expected that a future release of SQL/XML will include primitives for direct modification of documents stored using the XML data type.

**The function XMLSERIALIZE.** This function is less relevant to update operations, but it is appropriate to mention it here since it is the reverse of XMLPARSE. It takes a document of the XML data type and returns a string representation of that document.

One scenario in which this might be useful is when you want (for some reason) to store a copy of an XML document as a string. A more common case, however, is when SQL is embedded in a host language, such as C or C++, that does not understand XML. In this case, the embedded SQL query might need to convert the XML output into a string before the host program can deal with it. In the following example, we declare a cursor for query (17.8), which previously returned tuples with native XML documents in them. The only difference now is the use of the XMLSERIALIZE function.

---

```
EXEC SQL DECLARE GETENROLLED CURSOR FOR
  SELECT S.Id, XMLSERIALIZE(XMLEXTRACT(S.Details, '//Name'))
  FROM STUDENTXML S
  WHERE XMLEXISTS(S.Details, '//CrsTaken')
```

---

Since the XML documents, which are returned by the XPath expression '//Name', are now converted to strings, we can process each tuple in the result one by one using the following statement:

---

```
EXEC SQL FETCH GETENROLLED INTO :studId, :details;
```

---

## BIBLIOGRAPHIC NOTES

The semistructured data model had an important influence on a number of developments in the XML arena, especially on XML query languages. A more in-depth study of semistructured data can be found in [Abiteboul et al. 2000].

XML came as a result of an effort to bring some order to Web information processing. Conceptually, it is a rewrite and a simplification of the well-established SGML standard [SGML 1986]. Version 1 was approved in 1998 and became a widely accepted standard [XML 1998]. As with every new hot topic, many publications appeared in a short period of time. There are too many to list here, so we mention just two recent ones, [Ray 2001; Bradley 2000a].

XML Schema is covered in a number of books, but, because it was a moving target until recently, we recommend the authoritative sources [XMLSchema 2000a; XMLSchema 2000b].

XPath, the XML path expression language, is described in most recent publications on XML. The official W3C recommendation can be found in [XPath 1999]. The original idea of path expressions comes from [Zaniolo 1983]. The idea of enhancing path expressions with query capability was developed by [Kifer and Lausen 1989; Kifer et al. 1992; Frohn et al. 1994; Abiteboul et al. 1997; Deutsch et al. 1998] and others.

SQL/XML is a standard in the making, which will eventually be incorporated into SQL:2003. A number of relational vendors already support parts of this specification. Details on the SQL/XML standardization activity and the current document drafts can be found at <http://www.sqlx.org/>.

Two other important XML query languages, which have not been discussed here but are found in the full version of this book [Kifer et al. 2004], are XSLT [XSLT 1999] and XQuery [XQuery 2004].

## EXERCISES

- 17.1 Use XML to represent the contents of the STUDENT relation in Figure 3.2, page 36. Specify a DTD appropriate for this document. Do *not* use the representation proposed by the SQL/XML specification discussed in Section 17.4.2.
- 17.2 Specify a DTD appropriate for a document that contains data from both the COURSE table in Figure 4.34, page 116, and the REQUIRES table in Figure 4.35, page 117. Try to reflect as many constraints as the DTDs allow. Give an example of a document that conforms to your DTD.
- 17.3 Restructure the document in Figure 17.4, page 578, so as to completely replace the elements **Name**, **Status**, **CrsCode**, **Semester**, and **CrsName** with attributes in the appropriate tags. Provide a DTD suitable for this document. Specify all applicable ID and IDREF constraints.
- 17.4 Define the following simple types:
  - a. A type whose domain consists of lists of strings, where each list consists of 7 elements
  - b. A type whose domain consists of lists of strings, where each string is of length 7
  - c. A type whose domain is a set of lists of strings, where each string has between 7 and 10 characters and each list has between 7 and 10 elements
  - d. A type appropriate for the letter grades that students receive on completion of a course—A, A–, B+, B, B–, C+, C, C–, D, and F. Express this type in two different ways: as an enumeration and using the **pattern** tag of XML Schema.
- 17.5 Use the **key** statement of XML Schema to define the following key constraints for the document in Figure 17.4:
  - a. The key for the collection of all **Student** elements
  - b. The key for the collection of all **Course** elements
  - c. The key for the collection of all **Class** elements
- 17.6 Assume that any student in the document of Figure 17.4 on page 578 is uniquely identified by the last name and the status. Define this key constraint.
- 17.7 Use the **keyref** statement of XML Schema to define the following referential integrity for the document in Figure 17.4:
  - a. Every course code in a **CourseTaken** element must refer to a valid course.
  - b. Every course code in a **Class** element must refer to a valid course.
- 17.8 Express the following constraint on the document of Figure 17.4: no pair of **CourseTaken** elements within the same **Student** element can have identical values of the **CrsCode** attribute.

- 17.9 Rearrange the structure of the **Class** element in Figure 17.4 so that it becomes possible to define the following referential integrity: every student Id mentioned in a **Class** element references a student from the same document.
- 17.10 Write a unified XML schema that covers both documents in Figures 17.15 and 17.16. Provide the appropriate key and foreign-key constraints.
- 17.11 Use XML Schema to represent the fragment of the relational schema in Figure 3.6, page 43. Include all key and foreign-key constraints.
- 17.12 Use XPath to express the following queries to the document in Figure 17.15:
- Find all **Student** elements whose Ids end with 987 and who have taken **MAT123**.
  - Find all **Student** elements whose first names are **Joe** and who have taken fewer than three courses.
  - Find all **CrsTaken** elements that correspond to semester **S1996** and that belong to students whose names begin with **P**.

FIGURE 17.15 Transcripts at <http://xyz.edu/transcripts.xml>.

```
<?xml version="1.0" ?>
<Transcripts>
    <Transcript>
        <Student StudId="s111111111" Name="John Doe"/>
        <CrsTaken CrsCode="CS308" Semester="F1997" Grade="B"/>
        <CrsTaken CrsCode="MAT123" Semester="F1997" Grade="B"/>
        <CrsTaken CrsCode="EE101" Semester="F1997" Grade="A"/>
        <CrsTaken CrsCode="CS305" Semester="F1995" Grade="A"/>
    </Transcript>
    <Transcript>
        <Student StudId="s987654321" Name="Bart Simpson"/>
        <CrsTaken CrsCode="CS305" Semester="F1995" Grade="C"/>
        <CrsTaken CrsCode="CS308" Semester="F1994" Grade="B"/>
    </Transcript>
    <Transcript>
        <Student StudId="s123454321" Name="Joe Blow"/>
        <CrsTaken CrsCode="CS315" Semester="S1997" Grade="A"/>
        <CrsTaken CrsCode="CS305" Semester="S1996" Grade="A"/>
        <CrsTaken CrsCode="MAT123" Semester="S1996" Grade="C"/>
    </Transcript>
    <Transcript>
        <Student StudId="s023456789" Name="Homer Simpson"/>
        <CrsTaken CrsCode="EE101" Semester="F1995" Grade="B"/>
        <CrsTaken CrsCode="CS305" Semester="S1996" Grade="A"/>
    </Transcript>
</Transcripts>
```

**FIGURE 17.16** Classes at <http://xyz.edu/classes.xml>.

```

<?xml version="1.0" ?>
<Classes>
    <Class CrsCode="CS308" Semester="F1997">
        <CrsName>Software Engineering</CrsName>
        <Instructor>Adrian Jones</Instructor>
    </Class>
    <Class CrsCode="EE101" Semester="F1995">
        <CrsName>Electronic Circuits</CrsName>
        <Instructor>David Jones</Instructor>
    </Class>
    <Class CrsCode="CS305" Semester="F1995">
        <CrsName>Database Systems</CrsName>
        <Instructor>Mary Doe</Instructor>
    </Class>
    <Class CrsCode="CS315" Semester="S1997">
        <CrsName>Transaction Processing</CrsName>
        <Instructor>John Smyth</Instructor>
    </Class>
    <Class CrsCode="MAT123" Semester="F1997">
        <CrsName>Algebra</CrsName>
        <Instructor>Ann White</Instructor>
    </Class>
</Classes>
```

---

**17.13** Formulate the following XPath queries for the document in Figure 17.16:

- Find the names of all courses taught by Mary Doe in fall 1995.
- Find the set of all document nodes that correspond to the course names taught in fall 1996 or all instructors who taught MAT123.
- Find the set of all course codes taught by John Smyth in spring 1997.

**17.14** Write an XML Schema specification for a simple document that lists stockbrokers with the accounts that they handle and lists client accounts separately. The information about the accounts includes the account Id, ownership information, and the account positions (i.e., stocks held in that account). To simplify matters, it suffices to list the stock symbol and quantity for each account position. Use ID, IDREF, and IDREFS to specify referential integrity.

**17.15** Write a sample XML document, which contains

- A list of parts (part name and Id)
- A list of suppliers (supplier name and Id)
- A list of projects; for each project element, a nested list of subelements that represent the parts used in that project. Include the information on who supplies that part and in what quantity.

Write a DTD for this document and an XML Schema. Express all key and referential constraints.

Choose your representation in such a way as to maximize the number of possible key and referential constraints representable using DTDs.

- 17.16 Consider the relational schema in Figure 3.6, page 43. Assume that the contents of these relations are stored in a single XML column of a relation using the following XML format: the name of the relation is the top-level element, each tuple is represented as a tuple element, and each relation attribute is represented as an empty element that has a value attribute. For instance, the STUDENT relation would be represented as follows:

---

```
<Student>
  <tuple>
    <Id value="s1111111111"/> <Name value="John Doe"/>
    <Address value="123 Main St."/> <Status value="U1"/>
  </tuple>
  :
</Student>
```

---

Formulate the following queries in SQL/XML:

- (a) Create the list of all professors who ever taught MAT123. The information must include all attributes available from the PROFESSOR relation.  
(b) Create the list of all courses in which Joe Public received an A.  
(c) Create the list of all students (include student Id and name) who have taken a course from John Smyth and received an A.

- 17.17 Consider an SQL/XML database schema that consists of two relations:

- The SUPPLIER relation has the following attributes:
  - Id, an integer
  - Name, a string
  - Address, an XML type appropriate for addresses
  - Parts, an XML type that represents the parts supplied by the supplier. Each part has Id, Name, and Price.
- The PROJECT relation has the following attributes:
  - Project Name, a string
  - Project Members, an appropriate XML type
  - Project Parts, an XML type. This attribute represents the list of parts used by the project and supplied by a supplier. Each part has an Id, a Name, and a SupplierId.

Use SQL/XML to define the database schema. Make sure that a CHECK constraint validates the XML documents inserted into the database using appropriate XML Schema document. Then answer the following queries:

- (a) Find all projects that are supplied by Acme Inc. and that have Joe Public as a member.

- (b) Find all projects that are *not* supplied by Acme Inc., that is, none of the parts used by the project comes from Acme Inc.
- (c) Find all project members who participate in every project.
- (d) Find the projects with the highest number of members.
- 17.18** Consider the database depicted in Figure 3.5 on page 39. Use SQL/XML to answer the following queries:
- (a) Based on the relation TRANSCRIPT, output the same information in a different format. The output should have two attributes: StudId and Courses, where Courses should have XML type similar to the one used throughout this chapter for the CourseTaken element (e.g., as in Figure 17.4).
- (b) Repeat the previous query, but use TEACHING instead. The output should have the attributes ProfId and Courses. The latter should have XML type and should describe the courses that the professor with the given ProfId has even taught.
- (c) Produce a list of professors (Id and Name) along with the list of courses that professor teaches in spring 2004. The list of courses should have the XML type and should include CrsCode and DeptId as attributes of an element and CrsName as a text node.
- (d) For each course, produce a list of the professors who have ever taught it. The professor list should have the XML type. Choose your own schema.
- (e) Repeat the previous query, but output only the courses that have the greatest number of professors who have taught it.

# Bibliography

---

- Abiteboul, S., Buneman, P., and Suciu, D. (2000). *Data on the Web*. Morgan Kaufmann, San Francisco.
- Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley, Boston, MA.
- Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. (1997). The Lorel query language for semistructured data. *International Journal on Digital Libraries* 1(1): 68–88.
- Arisawa, H., Moriya, K., and Miura, T. (1983). Operations and the properties of non-first-normal-form relational databases. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Florence, 197–204.
- Armstrong, W. (1974). Dependency structures of database relations. *IFIP Congress*, Stockholm, 580–583.
- Astrahan, M., Blasgen, M., Gray, J., King, W., Lindsay, B., Lorie, R., Mehl, J., Price, T., Selinger, P., Schkolnick, M., Traiger, D. S. I., and Yost, R. (1981). A history and evaluation of System R. *Communications of the ACM* 24(10): 632–646.
- Atzeni, P., and Antonellis, V. D. (1993). *Relational Database Theory*. Benjamin-Cummings, San Francisco.
- Bancilhon, F., Delobel, C., and Kanellakis, P. (eds.) (1990). *Building an Object-Oriented Database System: The Story of O2*. Morgan Kaufmann, San Francisco.
- Bancilhon, F., and Spyros, N. (1981). Update semantics of relational views. *ACM Transactions on Database Systems* 6(4): 557–575.
- Batini, C., Ceri, S., and Navathe, S. (1992). *Database Design: An Entity Relationship Approach*. Benjamin-Cummings, San Francisco.
- Bayer, R., and McCreight, E. (1972). Organization and maintenance of large ordered indices. *Acta Informatica* 1(3): 173–189.
- Beeri, C., and Bernstein, P. (1979). Computational problems related to the design of normal form relational schemes. *ACM Transactions on Database Systems* 4(1): 30–59.
- Beeri, C., Bernstein, P., and Goodman, N. (1978). A sophisticate's introduction to database normalization theory. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, San Mateo, CA, 113–124.
- Beeri, C., Fagin, R., and Howard, J. (1977). A complete axiomatization for functional and multivalued dependencies in database relations. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Toronto, Canada, 47–61.

- Beeri, C., and Kifer, M. (1986a). Elimination of intersection anomalies from database schemes. *Journal of the ACM* 33(3): 423–450.
- Beeri, C., and Kifer, M. (1986b). An integrated approach to logical design of relational database schemes. *ACM Transactions on Database Systems* 11(2): 134–158.
- Beeri, C., and Kifer, M. (1987). A theory of intersection anomalies in relational database schemes. *Journal of the ACM* 34(3): 544–577.
- Beeri, C., Mendelson, A., Sagiv, Y., and Ullman, J. (1981). Equivalence of relational database schemes. *SIAM Journal of Computing* 10(2): 352–370.
- Bernstein, P. (1976). Synthesizing third normal form from functional dependencies. *ACM Transactions on Database Systems* 1(4): 277–298.
- Bernstein, P., and Newcomer, E. (1997). *Principles of Transaction Processing*. Morgan Kaufmann, San Francisco.
- Biskup, J., Menzel, R., and Polle, T. (1996). Transforming an entity-relationship schema into object-oriented database schemas. In J. Eder and L. Kalinichenko (eds.), *Advances in Databases and Information Systems*, Workshops in Computing. Springer-Verlag, Moscow, Russia, 109–136.
- Biskup, J., Menzel, R., Polle, T., and Sagiv, Y. (1996). Decomposition of relationships through pivoting. *Proceedings of the 15th International Conference on Conceptual Modeling*. In Vol. 1157 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, 28–41.
- Biskup, J., and Polle, T. (2000a). *Constraints in Object-Oriented Databases* (manuscript).
- Biskup, J., and Polle, T. (2000b). Decomposition of database classes under path functional dependencies and onto constraints. *Proceedings of the Foundations of Information and Knowledge-Based Systems*. In Vol. 1762 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, 31–49.
- Blaha, M., and Premerlani, W. (1998). *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, Englewood Cliffs, NJ.
- Blakeley, J., and Martin, N. (1990). Join index, materialized view, and hybrid-hash join: A performance analysis. *Proceedings of the International Conference on Data Engineering (ICDE)*, Los Angeles, 256–263.
- Blasgen, M., and Eswaran, K. (1977). Storage access in relational databases. *IBM Systems Journal* 16(4): 363–378.
- Booch, G. (1994). *Object-oriented Analysis and Design with Applications*. Addison-Wesley, Boston, MA.
- Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley, Boston, MA.
- Bourret, R. (2000). Namespace myths exploded. <http://www.xml.com/pub/a/2000/03/08/namespaces/index.html>.
- Bradley, N. (2000a). *The XML Companion*. Addison-Wesley, Boston, MA.
- Bray, T., Hollander, D., and Layman A. (1999). <http://www.w3.org/TR/1999/REC-xml-names-19990114/>.

- Buneman, P., Davidson, S., Hillebrand, G., and Suciu, D. (1996). A query language and optimization techniques for unstructured data. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, 505–516.
- Cattell, R., and Barry, D. (eds.) (2000). *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, San Francisco.
- Ceri, S., Negri, M., and Pelagatti, G. (1982). Horizontal partitioning in database design. *Proceedings of the International ACM SIGMOD Conference on Management of Data*, Orlando, FL, 128–136.
- Ceri, S., and Pelagatti, G. (1984). *Distributed Databases: Principles and Systems*. McGraw-Hill, New York.
- Chaudhuri, S. (1998). An overview of query optimization in relational databases. *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Seattle, 34–43.
- Chaudhuri, S., Krishnamurthy, R., Potamianos, S., and Shim, K. (1995). Optimizing queries with materialized views. *Proceedings of the International Conference on Data Engineering (ICDE)*, Taipei, Taiwan, 190–200.
- Chen, I.-M., Hull, R., and McLeod, D. (1995). An execution model for limited ambiguity rules and its application to derived data update. *ACM Transactions on Database Systems* 20(4): 365–413.
- Chen, P. (1976). The Entity-Relationship Model—Towards a unified view of data. *ACM Transactions on Database Systems* 1(1): 9–36.
- Cochrane, R., Pirahesh, H., and Mattos, N. (1996). Integrating triggers and declarative constraints in SQL database systems. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Bombay, India, 567–578.
- Codd, E. (1970). A relational model of data for large shared data banks. *Communications of the ACM* 13(6): 377–387.
- Codd, E. (1972). Relational completeness of data base sublanguages. *Data Base Systems*. In Vol. 6 of *Courant Computer Science Symposia Series*. Prentice Hall, Englewood Cliffs, NJ.
- Codd, E. (1979). Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems* 4(4): 397–434.
- Codd, E. (1990). *The Relational Model for Database Management, Version 2*. Addison-Wesley, Boston, MA.
- Copeland, G., and Maier, D. (1984). Making Smalltalk a database system. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Boston, 316–325.
- Cosmadakis, S., and Papadimitriou, C. (1983). Updates of relational views. *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Atlanta, 317–331.
- Date, C., and Darwen, H. (1997). *A Guide to the SQL Standard*. (4th ed.). Addison-Wesley, Boston, MA.

- Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. (1998). XML-QL: A query language for XML. *Technical Report W3C*. <http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/>.
- Deutsch, A., Fernández, M., and Suciu, D. (1999). Storing semistructured data with stored. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Philadelphia, 431–442.
- DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., and Wood, D. (1984). Implementation techniques for main-memory database systems. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Boston, 1–8.
- DOM (2000). Document Object Model (DOM). <http://www.w3.org/DOM/>.
- Eisenberg, A. (1996). New standard for stored procedures in SQL. *SIGMOD Record* 25(4): 81–88.
- Eswaran, K., Gray, J., Lorie, R., and Traiger, I. (1976). The notions of consistency and predicate locks in a database system. *Communications of the ACM* 19(11): 624–633.
- Fagin, R. (1977). Multivalued dependencies and a new normal form for relational databases. *ACM Transactions on Database Systems* 2(3): 262–278.
- Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. (1979). Extendible hashing—A fast access method for dynamic files. *ACM Transactions on Database Systems* 4(3): 315–344.
- Flach, P. A., and Savnik, I. (1999). Database dependency discovery: A machine learning approach. *AI Communications* 12(3): 139–160.
- Fowler, M., and Scott, K. (2003). *UML Distilled*, 3rd ed. Addison-Wesley, Boston, MA.
- Frohn, J., Lausen, G., and Uphoff, H. (1994). Access to objects by path expressions and rules. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Santiago, Chile, 273–284.
- Garcia-Molina, H., Ullman, J., and Widom, J. (2000). *Database System Implementation*, Prentice Hall, Englewood Cliffs, NJ.
- Gogola, M., Herzig, R., Conrad, S., Denker, G., and Vlachantonis, N. (1993). Integrating the E-R approach in an object-oriented environment. *Proceedings of the 12th International Conference on the Entity-Relationship Approach*, Arlington, TX, 376–389.
- Gottlob, G., Paolini, P., and Zicari, R. (1988). Properties and update semantics of consistent views. *ACM Transactions on Database Systems* 13(4): 486–524.
- Graefe, G. (1993). Query evaluation techniques for large databases. *ACM Computing Surveys* 25(2): 73–170.
- Gray, J. (1978). Notes on database operating systems. *Operating Systems: An Advanced Course*. In Vol. 60 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 393–481.
- Gray, J., Laurie, R., Putzolu, G., and Traiger, I. (1976). Granularity of locks and degrees of consistency in a shared database. *Modeling in Data Base Management Systems*, Elsevier, North Holland.
- Gray, J., and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco.

- Griffiths-Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., and Price, T. (1979). Access path selection in a relational database system. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Boston, 23–34.
- Gulutzan, P., and Pelzer, T. (1999). *SQL-99 Complete, Really*. R&D Books, Gilroy, CA.
- Gupta, A., and Mumick, I. (1995). Maintenance of materialized views: Problems, techniques, and applications. *Data Engineering Bulletin* 18(2): 3–18.
- Gupta, A., Mumick, I., and Ross, K. (1995). Adapting materialized views after redefinitions. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, CA, 211–222.
- Gupta, A., Mumick, I., and Subrahmanian, V. (1993). Maintaining views incrementally. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, DC, 157–166.
- Haerder, T., and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys* 15(4): 287–317.
- Harrison, G. (2001) *Oracle SQL: High-Performance Tuning* (2nd ed.). Prentice Hall, Upper Saddle River, New Jersey
- Huhtala, Y., Karkkainen, J., Porkka, P., and Toivonen, H. (1999). TANE: An efficient algorithm for discovery of functional and approximate dependencies. *The Computer Journal* 42(2): 100–111.
- Ioannidis, Y. (1996). Query optimization. *ACM Computing Surveys* 28(1): 121–123.
- Ito, M., and Weddell, G. (1994). Implication problems for functional constraints on databases supporting complex objects. *Journal of Computer and System Sciences* 49(3): 726–768.
- Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Boston, MA.
- Jaeschke, G., and Schek, H.-J. (1982). Remarks on the algebra of non-first-normal-form-relations. *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Los Angeles, 124–138.
- Kanellakis, P. (1990). Elements of relational database theory. In J. V. Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. B, *Formal Models and Semantics*. Elsevier, Amsterdam, 1073–1156.
- Kantola, M., Mannila, H., Raäihä, K.-J., and Siirtola, H. (1992). Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems* 7(7): 591–607.
- Keller, A. (1985). Algorithms for translating view updates to database updates for views involving selections, projections, and joins. *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Portland, OR, 154–163.
- Kifer, M., Kim, W., and Sagiv, Y. (1992). Querying object-oriented databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, DC, 393–402.
- Kifer, M., Bernstein, A. J., and Lewis, P. M. (2004). *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley, Boston, MA.

- Kifer, M., and Lausen, G. (1989). F-Logic: A higher-order language for reasoning about objects, inheritance and schema. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Portland, OR, 134–146.
- Kifer, M., Lausen, G., and Wu, J. (1995). Logical foundations of object-oriented and frame-based languages. *Journal of the ACM* 42(4): 741–843.
- Kitsuregawa, M., Tanaka, H., and Moto-oka, T. (1983). Application of hash to database machine and its architecture. *New Generation Computing* 1(1): 66–74.
- Knuth, D. (1973). *The Art of Computer Programming: Vol III, Sorting and Searching*, (1st ed.), Addison-Wesley, Boston, MA.
- Knuth, D. (1998). *The Art of Computer Programming: Vol III, Sorting and Searching*, (3rd ed.), Addison-Wesley, Boston, MA.
- Lampson, B., Paul, M., and Seigert, H. (1981). *Distributed Systems: Architecture and Implementation (An Advanced Course)*. Springer-Verlag, Heidelberg, Germany.
- Lampson, B., and Sturgis, H. (1979). Crash recovery in a distributed data storage system. *Technical Report*. Xerox Palo Alto Research Center, Palo Alto, CA.
- Langerak, R. (1990). View updates in relational databases with an independent scheme. *ACM Transactions on Database Systems* 15(1): 40–66.
- Larson, P. (1981). Analysis of index sequential files with overflow chaining. *ACM Transactions on Database Systems* 6(4): 671–680.
- Litwin, W. (1980). Linear hashing: A new tool for file and table addressing. *Proceedings of the International Conference on Very Large Databases (VLDB)*, Montreal, Canada, 212–223.
- Maier, D. (1983). *The Theory of Relational Databases*. Computer Science Press. Rockville, MD. (Available through Books on Demand: <http://www.umi.com/hp/Support/BOD/index.html>.)
- Makinouchi, A. (1977). A consideration on normal form of not-necessarily-normalized relations in the relational data model. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, 447–453.
- Mannila, H., and Raäihä, K.-J. (1992). *The Design of Relational Databases*. Addison-Wesley, Workingham, U.K.
- Mannila, H., and Raäihä, K.-J. (1994). Algorithms for inferring functional dependencies. *Knowledge Engineering* 12(1): 83–99.
- Maslak, B., Showalter, J., and Szczygielski, T. (1991). Coordinated resource recovery in VM/ESA. *IBM Systems Journal* 30(1): 72–89.
- Masunaga, Y. (1984). A relational database view update translation mechanism. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Singapore, 309–320.
- Melton, J. (1997). *Understanding SQL's Persistent Stored Modules*. Morgan Kaufmann, San Francisco.
- Melton, J., Eisenberg, A., and Cattell, R. (2000). *Understanding SQL and Java Together: A Guide to SQLJ, JDBC, and Related Technologies*. Morgan Kaufmann, San Francisco.

- Melton, J., and Simon, A. (1992). *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, San Francisco.
- Microsoft (1997). *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*. Microsoft Press, Seattle.
- Missaoui, R., Gagnon, J.-M., and Godin, R. (1995). Mapping an extended entity-relationship schema into a schema of complex objects. *Proceedings of the 14th International Conference on Object-Oriented and Entity Relationship Modeling*, Brisbane, Australia, 205–215.
- Mohania, M., Konomi, S., and Kambayashi, Y. (1997). Incremental maintenance of materialized views. *Database and Expert Systems Applications (DEXA)*. Springer-Verlag, Heidelberg, Germany.
- Mok, W., Ng, Y.-K., and Embley, D. (1996). A normal form for precisely characterizing redundancy in nested relations. *ACM Transactions on Database Systems* 21(1): 77–106.
- O'Neil, P. (1987). Model 204: Architecture and performance. *Proceedings of the International Workshop on High Performance Transaction Systems*. In Vol. 359 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, 40–59.
- O'Neil, P., and Graefe, G. (1995). Multi-table joins through bitmapped join indices. *SIGMOD Record* 24(3): 8–11.
- O'Neil, P., and Quass, D. (1997). Improved query performance with variant indexes. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, 38–49.
- Ozsoyoglu, Z., and Yuan, L.-Y. (1985). A normal form for nested relations. *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Portland, OR, 251–260.
- Paton, N., Diaz, O., Williams, M., Campin, J., Dinn, A., and Jaime, A. (1993). Dimensions of active behavior. *Proceedings of the Workshop on Rules in Database Systems*, Heidelberg, Germany, 40–57.
- Peterson, W. (1957). Addressing for random access storage. *IBM Journal of Research and Development* 1(2): 130–146.
- PostgreSQL (2000). PostgreSQL. <http://www.postgresql.org>.
- Pressman, R. (2002). *Software Engineering: A Practitioner's Approach*, 5th ed. McGraw-Hill, New York.
- Ram, S. (1995). Deriving functional dependencies from the entity-relationship model. *Communications of the ACM* 38(9): 95–107.
- Ray, E. (2001). *Learning XML*. O'Reilly and Associates, Sebastopol, CA.
- Reese, G. (2000). *Database Programming with JDBC and Java*. O'Reilly and Associates, Sebastopol, CA.
- Roth, M., and Korth, H. (1987). The design of non-1nf relational databases into nested normal form. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Francisco, 143–159.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenzen, W. (1991). *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ.

- Savnik, I., and Flach, P. (1993). Bottom-up induction of functional dependencies from relations. *Proceedings of the AAAI Knowledge Discovery in Databases Workshop (KDD)*, Ljubljana, Slovenia, 174–185.
- Schach, S. (1999). *Software Engineering*, 5th ed. Aksen Associates, Homewood, IL.
- Sciore, E. (1983). Improving database schemes by adding attributes. *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, New York, 379–383.
- SGML (1986). Information processing—text and office systems—Standard Generalized Markup Language (SGML). *ISO Standard 8879*. International Standards Organization, Geneva, Switzerland.
- Shasha, D., and Bonnet, P. (2003). *Database Tuning: Principles, Experiments and Troubleshooting Techniques*. Morgan Kaufman, San Francisco.
- Signore, R., Creamer, J., and Stegman, M. (1995). *The ODBC Solution: Open Database Connectivity in Distributed Environments*. McGraw-Hill, New York.
- Spaccapietra, S. (ed.) (1987). *Entity-Relationship Approach: Ten Years of Experience in Information Modeling, Proceedings of the Entity-Relationship Conference*, Elsevier, North Holland.
- SQL (1992). ANSI X3.135-1992, *American National Standard for Information Systems—Database Language—SQL*. American National Standards Institute, Washington, DC.
- SQLJ (2000). SQLJ. <http://www.sqlj.org>.
- Stallman, R. (2000). GNU coding standards. <http://www.gnu.org/prep/standards.html>.
- Standish (2000). Chaos. <http://standishgroup.com/visitor/chaos.htm>.
- Staudt, M., and Jarke, M. (1996). Incremental maintenance of externally materialized views. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Bombay, India, 75–86.
- Stonebraker, M. (1979). Concurrency control and consistency of multiple copies of data in INGRES. *IEEE Transactions on Software Engineering* 5(3): 188–194.
- Stonebraker, M. (1986). *The INGRES Papers: Anatomy of a Relational Database System*. Addison-Wesley, Boston, MA.
- Stonebreaker, M., and Kemnitz, G. (1991). The POSTGRES next generation database management system. *Communications of the ACM* 10(34): 78–92.
- Summerville, I. (2000). *Software Engineering*, 5th ed. Addison-Wesley, Boston, MA.
- Sun (2000). JDBC data access API. <http://java.sun.com/products/jdbc/>.
- Sybase (1999). Sybase Adaptive Server Enterprise Performance and Tuning Guide. <http://sybooks.sybase.com/onlinebooks/group-as/asg1200e/aseperf>.
- Teorey, T. (1999). *Database Modeling and Design: The E-R Approach*. Morgan Kaufmann, San Francisco.
- Thalheim, B. (1992). *Fundamentals of Entity-Relationship Modeling*. Springer-Verlag, Berlin.
- Ullman, J. (1988). *Principles of Database and Knowledge-Base Systems*, Volumes 1 and 2. Computer Science Press, Rockville, MD.
- Valduriez, P. (1987). Join indices. *ACM Transactions on Database Systems* 12(2): 218–246.

- Venkatrao, M., and Pizzo, M. (1995). SQL/CLI—A new binding style for SQL. *SIGMOD Record* 24(4): 72–77.
- Vincent, M. (1999). Semantic foundations of 4nf in relational database design. *Acta Informatica* 36(3): 173–213.
- Vincent, M., and Srinivasan, B. (1993). Redundancy and the justification for fourth normal form in relational databases. *International Journal of Foundations of Computer Science* 4(4): 355–365.
- Weddell, G. (1992). Reasoning about functional dependencies generalized for semantic data models. *ACM Transactions on Database Systems* 17(1): 32–64.
- Weihl, W. (1984). *Specification and Implementation of Atomic Data Types*. Ph.D. thesis, Department of Computer Science, Massachusetts Institute of Technology, Cambridge, MA.
- Whalen, G., Garcia, M., DeLuca, S., and Thompson, D. (2001). *Microsoft SQL Server 2000 Performance Tuning Technical Reference*. Microsoft Press, Redmond, Washington.
- Widom, J., and Ceri, S. (1996). *Active Database Systems*. Morgan Kaufmann, San Francisco.
- Wong, E., and Youssefi, K. (1976). Decomposition—A strategy for query processing. *ACM Transactions on Database Systems* 1(3): 223–241.
- XML (1998). Extensible Markup Language (XML) 1.0. <http://www.w3.org/TR/REC-xml>.
- XMLSchema (2000a). XML Schema, part 0: Primer. <http://www.w3.org/TR/xmlschema-0/>.
- XMLSchema (2000b). XML Schema, parts 1 and 2. <http://www.w3.org/XML/Schema>.
- XPath (1999). XML path language (XPath), version 1.0. <http://www.w3.org/TR/xpath/>.
- XQuery (2004). XQuery 1.0: An XML query language. Eds: S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, T. Robie, and T. Simeon. <http://www.w3.org/TR/xquery>.
- XSLT (1999). XSL transformations (XSLT), version 1.0. <http://www.w3.org/TR/xslt>.
- Zaniolo, C. (1983). The database language GEM. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, CA, 423–434.
- Zaniolo, C., and Melkanoff, M. (1981). On the design of relational database schemata. *ACM Transactions on Database Systems* 6(1): 1–47.
- Zhao, B., and Joseph, A. (2000). XSet: A lightweight XML search engine for Internet applications. <http://www.cs.berkeley.edu/~ravenben/xset/>.



# Index

---

- | in XPath, 623
- in SQL, 152
- ≠ in indexing, 443
- \_ in SQL, 154
- # in SQLJ, 303
- % in SQL, 154
- &gt and &lt in XPath, 622
- \* in SQL, 152
- + in Java and JDBC, 297
- : to specify
  - host variable in embedded SQL, 270
  - host variable in SQLJ, 305
- ? parameter
  - in dynamic SQL, 288
  - in JDBC, 297
  - in ODBC, 309
- abort
  - definition of, 22
  - in embedded SQL, 274
  - in JDBC, 302
  - in ODBC, 313
- abort message. *See* two-phase commit protocol
- abort record. *See* two-phase commit protocol
- ABSOLUTE row selector, 280
- absolute XPath expression, 618
- abstract data type, 552
- Abstract Object Data Model, 543
- acceptance test, 518
- access control in SQL, 63, 176
- access error in SQL, 301
- access path
  - binary search as, 389
  - cost, 324
  - covering relational operator, 389
  - definition of, 321, 389
  - file scan as, 389
  - index as, 346, 389
  - in query execution plan, 267
  - selectivity of, 390
- ACID Properties
  - definition of, 24
  - of distributed transactions, 477
  - related to correctness, 25
  - of a transaction processing system, 25
- active database, 251
- actor. *See* UML
- AFTER triggers, 258
- after image, 475, 476
- aggregate functions. *See* SQL
- aggregation. *See* SQL
- ALL operator, 162
- ALLOCATE DESCRIPTOR statement, 290
- alpha test, 518
- ALTER TABLE statement, 60
- anomaly
  - deletion, 194
  - dirty read, 461, 465
- insertion, 194
- lost update, 24, 461
- nonrepeatable read, 461, 465
- phantom, 463, 465
- update, 194
- write skew, 468
- anonymous type in XML Schema, 606
- ANY operator, 162
- API
  - JDBC as, 294
  - ODBC as, 307
- application programmer, 8
- arity of a relation, 35
- Armstrong's Axioms
  - augmentation, 201
  - definition of, 202
  - reflexivity, 201
  - soundness and completeness, 202
  - transitivity, 202
- association in UML, 97
- asynchronous-update replication, 482
- ATOMIC in SQL/PSM, 283
- atomic commit protocol.
  - See* two-phase commit protocol
- atomicity
  - of data, 36
  - definition of, 21
  - implementation of, 472
  - statement-level, 465

- attribute  
 closure, 204  
 domain of, 37, 71  
 of entity, 71  
 naming problem, 136  
 of relation, 35, 37  
 for relationship, 73  
 set valued, 71, 537  
 augmentation rule, 201  
 authorization  
 in SQL, 63  
 autocommit  
 in JDBC, 301  
 in ODBC, 312  
 availability, 476  
 as system requirement, 6  
 AVG function, 164  
 balanced tree. *See* B tree, B<sup>+</sup> tree  
 basic type, 546  
 BCNF. *See* normal form, Boyce-Codd  
 BEFORE triggers, 257  
 before image, 472, 475  
 BEGIN DECLARE SECTION, 270  
 begin record, 473  
 beta test, 518  
 binary large object. *See* blob  
 binary search  
 as access path, 389  
 of sorted file, 333  
 bitmap index  
 definition of, 371  
 for join, 373  
 for star join, 400  
 black box test, 515  
 blob, 541  
 blocking. *See* two-phase commit protocol  
 Boyce-Codd normal form.  
*See* normal form  
 B tree, 354  
 B<sup>+</sup> tree  
 as access path, 389  
 as balanced tree, 354  
 definition of, 353  
 deletions in, 357  
 fillfactor in, 358  
 insertions in, 355  
 as main index, 353  
 range search on, 353  
 as secondary index, 353  
 sibling, 358  
 sibling pointer, 353  
 splitting pages, 355  
 bucket  
 definition of, 360  
 level in extendable hashing, 366  
 splitting  
 in extendable hashing, 364  
 in linear hashing, 368  
 bulk insertion. *See* insertion  
 cache, 475  
 buffer pools, 431  
 clean page, 430  
 controller, 328  
 definition of, 430  
 dirty page, 430  
 disk, 325  
 hit, 325, 431  
 least-recently-used algorithm, 430  
 miss, 431  
 most-recently-used algorithm, 431  
 page replacement algorithm, 430  
 prefetching, 329, 432  
 procedure, 430  
 write-back, 329  
 write-gathering, 329  
 CALL statement, 285  
 CallableStatement class  
 in JDBC, 302  
 call-level interface  
 definition of, 268  
 JDBC as, 294  
 ODBC as, 307  
 in SQL:1999, 268  
 candidate key  
 compared with search key, 337  
 definition of, 42  
 in foreign-key constraint, 45  
 specifying in SQL, 47  
 in XML, 610  
 CARDINALITY  
 function in SQL:2003, 562  
 cardinality constraint  
 in E-R model, 76  
 cardinality of a relation, 35  
 Cartesian product operator, 135  
 catalog functions in ODBC, 312  
 catalog in SQL, 62  
 catch clause in Java, 301  
 CDATA, 583  
 chained transactions  
 in SQL, 274  
 CHECK clause, 49  
 checkpoint record, 473  
 checksum, 325  
 child  
 in XML, 571  
 chunk, 326  
 class  
 in CODM, 545  
 extent of, 545  
 in UML, 96  
 Class class in JDBC, 295  
 class diagram  
 in UML, 96  
 classification hierarchy, 79  
 CLI. *See* call-level interface  
 closed-world assumption, 145  
 close() method in JDBC, 297  
 clustered index  
 B<sup>+</sup> tree as, 353  
 definition of, 340  
 ISAM index as, 350  
 as main index, 340  
 range search on, 341  
 cluster in SQL, 63  
 coding techniques, 521  
 CODM  
 class in, 545

domain of type in, 547  
introduction to, 543  
IsA relationship in, 545  
oid in, 544  
subclass in, 545  
subtype in, 546  
type in, 546  
value in, 544  
cohort. *See* two-phase commit protocol  
collection value, 550  
column. *See* attribute in relational table, 14, 35  
column-accessor method in SQLJ, 305  
comments  
  in code, 521  
  in SQL, 152  
  in XML, 572  
commit  
  definition of, 22  
  in embedded SQL, 274  
  in JDBC, 302  
  in ODBC, 313  
**COMMIT AND CHAIN**  
  statement, 274  
commit message. *See* two-phase commit protocol  
commit() method in JDBC, 302  
commit order  
  with distributed databases, 481  
  with strict two-phase locking, 460  
  with two-phase commit, 481  
commit protocol. *See* two-phase commit protocol  
commit record. *See* two-phase commit protocol  
**COMMIT statement**, 274  
commuting operations  
  for read and write operations, 456  
completeness  
  of Armstrong's axioms, 202  
  of MVD inference rules, 235  
complete record. *See* two-phase commit protocol  
composition  
  in UML, 104  
compositor, 597  
Conceptual Object Data Model. *See* CODM  
conceptual schema, 32  
concurrency control  
  definition of, 458  
  for distributed systems, 480  
condition handler in SQL/PSM, 283  
configuration management, 9  
conflict  
  in asynchronous replication, 483  
  of lock requests, 459  
conflict table  
  for intention locking, 470  
  for two-phase locking, 459  
**CONNECT statement**, 273  
Connection class in JDBC, 294  
consistency  
  definition of, 21  
  destroyed by concurrent execution, 22  
  mutual, 482  
  related to integrity constraints, 20  
  strong mutual, 482  
  weak mutual, 482  
consistency constraint. *See* integrity constraint  
constraint. *See* integrity constraint  
adding or dropping in SQL, 60  
candidate key, 42  
domain, 37  
foreign-key, 38  
inclusion dependency, 45  
interrelational, 38  
intra-relational, 38  
key, 41  
primary key, 42  
reactive, 56  
semantic, 40, 45  
in SQL, 47  
static, 40  
superkey, 42  
and triggers, 57  
type, 37  
coordinator. *See* two-phase commit protocol  
correctness. *See* integrity constraint  
  related to  
    ACID properties, 25  
    serial execution, 22, 455  
    Serializable execution, 24  
correlated nested query, 158  
correlated nested subquery, 445  
correspondence  
  many-to-many, 77  
  many-to-one, 77  
  one-to-many, 77  
  one-to-one, 77  
covering index. *See* index-only queries  
covering of relational operators, 389  
crash  
  definition of, 472  
**CREATE ASSERTION**  
  statement, 49  
**CREATE DOMAIN statement**, 53  
**CREATE INDEX statement**, 339  
**CREATE SCHEMA statement**, 62  
**createStatement()**  
  method in JDBC, 296  
**CREATE TABLE statement**, 46

- CREATE TRIGGER**  
statement, 58
- CREATE VIEW statement**,  
59, 174
- critical path**, 519
- cross product operator.** *See*  
Cartesian product  
operator
- CURRENT\_DATE**, 118
- current node in XPath**, 618
- cursor**  
dynamic, 310  
in dynamic SQL, 293  
in embedded SQL, 277  
and impedance  
mismatch, 542  
**INSENSITIVE**, 277  
in JDBC, 297  
key\_set driven, 310  
in ODBC, 309  
static, 310  
updatable, 279  
updating and deleting  
through, 279, 299
- CURSOR STABILITY**  
definition of, 467
- cylinder of disk**, 322
- data**  
self-describing, 567  
semistructured, 569
- database.** *See relational*  
**database**  
active, 251  
definition of, 3  
dump. *See* dump  
global, 477  
instance, 38  
query language  
definition of, 127  
relational algebra as,  
128  
SQL as, 147  
schema, 38
- database administrator**, 9
- database designer**, 8
- database management**  
system, 4
- data definition language.**  
*See SQL*  
definition of, 33  
of SQL, 46
- data independence**  
conceptual, 33  
physical, 33
- data manipulation**  
language. *See SQL*  
definition of, 34  
of SQL, 147
- data mining**, 11, 245
- data model.** *See data*  
definition language;  
data manipulation  
language; storage  
definition language  
conceptual schema, 32,  
33  
constraints, 33  
definition of, 33  
external schema, 33  
physical schema, 31  
view abstraction level, 33
- data partitioning.** *See*  
partitioning
- data warehouse**, 10, 178
- DDL.** *See data definition*  
**language**
- deadlock**  
definition of, 462  
detection  
with timeout, 462  
with waits\_for graph,  
462
- distributed**, 481  
global, 480
- decision support system**, 9
- declarative language**, 127
- DECLARE CURSOR**  
statement, 277
- deferred consideration of**  
triggers, 253
- deferred mode for integrity**  
constraints, 56, 285
- degree of relationship**, 75
- DELETE statement**, 184
- deletion anomaly**, 194
- denormalization**, 244, 440
- dense index**, 342
- dependency**  
functional, 198  
join, 229  
multivalued, 230
- Dependency Chart**, 519
- DESCRIBE INPUT statement**,  
290
- DESCRIBE OUTPUT**  
statement, 290
- descriptor area in dynamic**  
SQL, 290
- Design Document**  
definition of, 509  
format of, 512  
readers, 509
- design review**, 514
- diagnostics area**, 271
- Diagnostics Pack for Oracle**,  
447
- DIAGNOSTICS SIZE**, 274
- directory in extendable**  
hash index, 363
- dirty read**  
definition of, 461
- DISCONNECT statement**,  
273
- disjunctive normal form**,  
391
- disk**  
block, 325  
cache, 325  
chunk, 326  
cylinder, 322  
page, 325  
platter, 322  
read/write head, 322  
rotational latency, 323  
sector, 322  
seek time, 323  
striping, 326  
track, 322  
transfer time, 323
- distributed transaction**  
definition of, 477  
non-ACID properties,  
480, 484
- division operator**  
in relational algebra, 144

- in SQL, 160
- DML. *See data manipulation language*
- document object model, 615
- document type definition, 570, 582
- DOM, 615
- domain
  - of attribute, 36, 37, 71
  - constraint, 37
  - definition of, 14
  - specifying in SQL, 48
  - of type, 547
  - user defined, 37
- done message. *See two-phase commit protocol*
- driver
  - in JDBC, 294
  - in ODBC, 307
- driver manager
  - in JDBC, 294
  - in ODBC, 307
- DriverManager** class in
  - JDBC, 294
- DROP ASSERTION** statement, 61
- DROP COLUMN** statement, 60
- DROP CONSTRAINT** statement, 61
- DROP DOMAIN** statement, 61
- DROP INDEX** statement, 340
- DROP SCHEMA** statement, 62
- DROP TABLE** statement, 61
- DROP VIEW** statement, 176
- DTD, 570, 582
- dump
  - of database, 476
  - fuzzy, 477
- durability
  - definition of, 22
  - implementation of, 472
- dynamic constraint, 40
- DYNAMIC cursor**, 310
- dynamic parameter, 288
- dynamic SQL
  - ? parameter in, 288
  - cursor in, 293
  - definition of, 286
  - descriptor area, 290
  - prepared statements in, 287
  - SQLDA, 290
  - stored procedures in, 293
- dynaset, 310
- ECA rule, 251
- embedded SQL
  - : in, 270
  - abort in, 274
  - commit in, 274
  - cursor in, 277
  - declarations in, 270
  - definition of, 147, 268
  - direct execution of SQL, 267
- does not support two-phase commit, 276
- host language variables
  - in, 270
- integrity constraints in, 285
- isolation levels in, 274
- in Java, 303
- status processing in, 271
- stored procedures in, 282
- transactions in, 274
- empty element. *See XML*
- entailment
  - definition of, 200
  - of join dependencies, 231
- entity
  - attribute, 71
  - definition of, 70
  - domain of attribute, 71
  - key constraint of, 72
  - translation into relational model, 86
  - type, 71
- entity instance
  - definition of, 70
- Entity-Relationship Model.**
  - See E-R Model*
- equality search
  - on B<sup>+</sup> trees, 353
  - definition of, 333
  - on hash indices, 360
  - on indexed files, 337
  - on sorted files, 333
- equi-join operator, 138
- equivalent schedule. *See schedule*
- E-R Model**
  - attribute in, 71
  - definition of, 70
  - entity in, 70
  - E-R diagram, 72
  - identifying relationship in, 84
  - IsA hierarchy in, 90
  - and join dependency, 229
  - key in, 72
  - and object databases, 122
  - participation constraint, 81
  - relationship attribute, 73
  - relationships in, 73
  - for Student Registration System, 111
  - translation into relational model
    - entity, 86
    - is-a hierarchy, 90
    - participation constraints, 92
    - relationships, 88
    - weak entity in, 84
  - error processing. *See status processing*
  - Error Report Form, 517
  - ER/Studio, 123
  - ERwin, 123
  - ESCAPE clause in SQL**, 154
  - escape syntax
    - in JDBC, 302
    - in ODBC, 313
  - event
    - causing a trigger to fire, 58, 251
    - as request by a transaction, 251, 254

- event-condition-action rule, 251  
**EXCEPT**  
 function in SQL:2003, 562  
 exception handling in JDBC, 300  
 exclusive part-of relationship  
 in conceptual modeling, 83  
**EXEC SQL** as prefix in embedded SQL, 269  
**EXECUTE** statement, 288  
**EXECUTE IMMEDIATE** statement, 289  
**executeQuery()** method in JDBC, 296  
**executeUpdate()** method in JDBC, 296  
**EXISTS** operator in SQL, 159  
 extendable hash indexing, 363  
 extensible markup language. *See* XML  
 extent of class, 545  
 external schema, 33  
 external sorting, 380  
**EXTRACT**, 118  
 fan-out of index, 349  
**FETCH** statement, 277, 280  
 Fifth Normal Form, 233  
 file scan as access path, 389  
 file structures, 31  
 fillfactor  
   in B<sup>+</sup> tree, 358  
   definition of, 335  
   in hash index, 362  
   in ISAM index, 352  
**FIRST** row selector, 280  
 force write, 478  
 foreign-key  
   definition of, 43  
   specifying in SQL, 53  
   violation of constraint, 56  
   in XML, 612  
 foreign key constraint  
   definition of, 38  
   and indices, 439  
   in translating E-R model, 89  
   and triggers, 57  
**forName()** method in JDBC, 295  
 fourth normal form. *See* normal form  
 fully inverted index, 340  
 functional dependency  
   and Armstrong's axioms, 202  
   and Boyce-Codd normal form, 208  
   definition of, 198  
   and entailment, 200  
   as an integrity constraint, 198  
   satisfaction of, 198  
   and third normal form, 210  
   trivial, 201  
   and update anomalies, 199  
 fuzzy dump, 477  
 Gantt chart, 519  
 generalization in UML, 101  
**getConnection** method in JDBC, 296  
**GET DESCRIPTOR** statement, 291  
**GET DIAGNOSTICS** statement, 272  
**getMetaData()** method in JDBC, 300  
 glass box test, 516  
 global deadlock. *See* deadlock  
 global element in XML Schema  
   definition of, 603  
   referencing of, 607  
 global integrity constraint.  
   *See* integrity constraint  
 global serialization. *See* serializable  
 global transaction.  
   *See* distributed transaction  
**GRANT** statement  
   definition of, 63  
 granularity of triggers, 254  
**GROUP BY** clause, 167  
 grouping in SQL, 167  
 guard in UML state diagram, 511  
 hash index  
   as access path, 389  
   bucket splitting  
     in extendable, 364  
     in linear, 368  
   definition of, 360  
   directory in extendable, 363  
   does not support range or partial key search, 361  
   dynamic, 363  
   equality search in, 360  
   extendable, 363  
   fillfactor in, 362  
   linear, 368  
   static, 362  
   hash-join, 398  
 HAVING clause, 168  
 heap file  
   definition of, 329  
   efficiency of access, 330  
 hint. *See* tuning  
 histograms. *See* tuning  
 host language, 267  
 host language variables  
   in embedded SQL, 270  
   in JDBC, 297  
   in ODBC, 309  
 ID attribute type in XML, 583  
 identifying relationship in E-R, 84  
**IDREF**, 583  
**IDREFS**, 583

- immediate consideration of triggers, 253  
immediate mode for integrity constraints, 56, 285  
impedance mismatch definition of, 542 and ODMG, 542 in SQL, 277, 542 and SQL:1999/2003, 542  
**IN** operator, 157  
inclusion dependency, 45  
incremental development, 523  
**index.** *See B tree, B<sup>+</sup> tree, hash index, ISAM index*  
automatic creation of, 339  
bitmap, 371  
choosing an, 373  
clustered, 340  
definition of, 321, 337  
dense, 342  
entry, 337  
fan-out of, 349  
hash, 360  
integrated, 338  
inverted, 340  
join, 372, 399  
leaf entry, 348  
leaf level, 349  
location mechanism, 347  
multilevel, 347, 348  
partial key search on, 345  
primary, 340  
search key of, 337  
secondary, 340  
separator, 348  
separator level, 349  
simple example of, 18  
sparse, 342  
tree, 347  
two-level, 348  
unclustered, 340  
**index-only queries**, 346, 436  
**index sequential access** method. *See ISAM index*  
**Information\_Schema** in SQL, 63  
inheritance in object databases, 560  
**inner join** in SQL, 170  
**INSENSITIVE cursor.** *See cursor*  
**INSERT statement**, 182  
**insertion** anomaly, 194  
bulk, 183  
**instance** of database, 38  
of object database, 549  
of relation, 35  
**instance-document**, 587  
**integrated index**, 338  
**integration test**, 515  
**integrity constraint** checking in decomposed tables, 211  
deferred mode in, 56, 285  
definition of, 20, 38  
dynamic, 40  
in embedded SQL, 285  
enforced by triggers, 255  
in E-R model, 70  
foreign-key, 43  
functional dependencies as, 198  
global, 477  
**immediate mode** in, 56, 285  
interrelational, 38  
intra-relational, 38  
key, 41  
local, 477  
primary key, 42  
referential integrity, 43  
related to business rules, 20  
in the relational model, 37  
semantic, 40, 45  
static, 40  
and triggers, 57  
**intention lock.** *See lock*  
**INTERSECT** function in SQL:2003, 562  
**intersection anomaly**, 239  
**intersection operator**, 133  
**inverted index**, 340  
**ISAM index** definition of, 350  
deletions from, 351  
insertions in, 352  
searching, 351  
**IsA relationship** in CODM, 545  
in E-R model, 79, 90  
and shortcoming of SQL, 540  
**IS A SET** predicate in SQL:2003, 562  
**IS lock**, 469  
**isolation** achieved by serializable schedules, 456  
definition of, 24  
related to serial schedules, 24  
statement-level, 465  
**isolation levels** **CURSOR STABILITY.** *See CURSOR STABILITY*  
definition of, 465  
in embedded SQL, 274  
in JDBC, 301  
locking implementations of, 465  
in ODBC, 312  
**READ COMMITTED.** *See READ COMMITTED*  
**READ UNCOMMITTED.** *See READ UNCOMMITTED*  
**REPEATABLE READ.** *See REPEATABLE READ*  
**SERIALIZABLE.** *See SERIALIZABLE*  
**SNAPSHOT.** *See SNAPSHOT*

- IS VALID INSTANCE OF
    - `predicate in SQL/XML`, 626
  - iterator
    - in SQLJ, 305
  - IX lock, 469
  - Java Transaction API, 302
  - Java Transaction Service. *See* JTS
  - JDBC
    - + concatenation symbol in, 297
    - ? parameters in, 297
    - abort in, 302
    - autocommit in, 301
    - CallableStatement
      - class, 302
      - called by SQLJ, 303
    - Class class, 295
    - close() method, 297
    - commit() method, 302
    - Connection class, 294
    - createStatement()
      - method, 296
    - cursor in, 297
    - description of, 294
    - does not support two-phase commit, 302
    - driver in, 294
    - DriverManager class, 294
    - driver manager in, 294
    - escape syntax in, 302
    - exception handling in, 300
    - executeQuery() method, 296
    - executeUpdate()
      - method, 296
    - forName() method, 295
    - forward-only result set, 299
    - getConnection()
      - method, 296
    - getMetaData() method, 300
    - isolation levels in, 301
  - PreparedStatement
    - class, 297
    - prepared statements in, 297
  - prepareStatement()
    - method, 297
  - ResultSet class, 294
  - ResultSetMetaData
    - class, 300
  - rollback() method, 302
  - scroll-insensitive result set, 299
  - scroll-sensitive result set, 299
  - setAutoCommit()
    - method, 301
  - setTransactionIsolation()
    - method, 301
  - setXXX() functions, 297
  - Statement class, 294
  - status processing in, 300
  - stored procedures in, 302
  - transactions in, 301
  - updatable cursors in, 299
- JDBC-ODBC bridge, 294
- join. *See index*
  - computing
    - with block-nested loops, 393
    - with index-nested loops, 394
    - with nested loops, 393
  - dependency
    - definition of, 229
    - and lossless, 230
    - satisfaction of, 230
  - hash, 398
  - left outer, 142
  - operator
    - definition of, 137
    - equi-join, 138
    - natural join, 140
    - theta-join, 138
  - order, 450
  - outer, 142
  - right outer, 142
  - sort-merge, 396
  - star, 399
- join index. *See index*
- JTA, 302
- JTS
  - supports two-phase commit, 302
  - as TP monitor, 302
- key. *See search key of index in E-R Model*
  - on entity type, 72
  - on relationship type, 75
- in relational model, 41
- in SQL
  - candidate, 42
  - constraint on relation, 41
  - foreign, 43
  - primary, 42
- in XML, 610
- KEYSET\_DRIVEN cursor
  - in embedded SQL, 279
  - in JDBC, 299
  - in ODBC, 310
- LAST row selector, 280
- latch, 465
- latency of disk, 323, 334
- leaf level of index tree, 349
- left outer join
  - in SQL, 170
- legal instance
  - of database, 38
  - of relation, 37
  - of schema, 198, 230
- lifetime of system, 7
- LIKE predicate in SQL, 154
- linear hashing, 368
- local integrity constraint.
  - See integrity constraint*
- location mechanism
  - definition of, 338
  - for tree index, 347
- lock
  - conflict table, 459, 470
  - exclusive, 469
  - granularity
    - definition of, 469
  - intention, 470

- intention exclusive, 469  
intention shared, 469  
IS, 469  
IX, 469  
page, 463  
read, 458, 469  
shared, 469  
shared intention  
  exclusive, 470  
SIX, 470  
table, 463  
tuple, 463  
write, 458, 469
- locking  
  protocol  
    for READ COMMITTED  
      isolation level, 466  
    for READ UNCOMMITTED  
      isolation level, 466  
    for REPEATABLE READ  
      isolation level, 466  
    for SERIALIZABLE  
      isolation level, 466  
  in relational databases, 463
- log  
  definition of, 472  
  placement of, 448  
  write-ahead, 475  
  used to implement  
    atomicity and  
    durability, 472
- log buffer, 475
- lossless decomposition  
  definition of, 213  
  and join dependency, 230
- lossy decomposition, 213
- lost update  
  in asynchronous  
    replication, 483  
  definition of, 461  
  eliminated with  
    SNAPSHOT isolation,  
    468  
  example of, 24
- machine learning, 245
- main index  
  B<sup>+</sup> tree as, 353  
  definition of, 340  
  ISAM index as, 350  
  is clustered index, 340
- many-to-many  
  correspondence  
    in E-R, 77
- mass storage, 322
- materialized view, 177
- MAX function, 164
- MEMBER OF  
  predicate in SQL:2003,  
    562
- merging, 380
- method signature, 549
- MIN function, 164
- minimal cover, 222
- minimality property of key  
  constraint, 41
- mirrored disk, 327, 476
- module test, 515
- MTS  
  supports two-phase  
    commit, 313
- multidatabase  
  definition of, 477
- multilevel index, 347, 348
- multiple attributes in search  
  key, 344
- multiplicity constraint in  
  UML, 98
- multiplicity of an attribute  
  in UML, 96
- multiset  
  returned by relational  
    operators, 151  
  in SQL:2003, 561
- MULTISET  
  function in SQL:2003,  
    562
- multivalued dependency  
  definition of, 230  
  splitting the left-hand  
    side of, 239
- multiversion concurrency  
  control  
    SNAPSHOT isolation, 467
- mutator method. *See*  
  SQL:1999/2003
- mutual consistency. *See*  
  consistency
- MVD. *See* multivalued  
  dependency
- named type in XML  
  Schema, 606
- namespace. *See* XML  
  namespace
- naming problem for  
  attributes, 136
- natural join operator, 140
- navigation axis in XPath,  
  620
- nested query  
  definition of, 157  
  in FROM-clause, 162  
  and tuning, 436  
  in WHERE-clause, 157
- NEXT row selector, 280, 305
- next() function in JDBC,  
  297
- non-exclusive part-of  
  relationship  
  in conceptual modeling,  
    83
- nonrepeatable read, 461
- normal form  
  Boyce-Codd, 207  
  algorithm for, 219  
  definition of, 208  
  properties of, 211  
  definition of, 207  
  and E-R diagrams, 87  
  fifth, 233  
  first, 207  
  fourth, 207, 228  
  of relation schema, 197  
  second, 207  
  third, 207  
    algorithm for, 221  
    definition of, 210  
    properties of, 211
- normalization theory. *See*  
  normal form
- NOT in SQL, 154

- NULL in SQL  
 definition of, 48  
 and indexing, 444  
 in query processing, 181
- O<sub>2</sub>, 543  
 object  
 compared with entity, 70  
 Id, 543  
 shortcoming of SQL, 541  
 in SQL:1999/2003, 550
- Object Constraint Language  
 in UML, 96
- object databases  
 schema, 548
- object id. *See* oid
- object-relational database, 550
- observer method. *See* SQL:1999/2003
- OCL in UML, 96
- ODBC, 268  
 ? parameters in, 309  
 abort in, 313  
 autocommit in, 312  
 catalog functions in, 312  
 commit in, 313  
 cursor in, 309  
 description of, 307  
 does not support two-phase commit, 313  
 driver in, 307  
 driver manager in, 307  
 dynamic cursor in, 310  
 escape syntax in, 313  
 isolation levels in, 312  
 keyset\_driven cursor in, 310  
 prepared statements in, 309  
 RETCODE, 312  
 SQLAllocConnect(), 307  
 SQLAllocEnv(), 307  
 SQLAllocStmt(), 308  
 SQLBindCol(), 309  
 SQLColAttributes(), 312  
 SQLColumns(), 312
- SQL\_COMMIT, 313  
 SQLConnect(), 307  
 SQLDescribeCol(), 312  
 SQLDisconnect(), 308  
 SQLError(), 312  
 SQLExecDirect(), 308  
 SQLExecute(), 309  
 SQLFetch(), 310  
 SQLFreeConnect(), 308  
 SQLFreeEnv(), 308  
 SQLFreeStmt(), 308  
 SQLGetData(), 310  
 SQLPrepare(), 309  
 SQL\_ROLLBACK, 313  
 SQLSetConnection-Option(), 312  
 SQLSetStmtOption(), 310  
 SQLTables(), 312  
 SQLTransact(), 313  
 static cursor in, 310  
 status processing in, 312  
 stored procedures in, 313  
 transactions in, 312
- ODMG  
 data model of, 543  
 and impedance mismatch, 542
- oid  
 compared with primary key, 543  
 definition of, 543
- OLAP, 10, 178, 441
- OLTP, 10, 441
- one-to-many correspondence  
 in E-R, 77
- one-to-one correspondence  
 in E-R, 77
- online analytic processing.  
*See* OLAP
- online transaction processing. *See* OLTP
- OPEN statement, 277
- Oracle Designer, 123
- ORDER BY clause, 170, 279
- outer join, 142. *See* join  
 in SQL, 170
- out parameter in embedded SQL, 271
- overflow page  
 in clustered index, 342  
 in hash index, 362  
 in ISAM index, 352  
 in sorted file, 335
- page buffer, 475
- page id in index entry, 337
- page lock. *See* lock
- page number, 329
- page of disk data, 325
- parallel query processing, 448
- parameter passing  
 in dynamic SQL, 288  
 in embedded SQL, 270  
 in JDBC, 297  
 in ODBC, 309
- parsed character data, 582
- partial key search  
 on B<sup>+</sup> tree, 353  
 definition of, 345  
 not supported by hash index, 361
- partial sorting, 380
- participation constraint, 81
- partitioning  
 horizontal, 442  
 to increase concurrency, 447  
 for tuning, 442  
 using ISA hierarchy, 80  
 vertical, 442
- part-of relationship  
 in conceptual modeling, 83  
 in UML, 104
- path expression  
 definition of, 540  
 in SQL:1999/2003, 554
- PCDATA, 582
- Persistent Stored Modules.  
*See* PSM language
- PERT chart, 519
- phantom  
 definition of, 463
- physical schema, 31

- pipeline**
  - in query processing, 414, 421
- placeholder**, 288
- platter of disk**, 322
- PowerDesigner**, 123
- preamble of procedure**, 521
- precompiler for embedded SQL**, 268
- precondition of procedure**, 513
- predicate**
  - definition of, 14
- prefetching**. *See* cache
- preparation of SQL statement**
  - definition of, 267
  - in dynamic SQL, 287
  - in embedded SQL, 268
  - in JDBC, 297
  - in ODBC, 309
  - in stored procedures, 285
- PREPARE statement**, 287
- prepared record**. *See* two-phase commit protocol
- prepared state**. *See* two-phase commit protocol
- PreparedStatement class**
  - in JDBC, 297
- prepared statements**
  - in dynamic SQL, 287
  - in embedded SQL, 268
  - in JDBC, 297
  - in ODBC, 309
  - in stored procedures, 285
- prepare message**. *See* two-phase commit protocol
- prepareStatement()**
  - method in JDBC, 297
- primary copy replication**.
  - See* replication
- primary key**
  - definition of, 42
  - and insertion anomaly, 210
  - specifying in SQL, 47
- in XML**, 610
- PRIMARY KEY**
  - and foreign key constraint, 53
- primitive value**, 544, 550
- PRIOR row selector**, 280
- procedural language**, 127
- procedure cache**. *See* cache procedures
  - see stored procedures, 282
- processing instruction in XML**, 572
- profile in UML**, 97
- projection of MVD**, 232
- Project Management Plan**, 524
- project manager**, 8, 518
- project meetings**, 519
- project operator**
  - computing, 384
  - definition of, 131
- Project Plan**, 518
- protocol**
  - two-phase commit. *See* two-phase commit protocol
  - two-phase locking. *See* two-phase locking
- PSM language**. *See* stored procedures
  - and SQL-92, 283
  - and SQL:1999/2003, 552
  - and triggers, 257
- QA test**, 515
- quantified predicate in SQL:1999**, 163
- query**
  - in database, 4
  - execution plan, 128, 267
  - index-only, 346, 436
  - processing, 128
  - result, 276
- Query Analyzer for SQL Server**, 447
- query execution plan**, 380
  - cost of, 410
  - definition of, 406
  - logical, 420
- query language**, 127
  - for OLAP queries, 10
  - SQL as, 147
- query optimization**
  - choosing an index, 433
  - definition of, 128
  - index-only strategy, 436
- query optimizer**, 380
  - cost based, 406, 449
  - definition of, 405
  - rule based, 406
- query tree**
  - definition of, 410
  - left-deep, 421
- RAID systems**, 326
- range query**, 387, 435
- range search**
  - in B<sup>+</sup> tree, 353
  - on clustered index, 341
  - definition of, 333
  - with ISAM index, 351
  - not supported by hash index, 361
  - of sorted file, 335
- Rational Rose**, 123
- reactive constraint**
  - definition of, 56
  - implemented with triggers, 57
- READ COMMITTED**
  - definition of, 465
  - locking implementation of, 466
- read lock**. *See* lock
- READ UNCOMMITTED**
  - definition of, 465
  - locking implementation of, 466
- record**
  - abort, 473
  - begin, 473
  - checkpoint, 473
  - commit, 473
  - undo, 472
  - update, 472
- recovery**
  - from disk failure, 476
  - using log, 476

- reduction factor in query optimization, 418
- redundant attribute, 222
- redundant FD, 222
- reference
  - data type in SQL:1999/2003, 553, 558
  - type, 546
  - value, 544
- referential integrity, 43
- reflexivity rule, 201
- relation
  - attribute
    - definition of, 35
    - domain of, 37
    - name of, 37
  - constraint
    - candidate key, 42
    - domain, 37
    - dynamic, 40
    - foreign-key, 43
    - integrity, 37, 38
    - interrelational, 40
    - intra-relational, 40
    - key, 41
    - primary key, 42
    - semantic, 40, 45
    - superkey, 42
    - type, 37
  - definition of, 14, 35
  - instance, 35
  - schema, 37
  - tuple, 35
  - union-compatible, 133
- relational algebra
  - Cartesian product, 135
  - cross product, 135
  - description of, 128
  - division, 144
  - equi-join, 138
  - intersection, 133
  - join, 137
  - natural join, 140
  - project, 131
  - renaming, 136
  - select, 128
  - set difference, 133
  - theta-join, 138
  - union, 133
- relationship
  - attribute, 73
  - binary, 75
  - definition of, 73
  - degree of, 75
  - key of, 75
  - role
    - definition of, 73
    - name of, 73
  - schema of, 75
  - ternary, 75
- translation into relational model, 88
- type, 73
- relationship instance
  - definition of, 73
- RELATIVE row selector, 280
- relative XPath expression, 618
- reliability, 6
- renaming operator, 136
- REPEATABLE READ
  - definition of, 465
- locking implementation of, 466
- repeating groups, 441
- replica control. *See* replication
  - definition of, 482
- replication
  - asynchronous-update, 482
  - definition of, 482
  - group, 483
  - mutual consistency. *See* consistency
  - primary copy, 484
  - read-one/write-all, 482
  - synchronous-update, 482
- Requirements Document, 489, 493
- response time, 7
- result set
  - definition of, 276
  - forward-only, 299
  - scroll-insensitive, 299
  - scroll-sensitive, 299
- ResultSet class in JDBC, 294
- ResultSetMetaData class in JDBC, 300
- RETCODE, 312
- revision history, 521
- REVOKE statement, 65
- rid
  - definition of, 329
  - in index entry, 337
- right outer join
  - in SQL, 170
- risk management plan, 525
- role
  - in E-R model, 73
- rollback
  - definition of, 22
  - in JDBC, 302
  - in ODBC, 313
  - using log, 473
- ROLLBACK AND CHAIN statement, 274
- ROLLBACK statement. *See* SQL
- root element in XML, 571

- root node in XPath, 616  
 rotational latency of disk, 323  
**row.** *See tuple*  
 in relational table, 14, 35  
**row id.** *See rid*  
**row-level granularity**, 254  
**ROW type constructor** in SQL:1999/2003, 551  
**ROW value constructor** in SQL:1999/2003, 551  
**row type** in SQL:1999/2003, 551  
 run in external sort, 381  
**safe triggers**, 264  
**schedule.** *See serializable*  
 definition of, 23  
 equivalence of, 456  
 non-serializable, 455  
 serial, 22, 455  
**schema**  
 conceptual, 32, 33  
 of database, 38  
 of entity type, 72  
 external, 33  
 legal instance of, 198, 230  
 of object database, 548  
 physical, 31  
 of relation, 35, 37  
 of relationship, 75  
 specified by CREATE TABLE command, 46  
 in SQL, 62  
**schema valid document**, 587  
**SDL.** *See storage definition language*  
**search key of index**  
 compared with candidate key, 337  
 definition of, 337  
 with multiple attributes, 344  
**secondary index**  
 B<sup>+</sup> tree as, 353  
 definition of, 340  
 is unclustered index, 340  
**sector of disk**, 322  
**security**  
 definition of, 7  
**seek time of disk**, 323  
**SELECT DISTINCT**  
 statement, 151  
**selection-condition**  
 in relational algebra, 129  
 in XPath queries, 621  
**selectivity of access path**, 390  
**select operator**  
 computing, 386  
 definition of, 128  
**SELECT statement**  
 definition of, 148  
 expressions in, 152  
**self-describing data**, 567  
**self-referencing column**, 554  
**semistructured data**, 569  
**separator in index file**, 347  
**sequence diagram**, 503  
**Serializable**  
 in commit order, 460  
 definition of, 24, 456  
 of global transactions, 480  
 schedules, 456  
**SERIALIZABLE isolation level**  
 definition of, 465  
 locking implementation of, 466  
**serial schedule.** *See schedule set*  
 constructor in SQL, 157  
**difference operator**  
 computing, 384  
 definition of, 133  
**function in SQL:2003**, 562  
 operations in SQL, 154  
 type, 546  
 value, 544  
 valued attribute, 537  
**setAutoCommit() method**  
 in JDBC, 301  
**SET CONNECTION statement**, 273  
**SET DESCRIPTOR statement**, 291  
**SET TRANSACTION statement**, 274  
**setTransactionIsolation() method** in JDBC, 301  
**set-valued attribute**  
 in E-R, 71  
**shared intention exclusive lock**, 470  
**sibling pointers in B<sup>+</sup> tree.**  
*See B<sup>+</sup> tree*  
**signature**  
 of method, 549  
**SIX lock**, 470  
**SLI.** *See statement-level-interface*  
**slot number**, 329  
**snapshot**  
 returned by insensitive cursor, 277, 310  
**SNAPSHOT isolation**  
 definition of, 467  
**Software Engineering.** *See Design Document; Requirements Document; Specification Document; Statement of Objectives; Waterfall model*  
**coding techniques**, 521  
**database design**, 69  
**design review**, 514  
**Entity-Relationship Diagram**, 70  
**incremental development**, 523  
**methodology**, 489  
**Project Plan**, 518  
**sequence diagram**, 503  
**Standish Report**, 8  
**state diagrams**, 510  
**Test Plan Document**, 515  
**UML**, 95, 490, 510  
**use case**, 490  
**sorted file**  
 binary search of, 333

sorted file (*continued*)  
     definition of, 333  
     range search of, 335  
 sorting  
     avoidance, 443  
     external, 380  
     partial, 380  
 sort-merge join, 396  
 soundness  
     of Armstrong's axioms, 202  
     of MVD inference rules, 235  
 sparse index, 342  
 Specification Document, 490, 502, 509  
 SQL. *See* dynamic SQL; embedded SQL  
     -- to start comments, 152  
     % in pattern matching, 154  
     \* in SELECT, 152  
     access control in, 63, 176  
     access error in, 301  
     aggregate function, 164  
     aggregation, 167  
     ALL, 162  
     ALLOCATE DESCRIPTOR, 290  
     ALTER TABLE, 60  
     ANY, 162  
     AVG, 164  
     base relation, 174  
     BEGIN DECLARE SECTION, 270  
     CALL, 285  
     CHECK, 49  
     comments, 152  
     COMMIT AND CHAIN, 274  
     COMMIT, 274  
     CONNECT, 273  
     COUNT, 164, 439  
     CREATE ASSERTION, 49  
     CREATE DOMAIN, 53  
     CREATE INDEX, 339  
     CREATE SCHEMA, 62  
     CREATE TABLE, 46  
     CREATE TRIGGER, 58  
     CREATE VIEW, 59, 174

CURRENT\_DATE, 118  
 data definition language, 46  
 DECLARE CURSOR, 277  
 DELETE, 184  
 DESCRIBE INPUT, 290  
 DESCRIBE OUTPUT, 290  
 diagnostics area, 271  
 DIAGNOSTICS SIZE, 274  
 direct execution of, 267  
 DISCONNECT, 273  
 division, 160  
 DROP ASSERTION, 61  
 DROP COLUMN, 60  
 DROP CONSTRAINT, 61  
 DROP DOMAIN, 61  
 DROP INDEX, 340  
 DROP SCHEMA, 62  
 DROP TABLE, 61  
 DROP VIEW, 176  
 ESCAPE, 154  
 EXECUTE, 288  
 EXECUTE IMMEDIATE, 289  
 EXISTS, 159  
 EXTRACT, 118  
 FETCH, 277, 280  
 FOREIGN KEY, 53  
 GET DESCRIPTOR, 291  
 GET DIAGNOSTICS, 272  
 GRANT, 63  
 GROUP BY, 167, 446  
 grouping, 167  
 HAVING, 168  
 IN, 157  
 INSERT, 182  
 join in, 149  
 LIKE, 154, 444  
 MAX, 164  
 MIN, 164  
 nested query, 157, 162  
 NOT, 154  
 NULL, 48, 181  
 null value, 48, 181  
 OPEN, 277  
 ORDER BY clause, 279  
 PREPARE, 287  
 PRIMARY KEY, 47, 53  
 PUBLIC, 63  
 query sublanguage, 147

restrictions on GROUP BY and HAVING, 173  
 REVOKE, 65  
 ROLLBACK AND CHAIN, 274  
 ROLLBACK, 274  
 SELECT, 148  
     expressions in SELECT clause, 152  
     expressions in WHERE clause, 152  
 SELECT DISTINCT, 151  
 set comparison operators, 162  
 SET CONNECTION, 273  
 set constructor, 157  
 set operations, 154  
 SET TRANSACTION, 274  
 shortcomings of, 539  
 SQLERROR, 271  
 SQLSTATE, 271, 285  
 START TRANSACTION, 274  
 static, 268  
 SUM, 164  
 UNIQUE, 47  
 UPDATE, 185  
 view, 174  
     access control, 176  
     materialized, 177  
     as subroutine, 174  
     and tuning, 444  
     updatable, 187  
     update, 185  
 WHENEVER, 271  
     in pattern matching, 154

SQL-92. *See* SQL  
     as basis for SQLJ, 303  
 SQL:1999  
     history of, 147  
     new features, 46  
     triggers in, 256  
 SQL:1999/2003  
     CREATE METHOD, 552  
     EXECUTE privilege, 556  
     and impedance mismatch, 542

- mutator method, 555  
objects in, 550, 553  
observer method, 555  
REF, 553, 558  
ROW type constructor,  
    551  
self-referencing column  
    in, 554  
tuple value, 550  
typed table in, 553  
UNDER, 552  
use of path expression,  
    554  
user-defined types, 552  
**SQL:2003**, 46, 294  
    MULTISET, 561  
    UNNEST, 561  
**SQLAllocConnect()**  
    function in ODBC,  
        307  
**SQLAllocEnv()** function in  
    ODBC, 307  
**SQLAllocStmt()** function  
    in ODBC, 308  
**SQLBindCol()** function in  
    ODBC, 309  
**SQL/CLI**, 307  
**SQLColAttributes()**  
    function in ODBC,  
        312  
**SQLColumns()** function in  
    ODBC, 312  
**SQL\_COMMIT** in ODBC, 313  
**SQLConnect()** function in  
    ODBC, 307  
**SQL-connection**, 273  
**SQLDA** in dynamic SQL,  
    290  
**SQLDescribeCol()**  
    function in ODBC,  
        312  
**SQLDisconnect()** function  
    in ODBC, 308  
**SQLERROR**, 271, 312  
**SQLException** in Java, 301  
**SQLExecDirect()** function  
    in ODBC, 308  
**SQLExecute()** function in  
    ODBC, 309  
**SQLFetch()** function in  
    ODBC, 310  
**SQLFreeConnect()**  
    function in ODBC,  
        308  
**SQLFreeEnv()** function in  
    ODBC, 308  
**SQLFreeStmt()** function in  
    ODBC, 308  
**SQLGetData()** function in  
    ODBC, 310  
**SQLJ**, 303  
**SQLPrepare()** function in  
    ODBC, 309  
**SQL/PSM** language, 256  
**SQL\_ROLLBACK** in ODBC,  
    313  
SQL-schema, 62  
SQL-server, 273  
SQL-session, 273  
**SQLSetConnection-**  
    **Option()** function in  
    ODBC, 312  
**SQLSetStmtOption()**  
    function in ODBC,  
        310  
**SQLSTATE**, 271, 285  
**SQLTables()** function in  
    ODBC, 312  
**SQLTransact()** function in  
    ODBC, 313  
SQL/XML, 623  
    goal of, 615  
    as query language, 570  
Staff Allocation Chart,  
    519  
Standish Report, 8  
star join  
    computing using  
        bitmapped join  
        index, 400  
    definition of, 399  
**START TRANSACTION**  
    statement, 274  
state diagram, 510  
**Statement class** in JDBC,  
    294  
statement-level atomicity  
    and isolation, 465  
statement-level granularity,  
    254  
statement-level-interface,  
    268  
**Statement of Objectives**, 13,  
    489  
static constraint, 40  
**STATIC cursor** in ODBC,  
    310  
static SQL, 268  
statistics. *See tuning*  
status processing  
    in embedded SQL, 271  
    in JDBC, 300  
    in ODBC, 312  
stereotype in UML, 97  
storage definition language,  
    34  
storage structure  
    B<sup>+</sup> tree as, 353  
    definition of, 321  
    hash index as, 360  
    heap file as, 329  
    ISAM index as, 350  
    sorted file as, 333  
stored procedures. *See PSM*  
    language  
        advantages of, 282  
        in dynamic SQL, 293  
        in embedded SQL, 282  
        in JDBC, 302  
        in ODBC, 313  
        recompilation of, 450  
stress test, 516  
strict two-phase locking. *See*  
    two-phase locking  
string value of XPath  
    expression, 621  
striping, 326  
**Student Registration System**  
    code for registration  
        transaction, 530  
    database design for, 111  
    deliverables for, 500  
    Design Document, 111,  
        525  
    design of Registration  
        Transaction, 528  
    E-R Diagram for, 111

- Student Registration**  
 System (*continued*)  
 integrity constraints for,  
 113, 495, 526  
**Requirements Document**,  
 111, 493  
 schema for, 111  
 sequence diagram for, 503  
**Specification Document**,  
 504  
**Statement of Objectives**,  
 13  
**Test Plan Document**, 515  
 use cases for, 495  
**subclass** in CODM, 545  
**SUBMULTISET OF**  
 predicate in SQL:2003,  
 562  
**subtransaction**  
 of distributed transaction,  
 477  
**subtype** in CODM, 546  
**SUM function**, 164  
**superkey**, 42, 208  
**supertable**, 560  
**synchronous-update**  
 replication, 482  
**system**  
 architecture, 9  
 status, 9  
**system administrator**, 9  
**system analyst**, 8  
**system catalog**, 46
- table.** *See relation*  
 in relational database, 14,  
 35  
**table locking.** *See lock*  
**tag**  
 local name of, 579  
 in XML, 570  
**target namespace.** *See XML*  
 namespace; XML  
 Schema  
**task dependency** in project  
 plan, 518  
**test**  
 acceptance, 518
- alpha, 518  
 beta, 518  
 black box, 515  
 glass box, 516  
 stress, 516  
**Testing Protocol Document**,  
 517  
**Test Plan Document**  
 definition of, 515  
 as script, 517  
 theta-join operator, 138  
 third normal form. *See*  
 normal form  
 throughput, 7  
**throws clause** in Java, 301  
**timeout**  
 to detect deadlock, 462  
**TP monitor**  
 in JTS, 302  
 in MTS, 313  
 as part of a transaction  
 processing system, 5  
**track of disk**, 322  
**transaction**  
 definition of, 4  
 description of, 20  
 in embedded SQL, 274  
 global. *See distributed*  
 transaction  
 in JDBC, 301  
 in ODBC, 312  
**transaction id**, 472  
**transaction manager.** *See*  
 two-phase commit  
 protocol  
 definition of, 478  
**transaction monitor.** *See TP*  
 monitor  
**transaction processing**  
 system  
 definition of, 5  
**transfer time of disk**, 323  
**transitivity rule**, 202  
**tree index.** *See B<sup>+</sup> tree and*  
 ISAM index  
 location mechanism for,  
 347  
 two interpretations of,  
 347
- trigger**  
 activation of, 252  
 after, 254  
**AFTER**, 258  
 before, 254  
**BEFORE**, 257  
 conflicts of, 255  
 consideration of, 252  
 deferred consideration,  
 253  
 definition of, 251  
 execution of, 253  
 firing of, 251  
 granularity of, 254  
 immediate consideration,  
 253  
 to implement active  
 database, 251  
 to implement reactive  
 constraints, 57  
 instead of, 254, 261  
 and integrity constraints,  
 255  
 safe, 264  
 in SQL, 57  
 in SQL:1999, 58, 256  
**triggering graph**, 264  
**trivial functional**  
 dependency, 201  
**try clause** in Java, 301  
**tuning**  
 hint, 450  
 histograms, 449, 450  
 statistics, 449, 450  
**tuple**  
 definition of, 14, 35  
 locking. *See lock*  
 type, 546  
 value, 544, 550  
**tuple lock.** *See lock*  
**two-level index**, 348  
**two-phase commit protocol**,  
 478. *See embedded*  
 SQL; JDBC; JTS; MTS;  
 ODBC  
 abort message, 479  
 abort record, 473, 479  
 blocking, 478, 479  
 cohort, 478

- commit message, 479  
commit record, 473, 478  
complete record, 479  
coordinator, 478  
done message, 479  
prepared record, 478  
prepared state, 478  
prepare message, 478  
and serializability, 480  
uncertain period, 478,  
    479  
vote message, 478  
two-phase locking  
    definition of, 458  
    strict, 458  
type  
    basic, 546  
    in CODM, 546  
    constraint, 37  
    reference, 546  
    set, 546  
    tuple, 546  
UML, 564  
    actor, 490, 491  
    aggregation in, 104  
    association, 97  
    association class, 97  
    class diagram, 96  
    composition in, 104  
    generalization, 101  
    for modeling databases,  
        69  
    multiplicity of an  
        attribute in, 96  
    multiplicity of a role, 98  
    Object Constraint  
        Language in, 96  
    OCL in, 96  
    profile in, 97  
    sequence diagram, 503  
    state diagram, 510  
    stereotype in, 97  
    use case, 491  
    use case diagram, 490,  
        491  
uncertain period. *See*  
    two-phase commit  
    protocol
- unclustered index  
    B<sup>+</sup> tree as, 353  
    definition of, 340  
    as secondary index, 340  
undo record, 472  
Unified Modeling Language,  
    564. *See* UML  
uniform resource identifier.  
    *See* URI  
uniform resource locator.  
    *See* URL  
union-compatible relations,  
    133  
union operator  
    computing, 384  
    definition of, 133  
UNIQUE  
    key constraint, 47  
    in XML, 610  
uniqueness property of key  
    constraint, 41  
UNNEST  
    function in SQL:2003,  
        561  
UPDATE statement, 185  
    in database, 4  
update anomaly, 194  
update record, 472  
URI  
    definition of, 579  
URL, 296, 579  
use case diagram in UML,  
    490  
user, 8  
user-defined type  
    in SQL:1999/2003, 552  
USING clause, 288
- value  
    in CODM, 544  
    collection, 550  
    primitive, 544, 550  
    reference, 544  
    set, 544  
    in SQL:1999/2003, 550  
    tuple, 544, 550  
version  
    of a database, 467
- view. *See* SQL view  
    in the relational model,  
        33  
Visual Explainer for DB/2,  
    447  
vote message. *See* two-phase  
    commit protocol
- waits\_for graph, 462  
Waterfall model, 489  
weak entity in E-R, 84  
weight of an attribute, 412  
WHENEVER statement, 271  
WHERE clause. *See* SQL  
write  
    lock. *See* lock  
    skew, 468  
write-ahead log. *See* log  
write-back cache. *See* cache  
write-gathering cache. *See*  
    cache  
write lock. *See* lock
- XML  
    comments in, 572  
    definition of, 569  
XMLAGG  
    in SQL/XML, 629  
XML attribute  
    of type ID, 576, 583  
    of type IDREF, 576, 583  
    of type IDREFS, 576, 583  
    xmlns, 579  
XML document  
    schema valid, 587  
    type definition, 582  
    valid, 582  
    well-formed, 577  
XML element, 571  
    ancestor of, 571  
    attribute of, 571  
    child of, 571  
    content of, 571  
    descendant of, 571  
    empty, 572  
    parent of, 571  
XMLELEMENT  
    in SQL/XML, 627

- XMLEXTRACT
  - in SQL/XML, 630
- XMLEN
  - in SQL/XML, 628
- XML namespace
  - declaration, 579
  - default, 579
  - definition of, 577
  - prefix, 579
  - in schema document, 587
  - target namespace, 587
  - `xmns`, 579
  - `xmns`. *See* XML attribute;
  - XML namespace
- XMLPARSE
  - in SQL/XML, 631
- XML processor, 571
- XML
  - as data type in SQL/XML, 626
- XML Schema, 570
  - anonymous type in, 606
  - global element, 603, 607
  - instance-document, 587
  - named type in, 606
  - target namespace, 587
- XML stylesheet
  - definition of, 570
- processing instruction, 572
- XML tag, 571
- XPath
  - description of, 616
  - expression, 621
  - goal of, 615
  - as query language, 570
  - selection condition, 621
- XQuery
  - as query language, 570
- XSLT
  - as query language, 570